Unbound User Manual

Release 1.19.3

NLnet Labs

Apr 19, 2024

GETTING STARTED

Unbound is a validating, recursive, caching DNS resolver. It is designed to be fast and lean and incorporates modern features based on open standards.

Note: Do you love to write and know your way around DNS and Unbound? *Help us expand this documentation* and we'll compensate you for your time. Contact us at docs@nlnetlabs.nl or find us on Twitter.

Unbound runs on FreeBSD, OpenBSD, NetBSD, MacOS, Linux and Microsoft Windows, with packages available for most platforms. It is included in the standard repositories of most Linux distributions. Installation and configuration is designed to be easy. Setting up a resolver for your machine or network can be done with only a few lines of configuration.

This documentation is an open source project and is edited via text files in the reStructuredText markup language and then compiled into a static website/offline document using Sphinx and ReadTheDocs.

We always appreciate your feedback and improvements. You can submit an issue or pull request on the GitHub repository, or post a message on the Unbound users mailing list. All the contents are under the permissive Creative Commons Attribution 3.0 (CC-BY 3.0) license, with attribution to NLnet Labs.

CHAPTER

ONE

INSTALLATION

To install your own copy of Unbound you have two options: Use the version provided by your package manager, or download the source and building it yourself.

Installing via the package manager is the easiest option, and on most systems even trivial. The downside is the distributed version can be outdated for some distributions or not have all the compile-time options included that you want. Building and compiling Unbound yourself ensures that you have the latest version and all the compile-time options you desire.

If you're a first-time user we recommend installing via a package manager.

1.1 Installing with a Package Manager

Most package managers maintain a version of Unbound, although this version can be outdated if this package has not been updated recently. If you like to upgrade to the latest version, we recommend *compiling Unbound yourself*.

1.1.1 Ubuntu 22.04 LTS

Installing Unbound with the built-in package manager should be as easy as:

```
sudo apt update
sudo apt install unbound
```

This gives you a compiled and running version of Unbound ready to *be configured*.

1.1.2 macOS Big Sur

In this tutorial we make use of the Brew package installer for MacOS. Install `brew` and, if you've never used `brew` before, give their website a read.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

Then use brew to install Unbound.

```
brew install unbound
```

This gives you a compiled and running version of Unbound ready to *be configured*.

1.2 Building from source/Compiling

To compile Unbound on any system you need to have the `openssl` and `expat` libraries, and their header files. To include the header files we need to get the development version, usually called `libssl-dev` and `libexpat1-dev` respectively.

1.2.1 Ubuntu 22.04 LTS

First of all, we need our copy of the Unbound code, so we download the tarball of the latest version and untar it.

```
wget https://nlnetlabs.nl/downloads/unbound/unbound-latest.tar.gz
tar xzf unbound-latest.tar.gz
```

We'll need some tools, such as a compiler and the **make** program.

Note: During installations with the package manager, a screen will come up asking which services need to be restarted. Using the defaults for this is fine.

```
sudo apt update
sudo apt install -y build-essential
```

The library components Unbound needs are: `libssl` `libexpat`, of which we need the "dev" version. Unbound also uses `libldns`, but this is included in the tarball we've already downloaded.

```
sudo apt install -y libssl-dev
sudo apt install -y libexpat1-dev
```

We'll also need the tools to build the actual program. For this, Unbound uses **make** and internally it uses `flex` and `yacc`, which we need to download as well.

```
sudo apt-get install -y bison
sudo apt-get install -y flex
```

With all the requirements met, we can now start the compilation process in the Unbound directory. The first step here is configuring. With `./configure -h` you can look at the extensive list of configuration options for Unbound. A nice feature is that `configure` will tell you what it's missing during configuration. A common error is for the paths to the two libraries we just installed, which can be manually specified with `--with-ssl=` and `--with-libexpat=`.

```
./configure
```

When **configure** gives no errors, we can continue to actually compiling Unbound. For this Unbound uses **make**. Be warned that compiling might take a while.

```
make
```

When we have a successful compilation, we can install Unbound to make available for the machine.

```
sudo make install
```

We now have fully compiled and installed version of Unbound, and *continue to testing it*.

Please note that the default configuration file is located at `/usr/local/etc/unbound/unbound.conf` and created during the **make** step. This file contains all possible configuration options for Unbound.

1.2.2 macOS Big Sur

In this tutorial we make use of the **brew** package installer for MacOS. Install **brew** and give their website a read if you've never used brew before.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

Then we use **brew** to install **wget**.

```
brew install wget
```

We can the use **wget** to download the latest version of Unbound from repository and unpack it.

```
wget https://nlnetlabs.nl/downloads/unbound/unbound-latest.tar.gz
tar xzf unbound-latest.tar.gz
```

To compile Unbound on MacOS (or anything really), we need to install the Mac specific development tools called "Xcode". This is available on the app store and requires ~12 GB space on the hard disk. Alternatively, if you don't want multiple Gigabytes of largely unused space on your hard disk a slimmed down version also exists called the "Command Line Tools". This includes all the tools to compile on a Mac can also be installed via the terminal.

```
xcode-select --install
```

This command will open a window where the selection can be made of what to install. If you just want the Command Line Tools select this option.

To verify that Xcode is installed correctly we check that we have the **gcc** compiler by asking for the version.

```
gcc --version
```

Next we install the required libraries using **brew**. Note that when installing these **brew** will tell you the path to where it has installed the library. The default is the `/opt/homebrew/Cellar/` directory, which can become important in the **configure** step.

```
brew install openssl@1.1
brew install expat
```

With all the requirements met, we can now start the compilation process in the Unbound directory. The first step here is configuring. With `./configure -h` you can look at the extensive list of configuration options for Unbound. A nice feature is that **configure** will tell you what it's missing during configuration. A common error is for the paths to the two libraries we just installed, which can be manually specified with `--with-ssl=` and `--with-libexpat=`.

```
./configure
```

Or alternatively, when **configure** cannot find `libssl` and `libexpat` and **brew** installed them at the default directory (make sure you fill in the correct version, at the time of writing the latest version of openssl is `1.1.1k` and of libexapt is `2.3.0`).

```
./configure --with-ssl=/opt/homebrew/Cellar/openssl@1.1/1.1.1k/ \
            --with-libexpat=/opt/homebrew/Cellar/expat/2.3.0
```

When **configure** gives no errors, we can continue to actually compiling Unbound. For this Unbound uses **make**. Be warned that compiling might take a while.

```
make
```

When we have a successful compilation, we can install Unbound to make available for the machine.

```
sudo make install
```

We now have fully compiled and installed version of Unbound, and can *continue to testing it*.

1.3 Testing

A simple test to determine if the installation was successful is to invoke the **unbound** command with the *-V* option, which is the "version" option. This shows the version and build options used, as well as proving that the install was successful. You may have to use sudo to run this, depending on the installation.

```
unbound -V
```

If all the previous steps were successful we can continue to configuring our Unbound instance.

Another handy trick you can use during testing is to run Unbound in the foreground using the *-d* option and increase the verbosity level using the *-v* option multiple times. This allows you to see steps Unbound takes and also where it fails. Another useful, more detailed trick in combination with the foreground is to make Unbound log on the foreground. To do this, the following line needs to be added to the configuration file.

```
server:
    use-syslog: no
```

Now that Unbound is installed we can *continue to configuring it*.

CHAPTER

TWO

CONFIGURATION

Unbound has a vast array of configuration options for advanced use cases, which can seem a little overwhelming at first. Luckily, all of the defaults are sensible and secure, so in a lot of environments you can run Unbound without changing any options. Below we will go through a basic, recommended configuration, but feel free to add and experiment with options as you need them.

Note: The instructions in this page assume that Unbound is already installed.

The basic configuration which you can use out of the box is shown below. To use it, you need to create a file with this configuration as its content (or copy the configuration to the default configuration file which can be found during the installation process).

```
server:
    # can be uncommented if you do not need user privilige protection
    # username: ""

    # can be uncommented if you do not need file access protection
    # chroot: ""

    # location of the trust anchor file that enables DNSSEC. note that
    # the location of this file can be elsewhere
    auto-trust-anchor-file: "/usr/local/etc/unbound/root.key"
    # auto-trust-anchor-file: "/var/lib/unbound/root.key"

    # send minimal amount of information to upstream servers to enhance privacy
    qname-minimisation: yes

    # specify the interface to answer queries from by ip-address.
    interface: 0.0.0.0
    # interface: ::0

    # addresses from the IP range that are allowed to connect to the resolver
    access-control: 192.168.0.0/16 allow
    # access-control: 2001:DB8/64 allow
```

By default the Unbound configuration uses chroot to provide an extra layer of defence against remote exploits. If Unbound is not starting because it cannot access files due to permission errors caused by **chroot**, a solution can be to enter file paths as full pathnames starting at the root of the file system (/). Otherwise, if **chroot** is not required you can disable it in the configuration file:

```
server:
    # disable chroot
    chroot: ""
```

By default Unbound assumes that a user named "unbound" exists. You can add this user with an account management tool available on your system; on Linux this is usually **useradd**). You can also disable this feature by adding `username: ""` in the configuration file:

```
server:
    # disable user privilige protection
    username: ""
```

If it is enabled, after the setup, any other user privileges are dropped and the configured username is assumed. If this user needs access to files (such as the 'trust anchor' mentioned below), these can be created by executing with `sudo -u unbound` in front of it.

Important: Unbound comes with the *unbound-checkconf(8)* tool. This tool allows you to check the config file for errors before starting Unbound. It is very convenient because if any errors are found it tells you where they are, which is particularly useful when Unbound is already running to avoid failure to restart due to a configuration error.

2.1 Testing the setup

After running the **unbound-checkconf** command to see if your config file is correct, you can test your setup by running Unbound in "debug" mode. This allows you to see what is happening during startup and catch any errors. The *unbound(8)* manpage shows that the *-d* flag will start Unbound in this mode. The manpage also shows that we can use the *-c* flag to specify the path to the configuration file, so we can use the one we created. We also recommend increasing the verbosity of the logging to 1 or 2, to see what's actually happening (*-v* or `-vv`):

```
unbound -d -vv -c unbound.conf
```

After Unbound starts normally (and you've sent it some queries) you can remove the *-v* and *-d* and run the command again. Then Unbound will fork to the background and run until you either kill it or reboot the machine.

You may run into an error where Unbound tells you it cannot bind to `0.0.0.0` as it's already in use. This is because the system resolver `systemd-resolved` is already running on that port. You can go around this by changing the IP address in the config to `127.0.0.1`. This looks like:

```
server:
    # specify the interface to answer queries from by ip-address.
    interface: 127.0.0.1
```

If you want to change this behaviour, on *this page* we show how to change the system resolver to be Unbound.

2.2 Set up Remote Control

A useful functionality to enable is the *unbound-control(8)* command. This makes starting, stopping, and reloading Unbound easier. To enable this functionality we need to add *remote-control:* to the configuration file:

```
remote-control:
    # enable remote-control
    control-enable: yes

    # location of the files created by unbound-control-setup
    # server-key-file: "/usr/local/etc/unbound/unbound_server.key"
    # server-cert-file: "/usr/local/etc/unbound/unbound_server.pem"
    # control-key-file: "/usr/local/etc/unbound/unbound_control.key"
    # control-cert-file: "/usr/local/etc/unbound/unbound_control.pem"
```

To use the **unbound-control** command, we need to invoke the **unbound-control-setup** command. This creates a number of files in the default install directory. The default install directory is `/usr/local/etc/unbound/` on most systems, but some distributions may put it in `/etc/unbound/` or `/var/lib/unbound`.

unbound-control-setup creates the cryptographic keys necessary for the control option:

```
unbound-control-setup
```

If you use a username like unbound in the configuration to run the daemon (which is the default setting), you can use **sudo** to create the files in that user's name, so that the user running Unbound is allowed to read the keys. This is also a solution if the `/usr/local/etc/unbound/` directory (or any other default directory) is write-protected, which is the case for some distributions.

```
sudo -u unbound unbound-control-setup
```

You can now control Unbound using the **unbound-control** command. Note that if your configuration file is not in the default location or not named `unbound.conf`, the name (and possibly path) need to be provided when using the command using the *-c* flag.

2.3 Set up Trust Anchor (Enable DNSSEC)

To enable DNSSEC, which we strongly recommend, we need to set up a trust anchor as it allows the verification of the integrity of the responses to the queries you send.

To help, we can use the *unbound-anchor(8)* command.

unbound-anchor performs the setup by configuring a trust anchor. This trust anchor will only serve as the initial anchor from built-in values. To keep this anchor up to date, Unbound must be able to read and write to this file. The default location that **unbound-anchor** creates this in is determined by your installation method. Usually the default directory is `/usr/local/etc/unbound/`.

Note: During the dynamic linking, this command could output an error about loading shared libraries. This is remedied by running `ldconfig` to reset the dynamic library cache.

```
unbound-anchor
```

Note that using a package manager to install Unbound, on some distributions, creates the root key during installation. On Ubuntu 22.04 LTS for example, this location is `/var/lib/unbound/root.key`. On macOS Big Sur this location

is /opt/homebrew/etc/unbound/root.key If you create the root key yourself (by using the **unbound-anchor** command), then the path to the anchor file in the configuration file should be changed to the correct location. To find out the default location you can use the **unbound-anchor** command again with the -vvv option enabled. To enable DNSSEC, we add auto-trust-anchor-file under the server clause in the configuration file.

```
server:
    # enable DNSSEC
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
```

Note that on some systems the /usr/local/etc/unbound/ directory might be write-protected.

If the **unbound-anchor** command fails due to the insufficient permissions, run the command as the correct user, here we use the user unbound as this is the default user.

```
sudo -u unbound unbound-anchor
```

This step is also important when using the chroot jail.

CHAPTER

THREE

RESOLVER FOR HOME NETWORKS

To start off, let's ask the all-important question "Why would you want Unbound as a resolver for your home network?"

Firstly, Unbound supports DNSSEC which, through an authentication chain, verifies that the DNS query responses you receive are unaltered, as opposed to query responses which are not DNSSEC-signed and could be changed by anyone who has access to the query. Secondly, by using your own resolver you stop sharing your DNS traffic with third parties and increase your DNS privacy. While you still send out (parts of) your queries unencrypted, you could configure Unbound to take it a step further, which we'll talk about in an upcoming guide. Lastly, when you run your own resolver your DNS cache will be local to your network. Even though the first time you resolve a domain name may be slightly slower than using your ISP's resolver, all subsequent queries for the name will likely be much faster.

In this tutorial we'll look at setting up Unbound as a DNS resolver; First for your own machine, and then for your entire network.

3.1 Setting up Unbound

Unbound is a powerful validating, recursive, caching DNS resolver. It's used by some of the biggest tech companies in the world as well as small-office / home-office users, who use it together with ad blockers and firewalls, or self-hosted resolvers. Setting it up for your home network can be quite simple as we'll showcase below.

Setting up a caching DNS server for your entire home network requires a recursive DNS resolver, and a dedicated machine on which the resolver runs; this (small) system is always on and accessible to the entire network. It can be as small as a Raspberry Pi or any other available Linux/Unix machine that is always online and has Internet connectivity via your router.

Because of the variety of machines that Unbound can run on we cannot create a comprehensive tutorial for all possible options. For this tutorial we will use **Ubuntu 22.04 LTS** as a stepping stone you can adapt and apply to other systems.

While you could download the code from GitHub and build it yourself, getting a copy can be as simple as running:

```
sudo apt update
sudo apt install unbound -y
```

This gives you a complete and running version of Unbound which behaves as a caching recursive DNS resolver out of the box for the system on which you install it. You can check which version of Unbound you have installed with `unbound -V`. The version installed will vary depending on the operating system. If the version is installed is quite old (at the time of writing it isn't) or you'd simply like to run the latest and greatest version you can download the latest release tarball from our website and build it yourself.

Do note that by default Unbound will be queriable only from the local host, i.e. the system on which you installed Unbound. We will change that later.

3.2 Testing the resolver locally

To verify that the server works correctly, it's a good idea to test it before committing the entire network to it. Luckily we can test this on the machine that you installed Unbound on (locally) and from any other machine (remotely) that will be using the resolver after we expose Unbound to the network.

The command for testing locally on the Unbound machine is:

```
dig example.com @127.0.0.1
```

Here we tell the **dig** tool to look up the IP address for example.com, and to ask for this information from the resolver running at the IP address `127.0.0.1`, which is where our Unbound machine is running by default. We can verify that Unbound has indeed answered our query instead of the default resolver that is present on Ubuntu by default. In the output of every **dig** command there is an `ANSWER SECTION` which gives the response to the query. In the footer section of the output, the server which has answered the query under the `SERVER` entry. The entry will look like:

```
;; SERVER: 127.0.0.1#53(127.0.0.1)
```

In the next section we will be disabling the default Ubuntu resolver. To verify that we do it correctly it is useful to know the address of the default resolver as a baseline. For this baseline we also use a **dig** query, but this time without specifying an IP address (which causes dig to use the machine's default DNS resolver).

```
dig example.com
```

While the response should be the same, the `SERVER` entry in the response should look like:

```
;; SERVER: 127.0.0.53#53(127.0.0.53)
```

Note that the final IPv4 digit is 53 and not 1, as with our Unbound instance.

3.3 Setting up for a single machine

Now that we have tested our Unbound resolver, we can tell our machine to use it by default. The resolver your machine uses by default is defined in `/etc/systemd/resolved.conf` in the `DNS` entry (It uses `127.0.0.53`). While just changing this file will work as long as the machine doesn't reboot, we need to make sure that this change is persistent. To do that, we need to change the `DNS` entry to be equal to `127.0.0.1` so the machine uses Unbound as default. To make the change persistent, we also need to set the `DNSStubListener` to `no` so that is not changed by our router (such as with a "recommended resolver" mentioned below). We also want to enable the `DNSSEC` option so that we can verify the integrity the responses we get to our DNS queries. With your favourite text editor (e.g. **nano**) we can modify the file:

```
nano /etc/systemd/resolved.conf
```

Here, under the `[Resolve]` section we add (or rather, enable by removing the "#") the options:

```
[Resolve]
DNS=127.0.0.1
#FallbackDNS=
#Domains=
DNSSEC=yes
#DNSOverTLS=no
#MulticastDNS=no
#LLMNR=no
```

(continues on next page)

(continued from previous page)

```
#Cache=no-negative
DNSStubListener=no
#DNSStubListenerExtra=
```

To actually have the system start using Unbound, we then need to create a symlink to overwrite `/etc/resolv.conf` to the one we modified.

```
ln -fs /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

With this file modified, we can restart using this configuration with:

```
systemctl restart systemd-resolved
```

If successful, the operating system should use our Unbound instance as default. A quick test a **dig** without specifying the address of the Unbound server should give the same result as specifying it did above (with `@127.0.0.1`).

```
dig example.com
```

Note that the "SERVER" section in the output from **dig** should also contain the local IP address of our server.

```
;; SERVER: 127.0.0.1#53(127.0.0.1)
```

3.4 Setting up for the rest of the network

While we currently have a working instance of Unbound, we need it to be reachable from within our entire network. With that comes the headache of dealing with (local) IP addresses. It's likely that your home router distributed local IP addresses to your devices. If this is the case (i.e. you didn't change it by hand), they should be **RFC 1918** ranges:

```
10.0.0.0 - 10.255.255.255 (10/8)
172.16.0.0 - 172.31.255.255 (172.16/12)
192.168.0.0 - 192.168.255.255 (192.168/16)
```

To find the IP address of the machine that is running Unbound, we use:

```
hostname --all-ip-addresses
```

If you just have one IP address as output from the **hostname** command that will be the correct one. If you have multiple IP addresses, the easiest way to determine which IP address to use, is to find out which connection goes to your home router. Keep in mind that using the wrong IP address here can be a source of connectivity errors further on. For the purpose of this tutorial we assume that our home router has the IP address `192.168.0.1`, as this is typical for home routers, and our resolver machine (the machine that is running our Unbound instance) has IP address `192.168.0.2`, which we will get into in the next section.

As a prerequisite for the next step, we need to configure our Unbound instance to be reachable from devices other than only the machine on which the Unbound is running. Unbound is a highly capable resolver, and as such has many options which can be set; the full example configuration file is almost 1200 lines long, but we'll need but a fraction of these settings. (If you are interested, all configuration options are documented in the extensive manual page of *unbound.conf(5)*).

The default configuration file is found at:

```
/etc/unbound/unbound.conf
```

If you open this for the first time it looks very empty. It is still usable as a resolver for one machine, as this is how the Unbound defaults are configured. It's not, however, enough for our purposes, so we will add the minimal configuration options needed.

The options that we add to the current configuration file to make it a "minimal usable configuration" are as follows. Note that the IPv6 options are commented out, but we recommend to uncomment them if your router and network supports it.

```
server:
    # location of the trust anchor file that enables DNSSEC
    auto-trust-anchor-file: "/var/lib/unbound/root.key"
    # send minimal amount of information to upstream servers to enhance privacy
    qname-minimisation: yes
    # the interface that is used to connect to the network (this will listen to all␣
↪interfaces)
    interface: 0.0.0.0
    # interface: ::0
    # addresses from the IP range that are allowed to connect to the resolver
    access-control: 192.168.0.0/16 allow
    # access-control: 2001:DB8/64 allow

remote-control:
    # allows controling unbound using "unbound-control"
    control-enable: yes
```

The interface is currently configured to listen to any address on the machine, and the access-control only allows queries from the `192.168.0.0/16` IP subnet range. Note that the IP address we chose above (`192.168.0.1` and `192.168.0.2`) fall within the `192.168.0.0/16` range.

To prepare our configuration we are going to modify the existing configuration in `/etc/unbound/unbound.conf`. If you open the file for the first time, you see that there is already an "include" in there. The "include" enables us to do DNSSEC, which allows Unbound to verify the source of the answers that it receives, as well as QNAME minimisation. For convenience these configuration options have already been added in the minimal configuration. The configuration also includes the *remote-control:* section in the configuration to enable controlling Unbound using the *unbound-control(8)* command, which is useful if you want to modify the configuration on the fly later on.

Using the text editor again, we can then add the minimal configuration shown above, making any changes to the access control where needed. When we've modified the configuration we check it for mistakes with the *unbound-checkconf(8)* command:

```
unbound-checkconf unbound.conf
```

If this command reports no errors, we need to stop the currently running Unbound instance and restart it with our new configuration. You can stop Unbound with:

```
sudo pkill -f unbound
```

And you can restart Unbound with:

```
unbound-control start
```

From this point on, we can *stop*, *start*, and *reload* Unbound with **unbound-control** if you want to make changes to the configuration.

3.5 Testing the resolver from a remote machine

So now we have a DNS resolver which should be reachable from within the network. To be able to verify that our resolver is working correctly, we want to test it from another machine in the network. As mentioned above, this tutorial uses the address `192.168.0.2` (not `127.0.0.1` as we saw earlier) as an example for the machine running Unbound. Armed with the IP address we can send a query to our DNS resolver from another machine which is within our home network. To do this we use the same dig command, only we change the IP address where the query is asked.

```
dig example.com @192.168.0.2
```

This should give the same result as above. The `SERVER` entry in the footer reflects from which server the response was received.

3.6 Where it all comes together

We should now have a functioning DNS resolver that is accessible to all machines in our network (**make sure you do before you continue**).

The next step then is a little tricky as there are many options and variations possible. We have a choice of which machines in our network will be using our configured DNS resolver. This can range from a single machine to all the machines that are connected. Since this tutorial cannot (and does not try to) be comprehensive for the range of choices, we will look at some of the basic examples which you can implement and expand on.

Most machines when they first connect to a network get a "recommended resolver" from your router using DHCP (Dynamic Host Configuration Protocol). To change this, we need to log into the router. Earlier in this tutorial we assume the home router was using `192.168.0.1`, though in reality this can differ. If this does differ, the unbound configuration needs to be changed as well.

To find the IP address of our home router, which is likely be under the `default gateway` entry from:

```
ip route
```

When you've found the IP address of your home router, you can copy the address to a web browser, which should give you access to the router configuration portal. If you can't find the portal using this method, consult the manual or the manufacturer's website. When you have access, you should change the DHCP configuration to advertise the IP address of the machine running Unbound as the default gateway. In the case of our example, that would be `192.168.0.2`.

Another possibility is a machine that does not use a resolver that is "recommended" by your router. This machine can be running its own resolver or be connected to a different one altogether. If you want these machines to use the Unbound resolver you set up, you need to change the configuration of the machine.

CHAPTER

FOUR

LOCAL DNS (STUB) RESOLVER FOR A SINGLE MACHINE

Unbound is a powerful validating, recursive, caching DNS resolver. It's used by some of the biggest tech companies in the world as well as home users, who use it together with ad blockers and firewalls, or self-run resolvers. Setting it up as a caching resolver for your own machine can be quite simple as we'll showcase below.

We strongly recommend setting up DNSSEC during the Unbound configuration step, as it allows the verification of the integrity of the responses to the queries you send.

If you need to install Unbound first visit the *Installation* page.

4.1 Configuring the Local Stub resolver

For configuring Unbound we need to make sure we have Unbound installed. An easy test is by asking the version number.

```
unbound -V
```

Once we have a working version of Unbound installed we need to configure it to be a recursive caching resolver (information about recursive resolvers can be found here, but is not necessary for our purposes here). Luckily for us Unbound already behaves as such by default, so for basic purposes we can use the configuration from the *Configuration* page. We always recommend enabling DNSSEC.

Once we have a configuration we are happy with, we need to tell our machine to use Unbound by default instead of what it is currently using. This works differently on different operating systems. Below we will go through this for a selection of OSes.

Note: Make sure your Unbound can run with the configuration we create. Steps for this can be found *on the configuration page*.

4.1.1 Ubuntu 22.04 LTS

The resolver your machine uses by default is defined in `/etc/systemd/resolved.conf` in the `DNS` entry and uses the IP address `127.0.0.53`.

We can test this by using **`dig`** to "example.com" and looking at the output.

```
dig example.com
```

Near the bottom of the output we can see `127.0.0.53` IP address.

```
;; SERVER: 127.0.0.53#53(127.0.0.53)
```

To change this, we are going to change the `resolved.conf`. While just changing this file will work as long as the machine doesn't reboot, we need to make sure that this change is *persistent*. To do that, we need to change the `DNS` entry to be equal to `127.0.0.1` (or whatever IP address Unbound is bound to in your configuration) so the machine uses Unbound as default. So the interface would look like this in the Unbound config:

```
server:
    # specify the interface to answer queries from by ip-address.
    interface: 127.0.0.1
```

To test that Unbound is running, we can tell **`dig`** to use a specific server with the `@`.

```
dig example.com @127.0.0.1
```

If Unbound is running, the output should contain the address that we specified in the config:

```
;; SERVER: 127.0.0.1#53(127.0.0.1)
```

If we changed `resolved.conf` now, the default resolver would be persistent until the router wants to update it. To make sure it doesn't do that we also need to set the `DNSStubListener` to `no` so that is not changed by our router (such as with a "recommended resolver" mentioned below). We also want to enable the `DNSSEC` option so that we can verify the integrity the responses we get to our DNS queries. With your favourite text editor (e.g. **`nano`**) we can modify the file:

```
nano /etc/systemd/resolved.conf
```

Here, under there `[Resolve]` header we add/substitute our changes to the options:

```
[Resolve]
DNS=127.0.0.1
#FallbackDNS=
#Domains=
DNSSEC=yes
#DNSOverTLS=no
#MulticastDNS=no
#LLMNR=no
#Cache=no-negative
DNSStubListener=no
#DNSStubListenerExtra=
```

To actually have the system start using our changed config, we then need to create a symlink to overwrite `/etc/resolv.conf` to the one we modified.

```
ln -fs /run/systemd/resolve/resolv.conf /etc/resolv.conf
```

Note: Make sure your Unbound is running at at the IP address from the modified resolv.conf before the next step, otherwise you might break your internet connection.

With the resolv.conf file modified, we can restart systemd using the new resolver configuration with:

```
systemctl restart systemd-resolved
```

If successful, the operating system should use our Unbound instance as default. A quick test a **dig** without specifying the address of the Unbound server should give the same result as specifying it did above (with `@127.0.0.1`).

```
dig example.com
```

Here we tell the **dig** tool to look up the IP address for `example.com`. We did not specify where **dig** should ask this, so it goes to the default resolver of the machine.

```
dig example.com
```

It should look the same as with the `127.0.0.1` IP specified as we did earlier.

```
;; SERVER: 127.0.0.1#53(127.0.0.1)
```

Note: Unbound is not persistent at this point, and will not start up when your system does (and possibly "breaking" your internet). This is fixed by restarting your Unbound upon reboot.

Package manager

To make Unbound persistent between restarts, we need to add it to the systemd service manager, for which we'll need a service file. If you installed Unbound via the package manager, this service file is already created for you and the only thing that is missing, is it executing our own configuration file.

To make sure we execute Unbound with our own configuration, we copy our config file to the default location of the config file: `/etc/unbound/unbound.conf`. Make sure Unbound starts using the copied configuration (this can be done with the `-c` flag to specify the config location).

Before you proceed to the next step, make sure to stop the Unbound that may still be running. Now we can start our Unbound with systemd, which will restart automatically when the system is rebooted.

```
systemctl start unbound
```

To check that everything is correct, you can see the status (which should be "active"):

```
systemctl status unbound
```

We can now **dig** a final time, to verify that this works.

Compilation

The steps for making Unbound persistent are almost exactly the same as if you installed it via the package manager, except that the service file that is needed by systemd does not exist yet. So instead of changing it, we create it and call it `unbound.service`, and copy the minimally modified service file supplied by the package manager. It should be located at: `/lib/systemd/system/unbound.service`.

So using your favorite text editor open the file:

```
nano /lib/systemd/system/unbound.service
```

and copy the file contents below:

```
[Unit]
Description=Unbound DNS server
Documentation=man:unbound(8)
After=network.target
Before=nss-lookup.target
Wants=nss-lookup.target

[Service]
Type=simple
Restart=on-failure
EnvironmentFile=-/usr/local/etc/unbound
ExecStart=/usr/local/sbin/unbound -d -p $DAEMON_OPTS
ExecReload=+/bin/kill -HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

Note that in this file `systemctl` uses the default config location. This location is different depending on the installation method used. In this case the default config file is located at `/usr/local/etc/unbound`. We need to copy the config that we are going to use here.

Once you have your config copied in the right location, we need to make sure the system can find it.

Because we change the service file on disk (we created it), systemctl needs to be reloaded:

```
systemctl daemon-reload
```

We then need to enable Unbound as a systemctl service:

```
systemctl enable unbound
```

If all steps went correctly, we can start Unbound now using systemctl. Note that any previous Unbound instances with the same config (specifically the same ip-address) needs to be stopped.

```
systemctl start unbound
```

We can then look at the status, which should be "active".

```
systemctl status unbound
```

If you succeeded Unbound should now be the default resolver on your machine and it will start when your machine boots.

4.1.2 macOS Big Sur

To find out which resolver your machine uses, we have two options: Look at the DNS tab under the Network tab in the System Preferences app, or we can use the **`scutil`** command in the terminal. The **`scutil`** command can be used to manage and give information about the system configuration parameters. When used for DNS, it will show you all the configured resolvers though we are only interested in the first.

```
scutil --dns
```

The output will show all the resolvers configured, but we are interested in the first entry. Before configuring Unbound to be our resolver, the first entry is (likely) the resolver recommended by your router.

The simplest method of changing the resolver of your Mac is by using the System Preferences Window (the option of doing this step via the command line terminal also exists if you want to script this step). The steps go as follows:

1. Open the Network tab in System Preferences.
2. Click on the Advanced button.
3. Go to the DNS Tab.
4. Click "+" icon
5. Add IP address of Unbound instance (here we use `127.0.0.1`)

Once the IP address is added we can test our Unbound instance (assuming it's running) with **`dig`**. Note that the Unbound instance cannot be reached before it has been added in the DNS tab in System Preferences.

```
dig example.com @127.0.0.1
```

> **Attention:** If you restart your Mac at this stage in the process, you will not have access to the internet anymore. This is because Unbound does not automatically restart if your machine restarts. To make remedy this, we need to add Unbound to the startup routine on your Mac.

Depending on your installation method, either via `Homebrew` or compiling Unbound yourself, the method of making Unbound persistent differs slightly. For both methods we use **`launchctl`** to start Unbound on the startup of your machine.

Homebrew

If you installed Unbound using Homebrew, the XML file required by **`launchctl`** is already supplied during installation. The file can be found at `/Library/LaunchDaemons/homebrew.mxcl.unbound.plist`. To load this file we invoke the following command.

```
sudo launchctl load /Library/LaunchDaemons/homebrew.mxcl.unbound.plist
```

Now every time you restart your machine, Unbound should restart too.

Compilation

If you installed Unbound by compiling it yourself, we need to create an XML file for **`launchctl`**. Conveniently we've created one for you:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
    <key>Label</key>
    <string>nl.nlnetlabs.unbound</string>
    <key>KeepAlive</key>
    <true/>
    <key>RunAtLoad</key>
    <true/>
    <key>ProgramArguments</key>
```

(continues on next page)

(continued from previous page)

```
    <array>
        <string>/usr/local/sbin/unbound</string>
        <string>-c</string>
        <string>/usr/local/etc/unbound/unbound.conf</string>
    </array>
    <key>UserName</key>
    <string>root</string>
    <key>StandardErrorPath</key>
    <string>/dev/null</string>
    <key>StandardOutPath</key>
    <string>/dev/null</string>
    </dict>
</plist>
```

The main components that interest us, are the items in the <array> which execute the command. Firstly, we invoke Unbound from the location that it has been installed (for example using `make install`). Secondly, we add the `-c` option to supply a configuration file. Lastly, we add the location of the default configuration file. The location in the XML can be changed to another location if this is convenient.

Using the text editor of choice, we then create the file `/Library/LaunchDaemons/nl.nlnetlabs.unbound.plist` and insert the above supplied XML code. To be able to use the file, we need to change the permissions of the file using **chmod**

```
sudo chmod 644 /Library/LaunchDaemons/nl.nlnetlabs.unbound.plist
```

We can then load the file with the following command.

```
sudo launchctl load /Library/LaunchDaemons/nl.nlnetlabs.unbound.plist
```

Now every time you restart your machine, Unbound should restart too.

New in version 1.17.0.

CHAPTER

FIVE

DOWNSTREAM PROXY SUPPORT

Since version 1.17.0, Unbound can play nicely in environments where supported DNS reverse-proxying is in place. It is able to use the proxied client information as the "real" client address for all functions, except in the actual network communication, where a client address is used, such as access control, logging, DNSTAP, RPZ and IP rate limiting.

The currently supported environment is PROXY protocol version 2 (PROXYv2).

5.1 PROXYv2

New in version 1.17.0.

Unbound supports PROXYv2 for downstream connections; that is clients (read proxies) talking to Unbound.

The PROXY protocol is protocol agnostic and can work with any layer 7 protocol even when encrypted. It works on both UDP and TCP based transports and in a nutshell it prepends the client information in the application's payload. This is done once at the start of a TCP stream, or in every UDP packet. The caveat is that both the proxy and the upstream server (i.e., Unbound) need to understand the PROXY protocol.

5.1.1 Configuration

Configuring Unbound for PROXYv2 is pretty straight forward. The following minimal configuration allows Unbound to listen for incoming queries on port 53 (the default) and marks the same port as a PROXYv2 port:

```
server:
        interface: eth0
        proxy-protocol-port: 53
        interface-action: eth0 allow
```

This means that Unbound **expects** PROXYv2 information on that port.

> **Warning:** In absence of a valid PROXYv2 header Unbound will terminate/drop the connection/packet.

The port configuration can be used alongside plain UDP and plain TCP ports (as in the example above), but also together with DNS over TLS ports.

Note: The coexistence of PROXYv2 together with either DNSCrypt or DNS over HTTP is not supported.

See also:

proxy-protocol-port in the *unbound.conf(5)* manpage.

New in version 1.6.0.

Changed in version 1.11.0: **RFC 8767** behavior is introduced

CHAPTER

SIX

SERVING STALE DATA

Unbound supports serving stale data from its cache, as described in **RFC 8767**. Serving stale data would normally break the contract between an authoritative name server and a caching resolver on the amount of time a record is permitted to be cached. However, the TTL definition of **RFC 8767#section-4** states that:

> "If the data is unable to be authoritatively refreshed when the TTL expires, the record MAY be used as though it is unexpired."

Serving expired records is not a novel idea and it was already present in various forms (e.g., increased cache-hit ratio, fallback when upstream is not reachable) in various resolvers. Unbound's own form is called *serve-expired:* and its main purpose was to increase the cache-hit ratio.

As the RFC landed in the standards track, Unbound gained support for it but still kept the original serve-expired logic. Certain aspects of the RFC, such as timers, were already present in Unbound and their functionality is shared by both modes of operation.

The following sections try to clarify the differences between serve-expired and **RFC 8767** (serve-stale) and give some insight into when one may be preferable over the other. I will refrain from using the RFC term serve-stale in order to avoid any confusion between the terms and the configuration options later on.

6.1 serve-expired

Since version 1.6.0, Unbound has the ability to answer with expired records. Before trying to resolve, Unbound will also consider expired cached records as possible answers. If such a record is found it is immediately returned to the client (cache response speed!). But contrary to normal cache replies, Unbound continues resolving and hopefully updating the cached record.

The immediate downside is obvious: the expired answers rely heavily on the cache state. Unbound already has the tools to try and tip the scales in its favor with the *prefetch:* and *serve-expired-ttl:* options.

With prefetch, Unbound tries to update a cached record (after first replying to the client) when the current TTL is within 10% of the original TTL value. The logic is similar to serve-expired: if a cached record is found and the record is within 10% of the TTL, it is returned to the client but Unbound continues resolving in order to update the record. Although prefetching comes with a small penalty of ~10% in traffic and load from the extra upstream queries, the cache is kept up-to-date, at least for popular queries.

Rare queries have the inescapable fate of having their records expired past any meaningful time. The option *serve-expired-ttl:* limits the amount of time an expired record is supposed to be served. **RFC 8767#section-5-11** suggests a value between one and three days.

Note: A note on the expired reply's TTL value: prior to the RFC, Unbound was using TTL 0 in order to signal that the expired record is only meant to be used for this DNS transaction and not to be cached by the client. The RFC now

RECOMMENDS a value of 30 to be returned to the client.

A simple configuration for the primal serve-expired behavior could then be:

```
server:
    prefetch: yes
    serve-expired: yes
    serve-expired-ttl: 86400  # one day, in seconds
```

This will allow Unbound to:

- prioritize (expired) cached replies,
- keep the cache fairly up-to-date, and
- in the likelihood that an expired record needs to be served (e.g., rare query, issue with upstream resolving), make sure that the record is not older than the specified limit.

6.2 RFC 8767

Starting with version 1.11.0, Unbound supports serving expired records following the RFC guidelines. The RFC behavior is mainly focused on returning expired answers as fallback for normal resolution. The option to control that is *serve-expired-client-timeout:* and setting it to a value greater than 0 enables the RFC behavior.

With the value set, Unbound has a limit on how much time it can spend resolving a client query. When that limit is passed, Unbound pauses resolution and checks if there are any expired records in the cache that can answer the initial query. If that is the case, Unbound answers with the expired record before resuming resolution. The result of the resolution will be used to update the cache if possible.

Similar to the client timeout, Unbound will also try and use expired answers instead of returning SERVFAIL to the client where possible.

A simple configuration for the RFC behavior could then be:

```
server:
    serve-expired: yes
    serve-expired-ttl: 86400            # one day, in seconds
    serve-expired-client-timeout: 1800  # 1.8 seconds, in milliseconds
```

This will allow Unbound to use expired answers only as fallback from normal resolving:

- when 1.8 seconds have passed since the client made the query,
- instead of returning SERVFAIL, or
- in the likelihood that an expired record needs to be served (e.g., issue with upstream resolving), make sure that the record is not older than the specified limit.

6.3 Conclusion

Unbound offers two distinct modes for serving expired records. The safest approach is to use the RFC behavior where expired records are used as a fallback to availability, network or configuration errors. This will serve expired records as a last resort instead of returning SERVFAIL or the client giving up.

If more client-side performance is required, the default original serve-expired behavior can keep the cache-hit ratio higher. Using it together with the prefetch option is highly recommended in order to try and keep an updated cache.

In all cases make sure to consult the *unbound.conf(5)* manpage of your installed Unbound for defaults and suggested values. And always remember that serving expired records should be approached with caution; you may be directing your clients to places long gone.

See also:

serve-expired, *serve-expired-ttl*, *serve-expired-ttl-reset*, *serve-expired-reply-ttl* and *serve-expired-client-timeout* in the *unbound.conf(5)* manpage.

CHAPTER

SEVEN

PERFORMANCE TUNING

Most users will probably not have a need to tune and optimise their Unbound installation, but it could be useful for large resolver installations. This page contains recommendations based on user feedback. If you have different experiences or have recommendations, please share them on the Unbound users mailing list.

7.1 Configuration

Set *num-threads:* equal to the number of CPU cores on the system. For example, for 4 CPUs with 2 cores each, use 8.

On Linux, set *so-reuseport: yes*, that will significantly improve UDP performance (on kernels that support it, otherwise it is inactive, the *unbound-control(8)* status command shows if it is active).

Set `*-slabs` to a power of 2 close to the `num-threads` value. Do this for *msg-cache-slabs:*, *rrset-cache-slabs:*, *infra-cache-slabs:* and *key-cache-slabs:*. This reduces lock contention.

Increase the memory size of the cache. Use roughly twice as much rrset cache memory as you use msg cache memory. For example, *rrset-cache-size: 100m* and *msg-cache-size: 50m*. Due to malloc overhead, the total memory usage is likely to rise to double (or 2.5x) the total cache memory that is entered into the configuration.

Set the *outgoing-range:* to as large a value as possible, see the sections below on how to overcome the limit of 1024 in total. This services more clients at a time. With 1 core, try 950. With 2 cores, try 450. With 4 cores try 200. The *num-queries-per-thread:* is best set at half the number of the `outgoing-range`, but you would like a whole lot to be able to soak up a spike in queries. Because of the limit on `outgoing-range` thus also limits `num-queries-per-thread`, it is better to compile with `libevent` (see the section below), so that there is no more 1024 limit on `outgoing-range`.

Set *so-rcvbuf:* to a larger value (4m or 8m) for a busy server. This sets the kernel buffer larger so that no messages are lost in spikes in the traffic. Adds extra 9s to the reply-reliability percentage. The OS caps it at a maximum, on Linux, Unbound needs root permission to bypass the limit, or the admin can use `sysctl net.core.rmem_max`. On BSD change `kern.ipc.maxsockbuf in /etc/sysctl.conf`.

On OpenBSD change header and recompile kernel. On Solaris `ndd -set /dev/udp udp_max_buf 8388608`.

Also set *so-sndbuf:* to a larger value (4m or 8m) for a busy server. Same as `so-rcvbuf`, but now for spikes in replies, and it is `net.core.wmem_max`. Might need a smaller value, as spikes are less common in replies, you can see rcv and snd buffer overruns with `netstat -su`, `RcvbufErrors` and `SndbufErrors`, and similar reports on BSD.

For the TCP listen backlog on Linux, it is possible to tweak the kernel parameters to allow larger values. Unbound attempts to increase this to enable it to handle spikes in incoming TCP or TLS connections. The number that unbound attempts is defined in `TCP_BACKLOG` in `services/listen_dnsport.c`, it does not need to be changed if the current value, about 256, is sufficient for you. However, the Linux kernel limits this value silently to a maximum configured into the kernel settings. The kernel can be tweaked to enable a higher number with `net.core.somaxconn = 256` and `net.ipv4.tcp_max_syn_backlog = 256`.

Here is a short summary of optimisation config:

```
# some optimisation options.
server:
    # use all CPUs
    num-threads: <number of cores>

    # power of 2 close to num-threads
    msg-cache-slabs: <same>
    rrset-cache-slabs: <same>
    infra-cache-slabs: <same>
    key-cache-slabs: <same>

    # more cache memory, rrset=msg*2
    rrset-cache-size: 100m
    msg-cache-size: 50m

    # more outgoing connections
    # depends on number of cores: 1024/cores - 50
    outgoing-range: 950

    # Larger socket buffer.  OS may need config.
    so-rcvbuf: 4m
    so-sndbuf: 4m

    # Faster UDP with multithreading (only on Linux).
    so-reuseport: yes
```

The default setup works fine, but when a large number of users have to be served, the limits of the system are reached. Most pressing is the number of file descriptors, the default has a limit of 1024. To use more than 1024 file descriptors, use libevent or the forked operation method. These are described in sections below.

7.2 Using Libevent

Libevent is a BSD licensed cross platform wrapper around platform specific event notification system calls. Unbound can use it to efficiently use more than 1024 file descriptors. Install `libevent` (and `libevent-devel`, if it exists) with your favorite package manager. Before compiling unbound run:

```
./configure --with-libevent
```

Now you can give any number you like for *outgoing-range:*. Also increase the *num-queries-per-thread:* value.

```
# with libevent
outgoing-range: 8192
num-queries-per-thread: 4096
```

Users report that libevent-1.4.8-stable works well. Users have confirmed it works well on Linux and FreeBSD with 4096 or 8192 as values. Double the *num-queries-per-thread:* and use that as *outgoing-range:*.

Stable(old) distributions can package older versions (such as libevent-1.1), for which there are crash reports, thus you may need to upgrade your libevent. In unbound 1.2.0 a race condition in the libevent calls was fixed.

Unbound can compile from the libevent or libev build directory to make this easy; e.g.,

```
configure --with-libevent=/home/user/libevent-1.4.8-stable
```

or

```
configure --with-libevent=/home/user/libev-3.52
```

Note: If you experience crashes anyway, then you can try the following. Update libevent. If the problem persists, libevent can be made to use different system-call back-ends by setting environment variables. Unbound reports the back-end in use when verbosity is at level 4. By setting `EVENT_NOKQUEUE`, `EVENT_NODEVPOLL`, `EVENT_NOPOLL`, `EVENT_NOSELECT`, `EVENT_NOEPOLL` or `EVENT_NOEVPORT` to yes in the shell before you start unbound, some back-ends can be excluded from use. The *poll(2)* backend is reliable, but slow.

7.3 Forked Operation

Unbound has a unique mode where it can operate without threading. This can be useful if libevent fails on the platform, for extra performance, or for creating walls between the cores so that one cannot poison another.

To compile for forked operation, before compilation use:

```
./configure --without-pthreads --without-solaris-threads
```

This disables threads and enable forked operation. Because no locking has to be done, the code speeds up (about 10 to 20%).

In the configuration file, *num-threads:* still specifies the number of cores you want to use (even though it uses processes and not threads). And note that the *outgoing-range:* and cache memory values are all per thread. This means that much more memory is used, as every core uses its own cache. Because every core has its own cache, if one gets cache poisoned, the others are not affected.

```
# with forked operation
server:
    # use all CPUs
    num-threads: <number of cores>

    msg-cache-slabs: 1
    rrset-cache-slabs: 1
    infra-cache-slabs: 1
    key-cache-slabs: 1

    # more cache memory, rrset=msg*2
    # total usage is 150m*cores
    rrset-cache-size: 100m
    msg-cache-size: 50m

    # does not depend on number of cores
    outgoing-range: 950
    num-queries-per-thread: 512

    # Larger socket buffer.  OS may need config.
    so-rcvbuf: 4m
```

Because every process is using at most 1024 file descriptors now, the effective maximum is the number of cores * 1024. The configuration above uses 950 per process, for 4 processes gives a respectable 3800 sockets. The number of queries per thread is half the number of sockets, to guarantee that every query can get a socket, and some to spare for queries-for-nameservers.

Using forked operation together with libevent is also possible. It may be useful to force the OS to service the file descriptors for different processes, instead of threads. This may have (radically) different performance if the underlying network stack uses (slow) lookup structures per-process.

CHAPTER

EIGHT

MONITORING AND REPORTING

There are several ways to configure statistics in Unbound. In this section we'll cover Munin and Cacti, but there are many other third-party options available as well, using for example Zabbix or Prometheus.

8.1 Configuration

Unbound has an option to enable extended statistics collection. If enabled, more statistics are collected, for example what types of queries are sent to the resolver. Otherwise, only the total number of queries is collected.

Statistics can be printed to the log file using *statistics-interval:*, but here we'll focus on using *unbound-control(8)* to obtain statistics on demand. If you set a `statistics-interval`, every interval it is printed to the logfile.

To use `unbound-control`, first set it up using the *unbound-control-setup* script. See howto setup unbound. Then you can use the *unbound-control stats* command to print out the statistics on demand.

Various graphing tools expect the counters to go up over time. Some may expect counters to be reset to 0 since the previous statistics printout. The *statistics-cumulative:* option controls the behaviour of Unbound. By default it is set to `no`, which resets values to zero after stat printout.

```
# enable extended statistics.
server:
   statistics-interval: 0
   extended-statistics: yes
   # set to yes if graphing tool needs it
   statistics-cumulative: no
```

8.2 Statistics with Munin

In the `contrib` directory in the source of Unbound is the `unbound_munin_` plugin script. It can be used with Munin to monitor the health of an Unbound server.

Install `munin` and `munin-node` with the appropriate package install tool. The plugin script for Unbound can be copied somewhere on the system (such as in the unbound directory). Then create symbolic links from `/etc/munin/plugins` to that file.

```
$ ln -s /etc/unbound/unbound_munin_ /etc/munin/plugins/unbound_munin_hits
```

In the `/etc/munin/plugin-conf.d/plugins.conf` file you can setup the Munin plugin for Unbound. Below are the default values. Set the correct values for your system. The `statefile` is a temporary file.

```
[unbound*]
user root
env.statefile /usr/local/var/munin/plugin-state/unbound-state
env.unbound_conf /usr/local/etc/unbound/unbound.conf
env.unbound_control /usr/local/sbin/unbound-control
env.spoof_warn 1000
env.spoof_crit 100000
```

Restart the munin-node daemon. Munin will automatically pick up the new graph and plot it with `rrdtool`.

Additional graphs are possible, below is a list of them, and examples. Create (additional) symbolic links to `unbound_munin_` with the names (in bold) of those graphs to enable their display. Several require that *extended-statistics:* is enabled in the configuration. Pictures included are samples, your statistics may look different :-) .

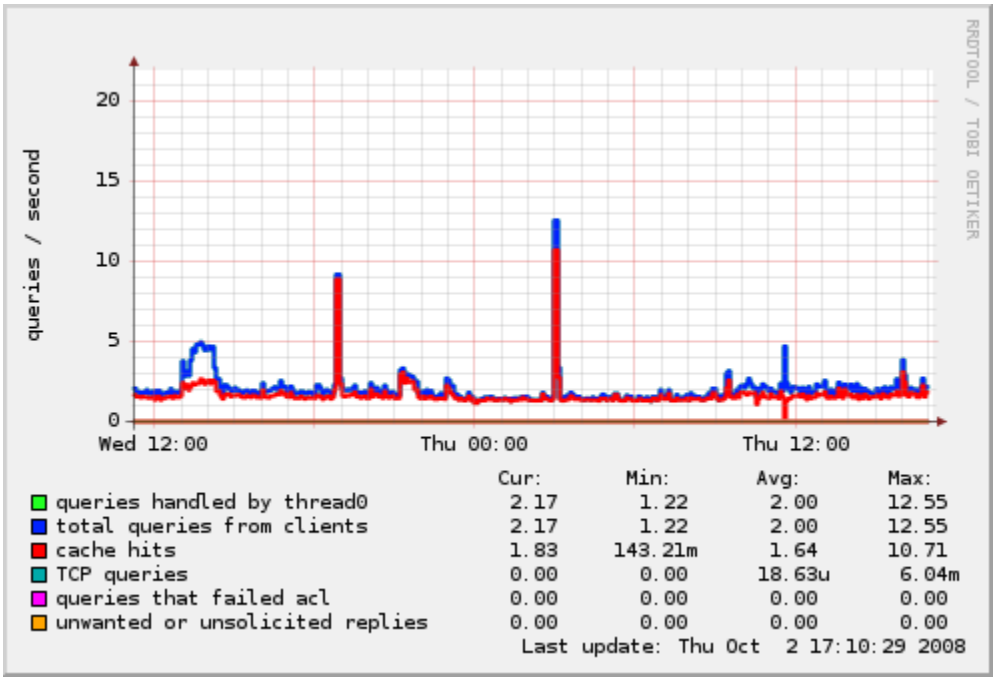


Fig. 1: unbound_munin_hits - base volume, cache hits, unwanted traffic

A sharp increase in unwanted traffic indicates a possible spoof run in progress.

You can see that the server was restarted during the day.

The types received are shown.

Usually only IN (internet) class.

Usually only QUERY (normal query).

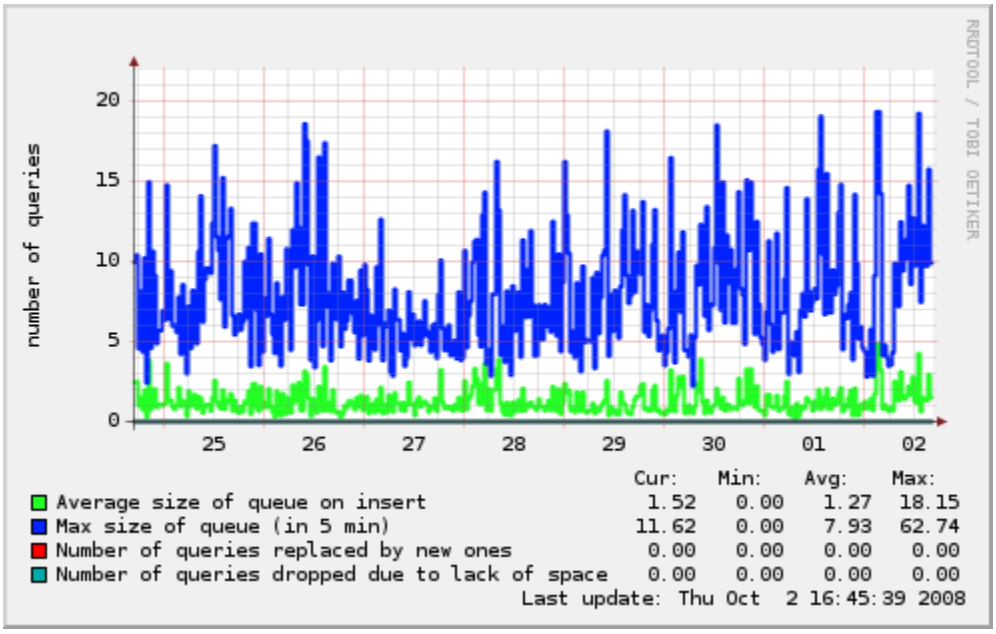


Fig. 2: unbound_munin_queue - to monitor the internal requestlist

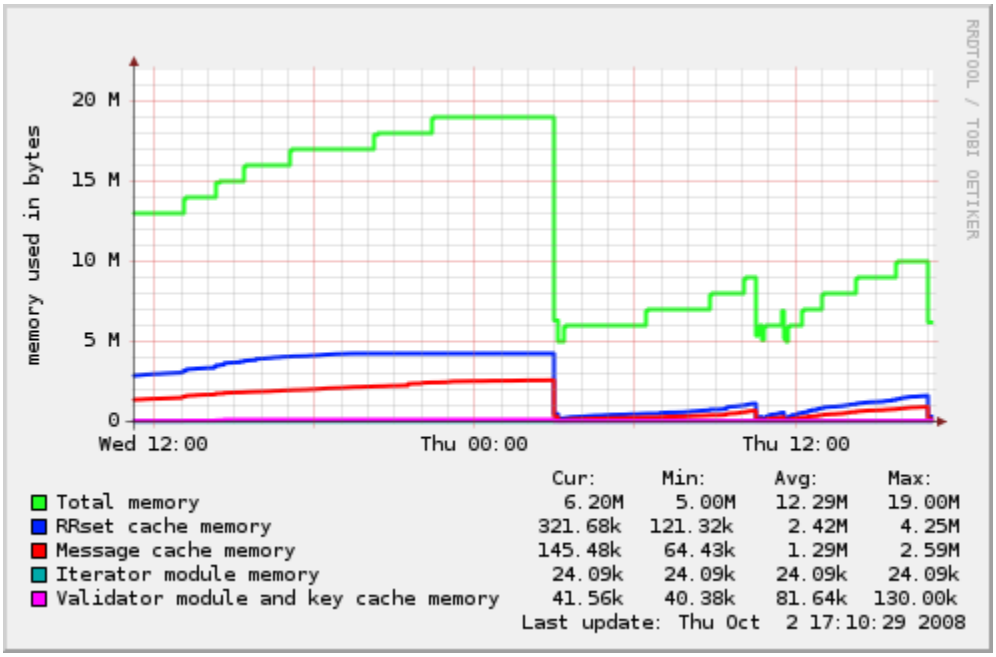


Fig. 3: unbound_munin_memory - memory usage

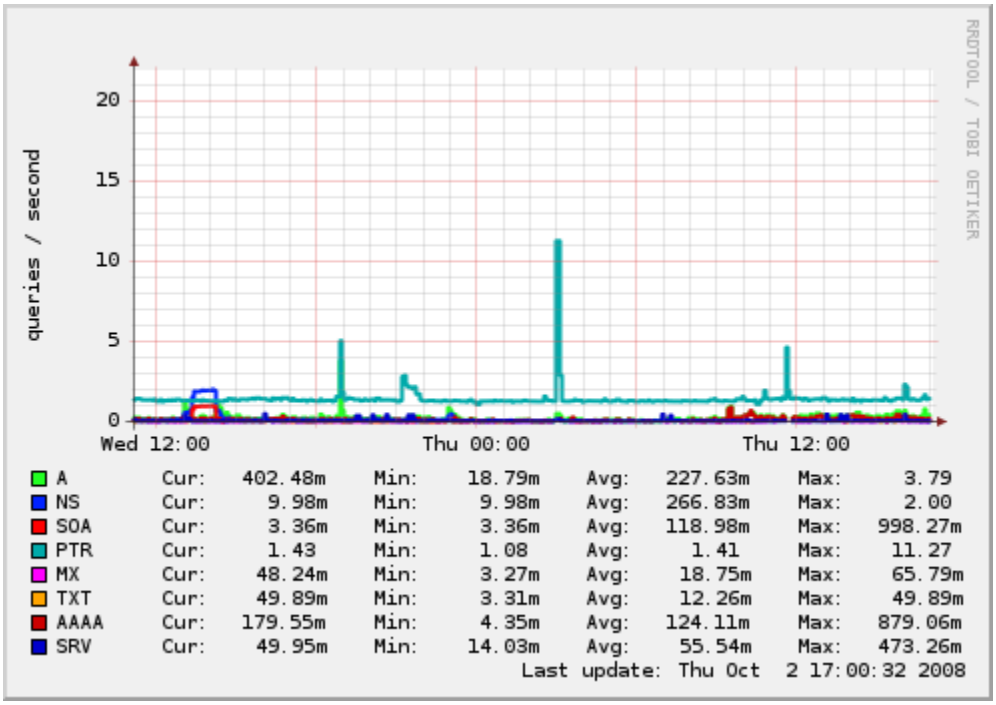


Fig. 4: unbound_munin_by_type - incoming queries by type

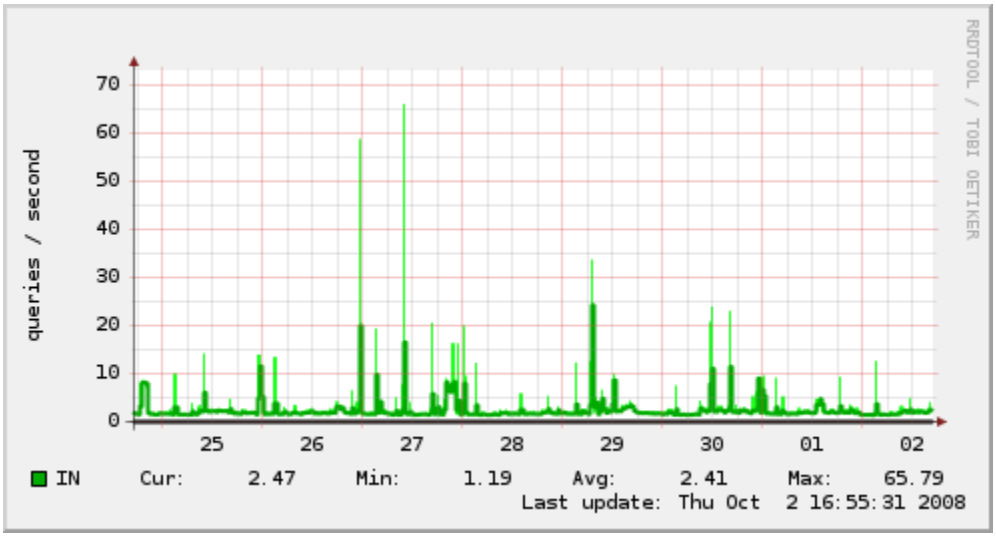


Fig. 5: unbound_munin_by_class - incoming queries by class

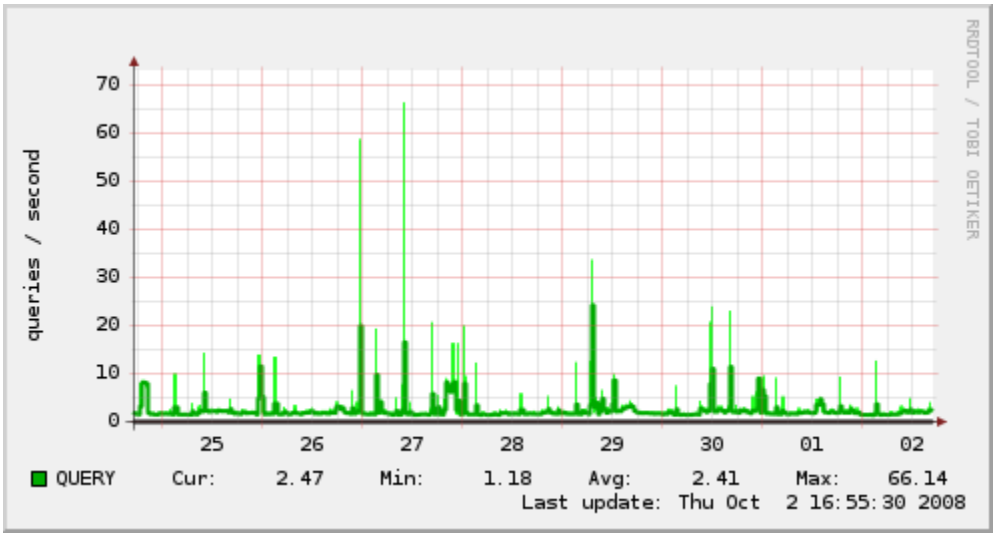


Fig. 6: unbound_munin_by_opcode - incoming queries by opcode

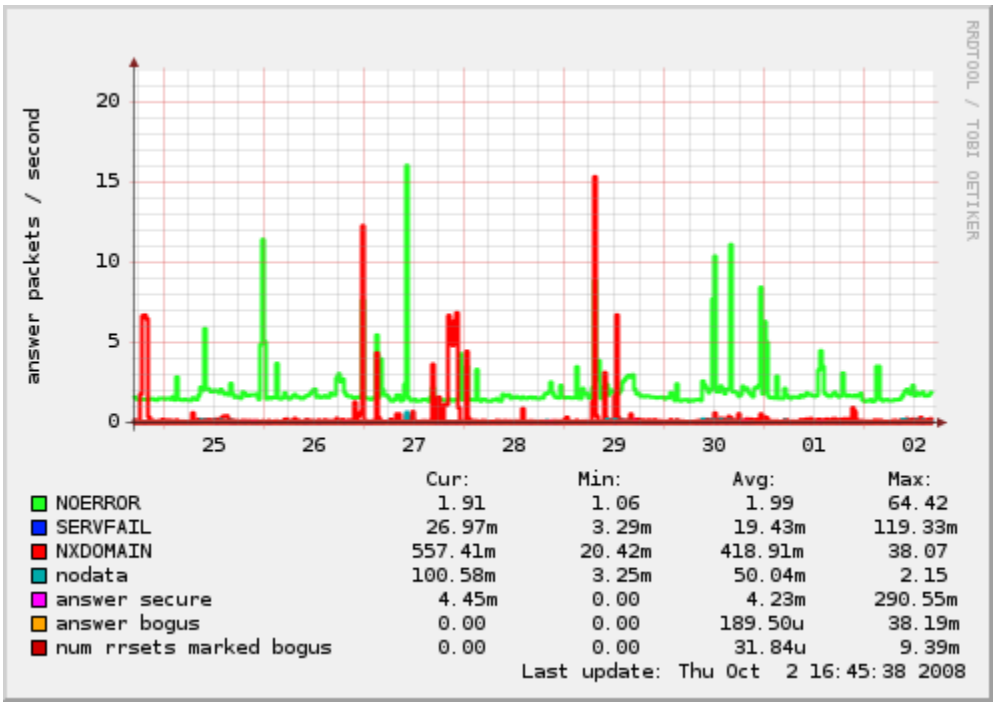


Fig. 7: unbound_munin_by_rcode - answers by rcode, validation status

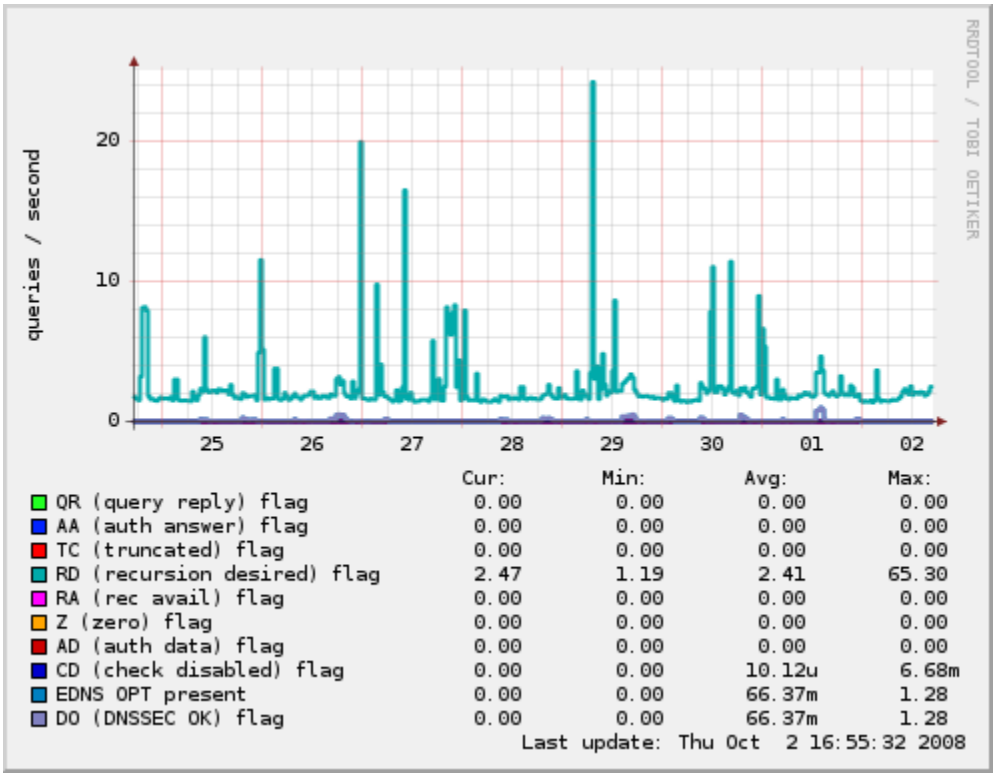


Fig. 8: unbound_munin_by_flags - incoming queries by flags

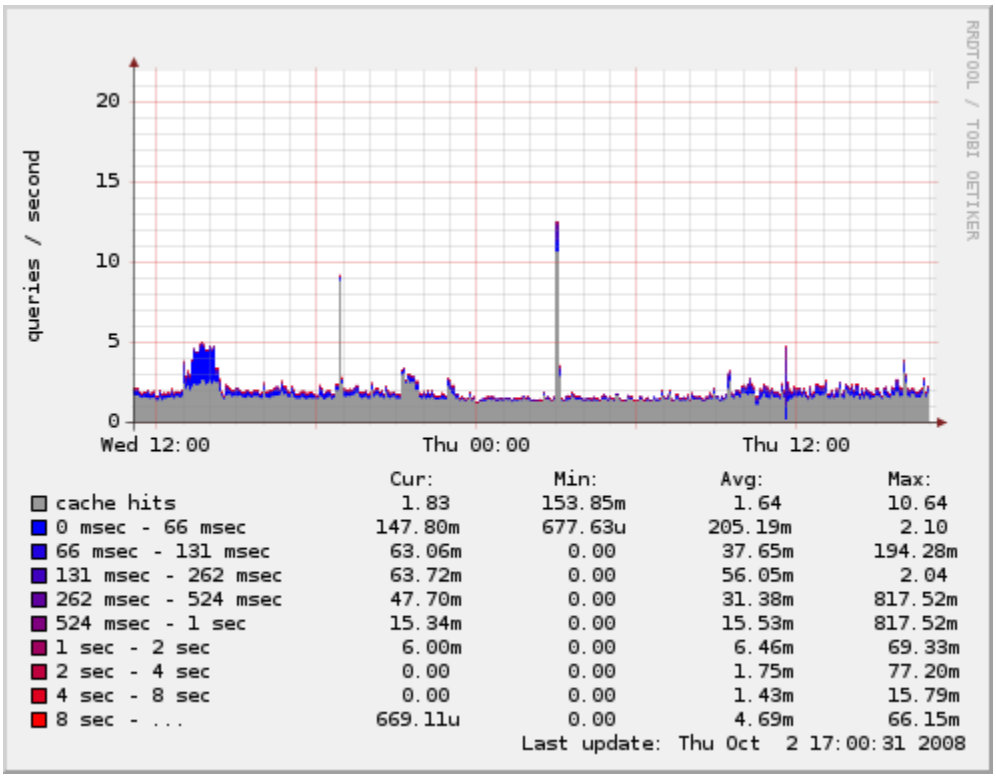
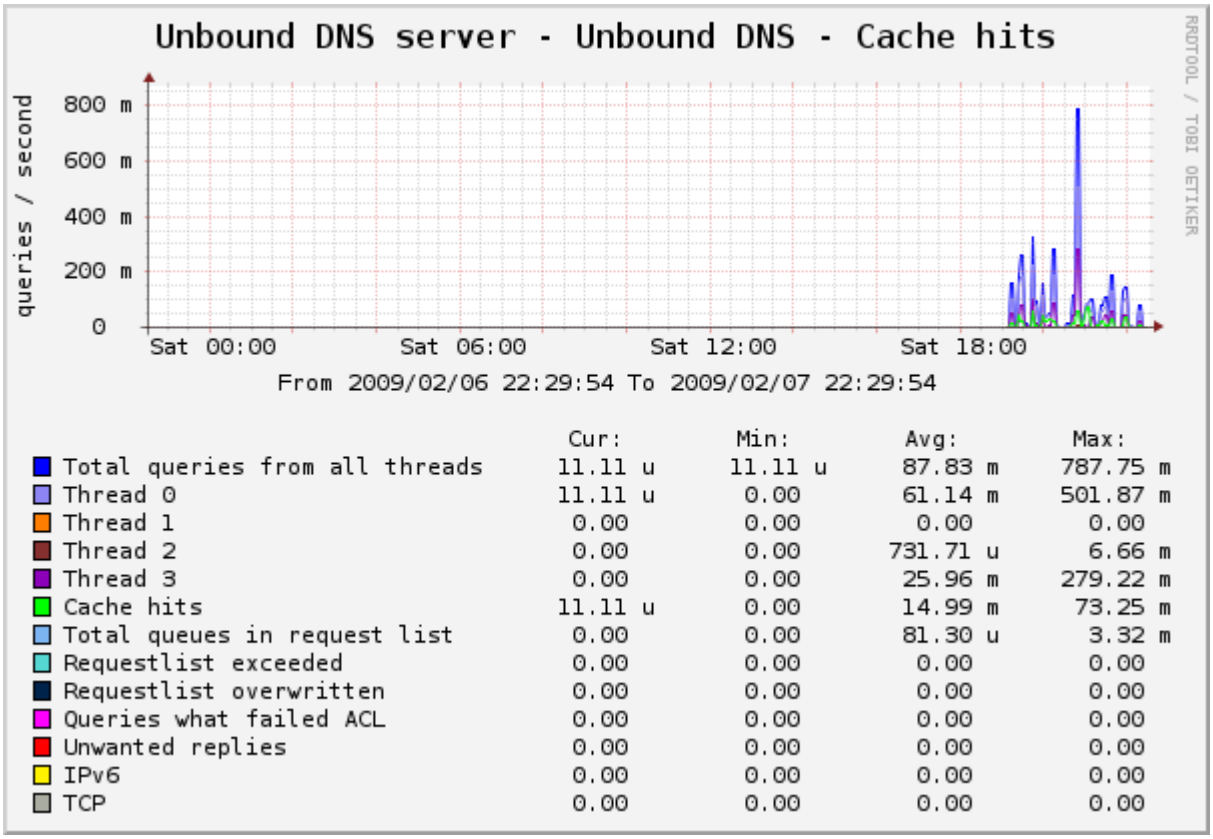


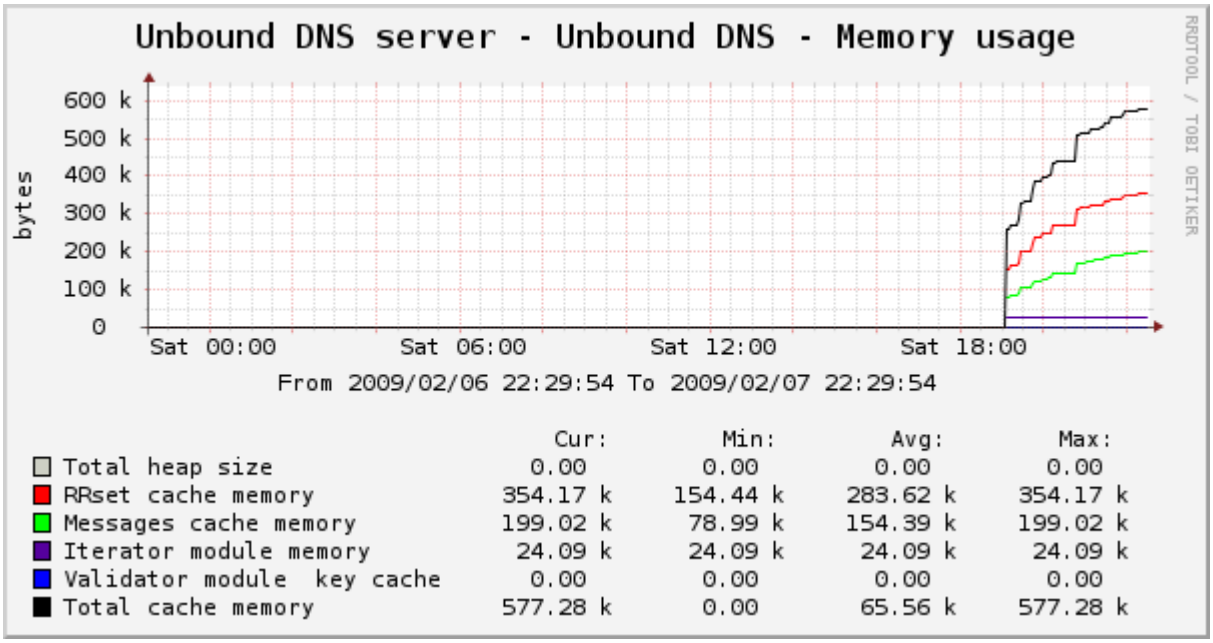
Fig. 9: unbound_munin_histogram - histogram of query resolving times

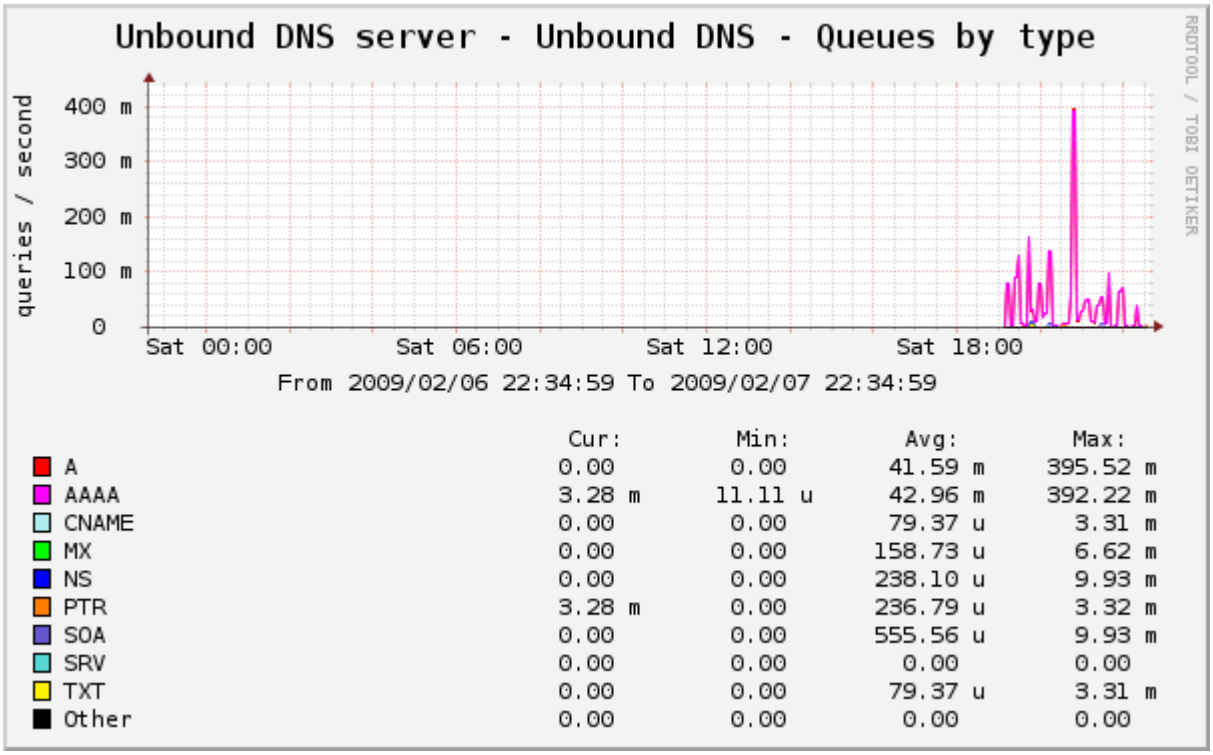
8.3 Statistics with Cacti

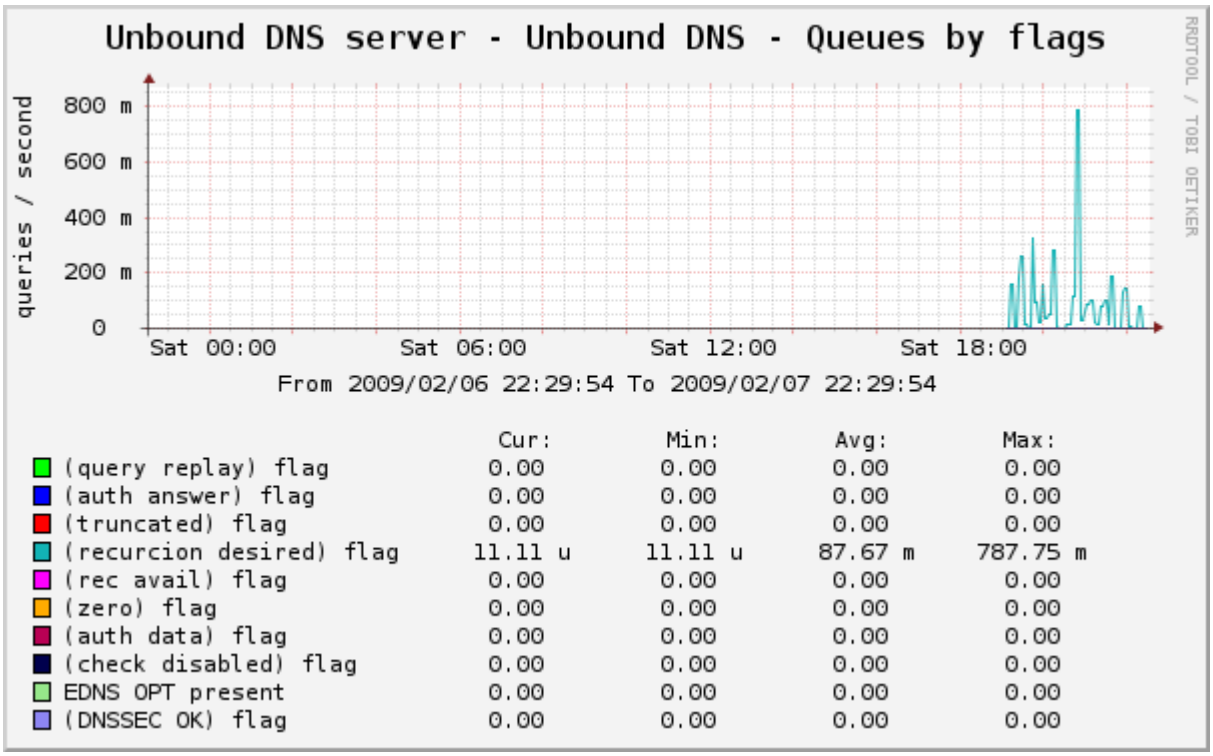
The setup is described in the README in the tarball in the Unbound source `contrib` directory: `contrib/unbound_cacti.tar.gz` (contributed by Dmitriy Demidov).

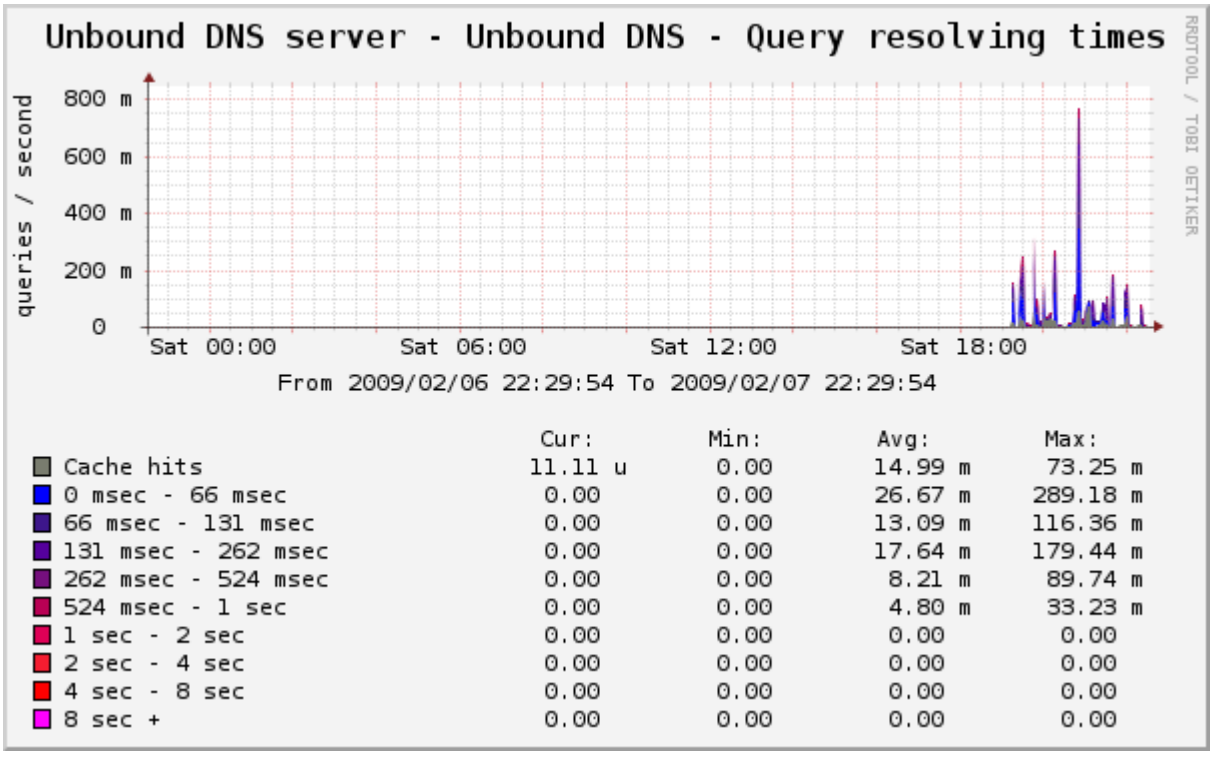
Example output from unbound cacti statistics:

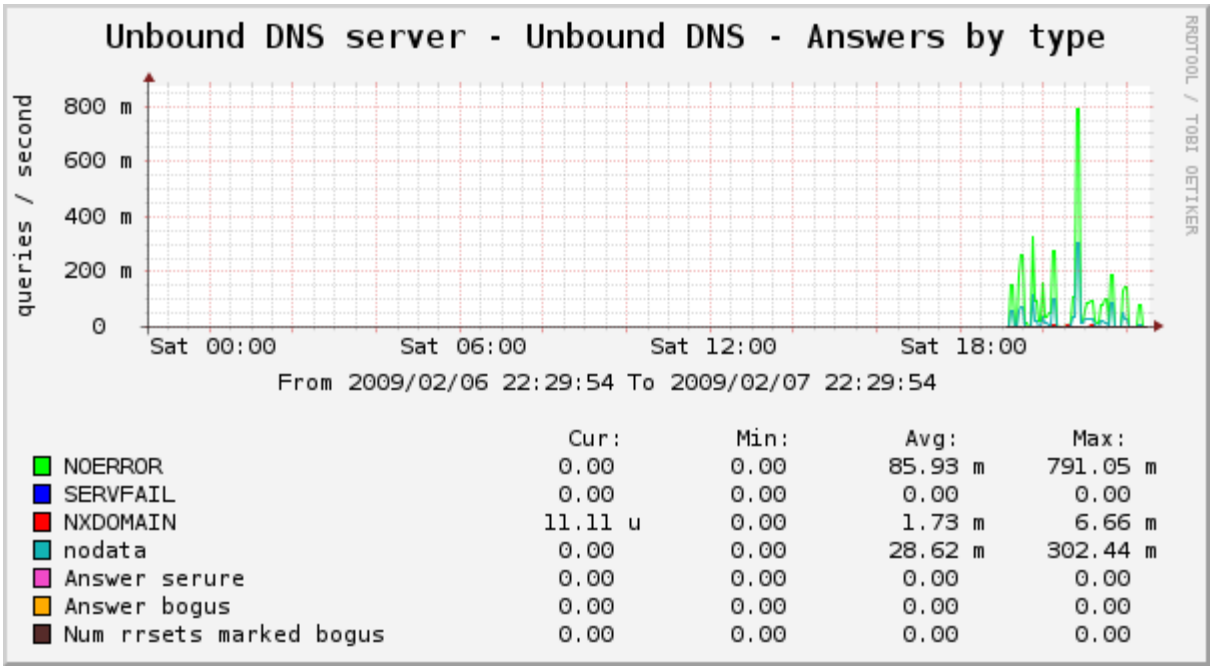



Unbound DNS server - Unbound DNS - Memory usage
RRDTOOL / TOBI OETIKER
bytes
600 k
500 k
400 k
300 k
200 k
100 k
0
Sat 00:00
Sat 06:00
Sat 12:00
Sat 18:00
From 2009/02/06 22:29:54 To 2009/02/07 22:29:54
Cur: Min: Avg: Max:
Total heap size 0.00 0.00 0.00 0.00
RRset cache memory 354.17 k 154.44 k 283.62 k 354.17 k
Messages cache memory 199.02 k 78.99 k 154.39 k 199.02 k
Iterator module memory 24.09 k 24.09 k 24.09 k 24.09 k
Validator module key cache 0.00 0.00 0.00 0.00
Total cache memory 577.28 k 0.00 65.56 k 577.28 k


Unbound DNS server - Unbound DNS - Queues by type
RRDTOOL / TOBI OETIKER
queries / second
400 m
300 m
200 m
100 m
0
Sat 00:00
Sat 06:00
Sat 12:00
Sat 18:00
From 2009/02/06 22:34:59 To 2009/02/07 22:34:59
Cur: Min: Avg: Max:
A 0.00 0.00 41.59 m 395.52 m
AAAA 3.28 m 11.11 u 42.96 m 392.22 m
CNAME 0.00 0.00 79.37 u 3.31 m
MX 0.00 0.00 158.73 u 6.62 m
NS 0.00 0.00 238.10 u 9.93 m
PTR 3.28 m 0.00 236.79 u 3.32 m
SOA 0.00 0.00 555.56 u 9.93 m
SRV 0.00 0.00 0.00 0.00
TXT 0.00 0.00 79.37 u 3.31 m
Other 0.00 0.00 0.00 0.00


Unbound DNS server - Unbound DNS - Queues by flags
queries / second
800 m
600 m
400 m
200 m
0
Sat 00:00
Sat 06:00
Sat 12:00
Sat 18:00
From 2009/02/06 22:29:54 To 2009/02/07 22:29:54
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
(query replay) flag 0.00 0.00 0.00 0.00
(auth answer) flag 0.00 0.00 0.00 0.00
(truncated) flag 0.00 0.00 0.00 0.00
(recurcion desired) flag 11.11 u 11.11 u 87.67 m 787.75 m
(rec avail) flag 0.00 0.00 0.00 0.00
(zero) flag 0.00 0.00 0.00 0.00
(auth data) flag 0.00 0.00 0.00 0.00
(check disabled) flag 0.00 0.00 0.00 0.00
EDNS OPT present 0.00 0.00 0.00 0.00
(DNSSEC OK) flag 0.00 0.00 0.00 0.00


Unbound DNS server - Unbound DNS - Query resolving times
queries / second
800 m
600 m
400 m
200 m
0
Sat 00:00
Sat 06:00
Sat 12:00
Sat 18:00
From 2009/02/06 22:29:54 To 2009/02/07 22:29:54
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
Cache hits 11.11 u 0.00 14.99 m 73.25 m
0 msec - 66 msec 0.00 0.00 26.67 m 289.18 m
66 msec - 131 msec 0.00 0.00 13.09 m 116.36 m
131 msec - 262 msec 0.00 0.00 17.64 m 179.44 m
262 msec - 524 msec 0.00 0.00 8.21 m 89.74 m
524 msec - 1 sec 0.00 0.00 4.80 m 33.23 m
1 sec - 2 sec 0.00 0.00 0.00 0.00
2 sec - 4 sec 0.00 0.00 0.00 0.00
4 sec - 8 sec 0.00 0.00 0.00 0.00
8 sec + 0.00 0.00 0.00 0.00


Unbound DNS server - Unbound DNS - Answers by type
queries / second
800 m
600 m
400 m
200 m
0
Sat 00:00
Sat 06:00
Sat 12:00
Sat 18:00
From 2009/02/06 22:29:54 To 2009/02/07 22:29:54
RRDTOOL / TOBI OETIKER
Cur: Min: Avg: Max:
NOERROR 0.00 0.00 85.93 m 791.05 m
SERVFAIL 0.00 0.00 0.00 0.00
NXDOMAIN 11.11 u 0.00 1.73 m 6.66 m
nodata 0.00 0.00 28.62 m 302.44 m
Answer serure 0.00 0.00 0.00 0.00
Answer bogus 0.00 0.00 0.00 0.00
Num rrsets marked bogus 0.00 0.00 0.00 0.00

CHAPTER

NINE

AGGRESSIVE NSEC

Unbound has implemented the aggressive use of the DNSSEC-Validated cache, also known as *Aggressive NSEC*, based on **RFC 8198**. This section first describes how NSEC works, and then covers how synthesised answers can be generated based on the DNSSEC-Validated cache.

9.1 Introduction

DNS relies heavily on caching. A lot of performance can be gained by storing answers to previous queries close to the client. If an authoritative name server would have to be queried for every single request, performance would be severely impacted.

In addition to caching the positive answer to queries, negative answers are also cached. These negative answers are an acknowledgement from the name server that a name does not exist (an answer with the response code set to NXDOMAIN) or that the type in the query does not exist for the name in the query. The latter is known as an answer with the NODATA pseudo response code, as specified in **RFC 2308#section-1**.

9.2 NSEC (Next Secure) Records

DNSSEC is not only used to prove the authenticity of records in a DNS answer by verifying the DNSSEC signatures of the records, it is also used to prove the absence of records. DNSSEC uses NSEC (next secure), as well as NSEC3 records for these proof of non-existence answers. An NSEC record indicates that there are no records that are sorted between the two domain names it contains. The canonical DNS name order is used for the sorting, as described in **RFC 4034#section-6.1**. An NSEC record also has a type bitmap which specifies the record types that exist for the owner name of the NSEC record. Like any other DNS record, the authenticity of NSEC records can be validated using its DNSSEC signature which is located in the RRSIG record.

This NSEC record is taken as an example:

```
golf.example.net NSEC kilo.example.net TXT RRSIG NSEC
```

The record indicates that the owner name `golf.example.net` exists and that the owner name has records for the *TXT*, *RRSIG* and *NSEC* types. It therefore proves that there is no *A* record for `golf.example.net`. This NSEC record also proves that there are no records alphabetically sorted between its owner and its next domain name (kilo.example.net). This record therefore proves that there is no record for juliett.example.net.

9.3 DNSSEC Signatures on Wildcard Records

Wildcard expansion on NSEC record is specifically allowed by **RFC 4592**. In order to answer a DNS query using a wildcard record, an authoritative nameserver replaces the owner name of the wildcard record with the name in the query. DNSSEC is designed in such way that it can sign a complete zone before it starts serving. Because the query name that will be used for the wildcard record is not known when the zone is being signed, it is not possible to make a DNSSEC signature for it. Therefore the original owner name with the wildcard label is used for the signature.

The labels field that is part of the RRSIG record indicates the number of labels of the owner name without the wildcard label. This labels field can be used by a DNSSEC validator to detect that this is a signature for a wildcard record. A DNSSEC validator then knows it needs to validate the signature using the original wildcard owner, and not the expanded owner that matches the query name. A validator gets the original owner name by taking the number of rightmost labels defined in the labels field from the expanded owner and then prepend it with the wildcard label *.

This is an example of a wildcard expanded RRSIG record, with the signature omitted to keep the text compact:

```
zebra.example.net.  2710 IN RRSIG NSEC 8 2 10200 20180226143900 20180129143900 42393␣
↪example.net. [..]
```

This RRSIG record has a label count of two, while the number of labels in the owner name (excluding the root label) is three. Using that information a validator will take the two last labels of the owner name (example.net) and prepend the wildcard label to it. It therefore checks the signature using the original wildcard name, which is `*.example.net`.

9.4 Generating NODATA Answers

The traditional Unbound cache implementation is based on exactly matching cached messages to the query name, query type and query class. If a client asks for a *TXT* record for `example.net`, the resolver will search the cache and if that fails go and look up the answer at the authoritative name server. This query to the authoritative name server will result in a response containing the existing *TXT* record. If the resolver now receives a query for the same name but for the *TLSA* type, the resolver will check its cache, in this case can not find a matching record in the cache and will, as a result, send a query to the authoritative name server. That name server will now reply with a NODATA answer, indicating that the `example.net` name does exist, but there is no record for that name with the TLSA record. A third query for the same name for another non-existing type, for example *SRV*, will once again not result in a cache hit and will generate yet another query with again a NODATA answer as result.

In this example the `example.net` zone is DNSSEC signed. This means the absence of these records need to be proven using NSEC records. NSEC records indicate which types exist for a name and which names exist in a zone. NSEC records have a cryptographic signature which make them tamper proof. By knowing the existing record and types in a zone, a DNSSEC validator can prove that the combination of query name and query type indeed does not exist.

The NODATA answer for the `example.net` name with the *TLSA* query type could for example contain this NSEC record:

```
example.net. 3600 IN NSEC !.example.net. A NS SOA MX TXT AAAA NAPTR RRSIG NSEC DNSKEY
```

This record proves which types exist for `example.net` (*A, NS, SOA* etc.) and thereby proves that the *TLSA* record indeed does not exist. The NODATA response to the third query in above example (the *SRV* query for `example.net`) will contain exactly the same NSEC record to prove the absence of the *SRV* record. Because this NSEC record was already cached after the lookup for the TLSA record we could have used that already obtained NSEC record to generate a DNSSEC secure answer, without the need to send another query to the authoritative name server.

Important: To use previously cached NSEC records to generate responses in Unbound, use the *aggressive-nsec* option in the configuration file:

```
aggressive-nsec: yes
```

9.5 Generating NXDOMAIN Answers

An answer with the NXDOMAIN response code indicates that a name does not exist at all, which is also proven using an NSEC record. If `example.net` would contain these alphabetically sorted records (some simplification ahead):

```
example.net.           IN SOA [..]
                       IN NS alfa.example.net.
alfa.example.net.      IN A 198.51.100.52
sierra.example.net.    IN A 198.51.100.98
```

then DNSSEC would make sure these NSEC records are inserted and signed:

```
example.net.         IN NSEC alfa.example.net.   NS SOA DNSKEY
alfa.example.net.    IN NSEC sierra.example.net. A
sierra.example.net.  IN NSEC example.net.        A
```

They attest that no name exists between `alfa.example.net` and `sierra.example.net`. So if you query for `lima.example.net`, you will get back the NXDOMAIN from the authoritative name server, as well as the NSEC record for `alfa.example.net` — `sierra.example.net` as proof that the query name does not exist and the NSEC record for `example.net` — `alfa.example.net` as proof that the `*.example.net` wildcard record does not exist.

If the user now queries for for `delta.example.net`, resolvers would normally ask the authoritative server again because there is no message cached for that name. But because the NSEC records for `alfa.example.net` — `sierra.example.net` and `example.net` — `alfa.example.net` are already cached, the implementation of **RFC 8198** will allow Unbound to deduce that it doesn't need to send a new query. It is already able to prove that the name doesn't exist and immediately, or *aggressively* if you will, returns an NXDOMAIN answer.

9.6 Generating Wildcard Answers

There is one more type of message that can be generated using cached NSEC records, namely wildcard answers. A DNSSEC validator only accepts a wildcard answer when there is proof that there is no record for the query name. When we have this zone containing a wildcard record:

```
example.net.          IN SOA [..]
                      IN NS alfa.example.net.
*.example.net.        IN TXT "A wildcard record"
alfa.example.net.     IN A 198.51.100.52
sierra.example.net.   IN A 198.51.100.98
```

then a TXT query for `delta.example.net` will be answered with the following records, indicating that there is no direct match for the query name but that there is a matching wildcard record:

```
;; ANSWER SECTION:
delta.example.net.    IN TXT "A wildcard record"
delta.example.net.    IN RRSIG TXT 8 2 [..]

;; AUTHORITY SECTION:
```

(continues on next page)

(continued from previous page)

```
alfa.example.net.     IN NSEC sierra.example.net.   A
```

The `alfa.example.net` — `sierra.example.net` NSEC record indicates that there is no `delta.example.net` record. The labels field in the signature indicates that the returned TXT record is expanded using the `*.example.net` record.

Unbound uses this knowledge to store the wildcard RRset also under the original owner name, containing the wildcard record, when aggressive use of NSEC is enabled. After receiving a query for `echo.example.net`, Unbound finds the NSEC record proving the absence in its cache. Unbound will then look in the cache for a `*.example.net` *TXT* record, which also exists. These records are then used to generate an answer without sending a query to the name server.

Note: Aggressive NSEC can result in a reduction of traffic on all levels of the DNS hierarchy but it will be most noticeable at the root, as typically more than half of all responses are NXDOMAIN.

Another benefit of a wide deployment of aggressive NSEC is the incentive to DNSSEC sign your zone. If you don't want to have a large amount of queries for non-existing records at your name server, signing your zone will prevent this.

New in version 1.12.0.

CHAPTER

TEN

DNS-OVER-HTTPS

DNS-over-TLS (DoT) makes it possible to encrypt DNS messages and gives a DNS client the possibility to authenticate a resolver. As implied by the name, this is done by sending DNS messages over TLS. Unbound can handle TLS encrypted DNS messages since 2011, long before the IETF DPRIVE working group started its work on the DoT specification.

There are, however, DNS clients that do not support DoT but are able to use DNS-over-HTTPS (DoH) instead. Where DoT sends a DNS message directly over TLS, DoH has an HTTP layer in between. Where DoT uses its own TCP port (853), DoH uses the standard HTTPS port (443).

By adding downstream DoH support to Unbound we hope to increase the ratio of encrypted DNS traffic and increase the number of resolvers that offer encrypted services in home networks, enterprise networks, ISPs, and public resolvers.

10.1 Implementation Details

The DoH implementation in Unbound requires TLS, and only works over HTTP/2. The query pipelining and out-of-order processing functionality that is provided by HTTP/2 streams is needed to be able to provide performance that is on par with DoT. The HTTP/2 capability is negotiated using Application-Layer Protocol Negotiation (ALPN) TLS extension, which is supported in OpenSSL from version 1.0.2 onward.

Unbound uses the nghttp2 library to handle the HTTP/2 framing layer. This library does not take care of any I/O handling, which makes it possible to easily integrate it in the existing Unbound event loop and TCP handling. Adding HTTP/2 on top of the existing TCP code makes it possible to also use the existing TCP configuration options for the DoH connections. These existing options include the number of allowed incoming TCP connections, the TCP timeout settings, and the limits on TCP connections per client IP address or netblock.

The use of HTTP makes it possible to change the DNS message format by using new media types. Unbound currently only supports the `application/dns-message` media type, as this is the only format standardised in the IETF standards track, and the only supported format by popular DNS clients. We are keeping an eye on the new possibilities here, such as using the `application/oblivious-dns-message` media type.

The use of the HTTP layer also makes it possible to return more detailed information to a client in case of malformed requests. This can be done by using a non-successful HTTP status code, or by closing an individual stream by sending an RST_STREAM frame. The HTTP status codes that can be returned by Unbound are:

200 OK
: Unbound is able to process the query, and return an answer. This could be a negative answer or an error like SERVFAIL or FORMERR.

404 Not Found
: The request is directed to a path other than the configured endpoint in http-endpoint (default `/dns-query`).

413 Payload Too Large
: The payload received in the POST request is too large. Payloads cannot be larger than the content-length com-

municated in the request header. The payload length is limited to 512 bytes if *harden-large-queries:* is enabled, and otherwise limited to the value configured in *msg-buffer-size:* (default 65552 bytes). To prevent the allocation of overly large buffers, the maximum size is limited to the size of the first DATA frame if no content-length is received in the request.

414 URI Too Long
The base64url encoded DNS query in the GET request is too large. The DNS query length is limited to 512 bytes if *harden-large-queries:* is enabled, and limited to *msg-buffer-size:* otherwise.

415 Unsupported Media Type
The media type of the request is not supported. This happens if the request contains a content-type header that is set to anything but `application/dns-message`. Requests without content-type will be treated as `application/dns-message`.

400 Bad Request
No valid query received, not matched by any of the above 4xx status codes.

501 Not Implemented
The method used in the request is not GET or POST.

10.2 Using DoH

As mentioned above, the nghttp2 library is required to use Unbound's DoH functionality. Compiling and installing Unbound with libnghttp2 can be done using:

```
./configure --with-libnghttp2
make && make install
```

Enabling DoH in Unbound is as simple as configuring the TLS certificate and the corresponding private key that will be used for the connection, and configuring Unbound to listen on the HTTPS port:

```
server:
    interface: 127.0.0.1@443
    tls-service-key: "key.pem"
    tls-service-pem: "cert.pem"
```

The port that Unbound will use for incoming DoH traffic is by default set to 443 and can be changed using the *https-port:* configuration option.

`dohclient`, an Unbound test utility which can be built with `make dohclient` in Unbound's source tree, shows that Unbound is now ready to handle DoH queries on the default HTTP endpoint, which is `/dns-query`:

```
$ ./dohclient -s 127.0.0.1 nlnetlabs.nl AAAA IN
Request headers
:method: GET
:path: /dns-query?dns=AAABAAABAAAAAAAABCW5sbmV0bGFicwJubAAAHAABAAApEAAAAIAAAAA
:scheme: https
:authority: 127.0.0.1
content-type: application/dns-message
:status 200
content-type application/dns-message
;; ->>HEADER<<- opcode: QUERY, rcode: NOERROR, id: 0
;; flags: qr rd ra ad ; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; QUESTION SECTION:
nlnetlabs.nl. IN AAAA
```

(continues on next page)

(continued from previous page)

```
;; ANSWER SECTION:
nlnetlabs.nl. 10200 IN AAAA 2a04:b900::1:0:0:10
nlnetlabs.nl. 10200 IN RRSIG AAAA 8 2 10200 20200723194739 20200625194739 42393␣
↪nlnetlabs.nl. ML5NkbykTetqBPyA0xG5fuq1t/0ojsMUixgEhcewG93jZpF+vz8WhVo6czzdRMo/
↪qq2kAmh3aFmU94wVWn+AULEEz6a/7B1Sxz9O+bXivZiWVitUopheSya68CNHO/
↪zCl7j23QirecLGoXozbVqMIbinqG0LS32bHS+WOsJgQCQ= ;{id = 42393}

;; AUTHORITY SECTION:

;; ADDITIONAL SECTION:
; EDNS: version: 0; flags: do ; udp: 4096
;; MSG SIZE  rcvd: 241
```

Queries to other paths will be answered with a `404` status code. The endpoint can be changed using the *http-endpoint:* configuration option.

The maximum number of concurrent HTTP/2 streams can be configured using the *http-max-streams:* configuration option. The default for this option is 100, as per HTTP/2 RFC recommended minimum. This value will be in the `SETTINGS` frame sent to the client, and enforced by Unbound.

Because requests can be spread out over multiple HTTP/2 frames, which can be interleaved between frames of different streams, we have to create buffers containing partial queries. A new counter is added to Unbound to limit the total memory consumed by all query buffers. The limit can be configured using the *http-query-buffer-size:* option. New streams will be closed by sending an `RST_STREAM` frame when this limit is exceeded.

After Unbound is done resolving a request the DNS response will be stored in a buffer, waiting until Unbound is ready to sent them back to the client using HTTP. These buffers also have a maximum amount of memory they are allowed to consume. This maximum is configurable using the *http-response-buffer-size:* configuration option.

10.3 Metrics

Three DoH related metrics are available in Unbound; *num.query.https* counts the number of queries that have been serviced using DoH. The *mem.http.query_buffer*, and *mem.http.response_buffer* counters keep track of the memory used for the DoH query and response buffers.

CHAPTER

ELEVEN

TAGS AND VIEWS

The tags and views functionality make it possible to send specific DNS answers based on the IP address of the client.

11.1 Tags

The tags functionality makes it possible to divide incoming client queries in categories (tags), and use *local-zone:* and *local-data:* information for these specific tags.

Before these tags can be used, you need to define them in the Unbound configuration using *define-tag:*. In this example, a tag for domains containing malware is set, along with one for domains of gambling sites:

```
define-tag: "malware gambling"
```

Now that Unbound is aware of the existing tags, it is possible to start using them.

The *access-control-tag:* element is used to specify the tag to use for client source address. Alternatively, the *interface-tag:* element is used to specify the tag to use for clients on a specific listening interface. You can add multiple tags to these elements:

```
# Per client IP ...
access-control-tag: 10.0.1.0/24 "malware"
access-control-tag: 10.0.2.0/24 "malware"
access-control-tag: 10.0.3.0/24 "gambling"
access-control-tag: 10.0.4.0/24 "malware gambling"

# ... and/or per listening interface
interface-tag: eth0 "malware"
interface-tag: 10.0.0.1 "malware gambling"
```

Note: Any `access-control*:` setting overrides all `interface-*:` settings for targeted clients.

Unbound will create a `*-tag` element with the "allow" type if the IP address block / listening interface in the `*-tag` element does not match an existing access control rule.

When a query comes in that is marked with a tag, Unbound starts searching its local-zone tree for the best match. The best match is the most specific local-zone with a matching tag, or without any tag. That means that local-zones without any tag will be used for all queries and tagged local-zones only for queries with matching tags.

Adding tags to local-zones can be done using the *local-zone-tag:* element:

```
local-zone: malwarehere.example refuse
local-zone: somegamblingsite.example static
local-zone: matchestwotags.example transparent
local-zone: notags.example inform

local-zone-tag: malwarehere.example malware
local-zone-tag: somegamblingsite.example malware
local-zone-tag: matchestwotags.example "malware gambling"
```

A local-zone can have multiple tags, as illustrated in the example above. The tagged local-zones will be used if one or more tags match the query. So, the matchestwotags.example local-zone will be used for all queries with at least the malware or gambling tag. The used local-zone type will be the type specified in the matching local-zone. It is possible to depend the local-zone type on the client and tag combination. Setting tag specific local-zone types can be done using *access-control-tag-action:* and/or *interface-tag-action:*:

```
# Per client IP ...
access-control-tag-action: 10.0.1.0/24 "malware" refuse
access-control-tag-action: 10.0.2.0/24 "malware" deny

# ... and/or per listening interface
interface-tag-action: eth0 "malware" refuse
interface-tag-action: 10.0.0.1 "malware" deny
```

In addition to configuring a local-zone type for specific clients/tag match, it is also possible to set the used local-data RRs. This can be done using the *access-control-tag-data:* and/or *interface-tag-data:* elements:

```
# Per client IP ...
access-control-tag-data: 10.0.4.0/24 "gambling" "A 127.0.0.1"

# ... and/or per listening interface
interface-tag-data: 10.0.0.1 "gambling" "A 127.0.0.1"
```

Sometimes you might want to override a local-zone type for a specific IP prefix or interface, regardless the type configured for tagged and untagged local zones, and regardless the type configured using *access-control-tag-action:* and/or *interface-tag-action:*. This override can be done using *local-zone-override:*.

11.2 Views

Tags make is possible to divide a large number of local-zones in categories, and assign these categories to a large number of IP address blocks. As tags on the clients and local-zones are stored in bitmaps, it is advised to keep the number of tags low. Specifically for client prefixes (i.e., `access-control-tag*:`), if a lot of clients have their own local-zones, without sharing these to other IP prefixes, it can result in a large amount tags. In this situation it is more convenient to give the clients' IP prefix its own tree containing local-zones. Another benefit of having a separate local zone tree is that it makes it possible to apply a local-zone action to a part of the domain space, without having other local-zone elements of subdomains overriding this. Configuring a client specific local-zone tree can be done using views.

A view is a named list of configuration options. The supported view configuration options are *local-zone:* and *local-data:*.

A view is configured using a **view:** clause. There may be multiple view clauses, each with a unique name. For example:

```
view:
    name: "firstview"
    local-zone: example.com inform
    local-data: 'example.com TXT "this is an example"'
    local-zone: refused.example.nl refuse
```

Mapping a view to a client can be done using the *access-control-view:* element:

```
access-control-view: 10.0.5.0/24 firstview
```

Alternatively, mapping a view to clients in a specific interface can be done using the *interface-view:* element:

```
interface-view: eth0 firstview
```

By default, view configuration options override the global configuration outside the view. When a client matches a view it will only use the view's local-zone tree. This behaviour can be changed by setting *view-first:* to yes. If view-first is enabled, Unbound will try to use the view's local-zone tree, and if there is no match it will search the global tree.

See also:

View Options in the *unbound.conf(5)* manpage.

New in version 1.10.0: Intial release with QNAME and Response IP Address triggers

Changed in version 1.14.0: Full set of RPZ triggers and actions are supported

CHAPTER

TWELVE

RESPONSE POLICY ZONES

Response Policy Zones (RPZ) is a mechanism that makes it possible to define your local policies in a standardised way and load your policies from external sources.

12.1 Introduction

Unbound has support for local-zone and local-data. This makes it possible to give a custom answer back for specified domain names. It also contains the `respip` module which makes it possible to rewrite answers containing specified IP addresses. Although these options are heavily used by users, they are Unbound specific. If you operate multiple resolvers from multiple vendors you have to maintain your policies for multiple configurations, which all will have their own syntax. Using the Unbound specific configuration also makes it challenging to consume policies from external sources.

To get these external sources to work manually, you have to fetch the external policies in the offered format, reformat it in such a way that Unbound will understand, and keep this list up-to-date, for example using *unbound-control(8)*.

To automate this process in a generic, standardised way, Response Policy Zones (RPZ) is a policy format that will work on different resolver implementations, and that has capabilities to be directly transferred and loaded from external sources.

We'll first discuss the different policies and RPZ actions with examples, and then show how to implement RPZ in a configuration.

12.2 RPZ Policies

RPZ policies are formatted in DNS zone files. This makes it possible to easily consume and keep them to up-to-date by using DNS zone transfers. Something that Unbound is already capable of doing for its *auth-zone* feature.

Each policy in the policy zone consists of a trigger and an action. The trigger describes when the policy should be applied. The action describes what action should be taken if the policy needs to be applied. Each trigger and action combination is defined as a Resource Record (RR) in the policy zone. The owner of the RR states the trigger, the type and RDATA state the action.

Unbound supports all the RPZ policies described in the RPZ internet draft:

Trigger	Description and example
QNAME	The query name: example.com
Client IP Address	The IP address of the client: 24.0.2.0.192.rpz-client-ip
Response IP Address	response IP address in the answer: 24.0.2.0.192.rpz-ip
NSDNAME	The nameserver name: ns.example.com.rpz-nsdname
NSIP	The nameserver IP address: 24.0.2.0.192.rpz-nsip

Note that the IP address encoding for RPZ triggers in the IN-ADDR.ARPA naming convention. So 192.0.2.24 will be written as 24.2.0.192.

In the implementation step we will go trough all the triggers.

12.3 RPZ Actions

Aside from RPZ triggers, RPZ also specifies actions as a result of these triggers. Unbound currently supports the following actions: **NXDOMAIN**, **NODATA**, **PASSTHRU**, **DROP**, **Local Data**, and **TCP-only**.

The **Local Data** action responds with a preconfigured resource record. Queries for types that do not exist in the policy zones will result in a NODATA answer.

Other RPZ actions that are supported by Unbound are the **NXDOMAIN**, **NODATA**, **PASSTHRU**, **DROP** and **TCP-Only** actions. All of these actions are defined by having a CNAME to a specific name.

The CNAME targets for the other RPZ actions are:

Action	RR type and RDATA
NXDOMAIN	CNAME .
NODATA	CNAME *.
PASSTHRU	CNAME rpz-passthru.
DROP	CNAME rpz-drop.
TCP-Only	CNAME rpz-tcp-only.

The **NODATA** action returns a response with no attached data. The **DROP** action ignores (drops) the query. The **TCP-Only** action responds to the query over TCP. The **PASSTHRU** action makes it possible to exclude a domain, or IP address, from your policies so that if the **PASSTHRU** action is triggered no other policy from any of the available policy zones will be applied.

12.4 How to use RPZ with Unbound

The RPZ implementation in Unbound depends on the respip module, this module needs to be loaded using *module-config:*. Each policy zone is configured in Unbound using the **rpz:** clause. The full documentation for RPZ in Unbound can be found in the *unbound.conf(5)*. A minimal configuration with a single policy zone can look like the following, where additional elements can be uncommented:

```
server:
    module-config: "respip validator iterator"
rpz:
    # The name of the RPZ authority zone
    name: rpz.nlnetlabs.nl
```

(continues on next page)

(continued from previous page)

```
    # The filename where the zone is stored. If left empty
    zonefile: rpz.nlnetlabs.nl

    # The location of the remote RPZ zonefile.
    # url: http://www.example.com/example.org.zone (not a real RPZ file)

    # Always use this RPZ action for matching triggers from this zone.
    # Possible action are: nxdomain, nodata, passthru, drop, disabled,
    # and cname.
    # rpz-action-override: nxdomain

    # Log all applied RPZ actions for this RPZ zone. Default is no.
    # rpz-log: yes

    # Specify a string to be part of the log line.
    # rpz-log-name: nlnetlabs
```

In above example the policy zone will be loaded from the file `rpz.nlnetlabs.nl`. An example RPZ file with all the triggers and actions looks like this:

```
$ORIGIN rpz.nlnetlabs.nl.

# QNAME trigger with local data action
example.com.rpz.nlnetlabs.nl.    TXT  "trigger for example.com"
*.example.com               CNAME   .

# IPv4 subnet (192.0.2.0/28) which drops clients and IPv6 subnet
(2001:db8::3/128) which is not subject to policy
28.0.2.0.192.rpz-client-ip      CNAME rpz-drop.
128.3.zz.db8.2001.rpz-client-ip CNAME rpz-passthru.
# Clients at 192.2.0.64 only get responses over TCP.
64.2.0.192.rpz-client-ip        CNAME rpz-tcp-only.

# Fills the responses for these queries with NXDOMAIN and the correct
# answers respectively
24.0.2.0.192.rpz-ip         CNAME   .
32.2.2.0.192.rpz-ip         CNAME   rpz-passthru.

# Answers queries for the nlnetlabs.nl nameserver with NXDOMAIN
ns.nlnetlabs.nl.rpz-nsdname CNAME   .

# Drops queries for the nameserver at 192.0.2.0/24 subnet
24.0.2.0.192.rpz-nsip       CNAME   rpz-drop.
```

It is also possible to load the zone using DNS zone transfers. Both AXFR and IXFR is supported, all additions and deletion in the zone will be picked up by Unbound and reflected in the local policies. Transferring the policy using a DNS zone transfer is as easy as specifying the server to get the zone from:

```
server:
    module-config: "respip validator iterator"
rpz:
```

(continues on next page)

(continued from previous page)

```
    name: rpz.nlnetlabs.nl
    master: <ip address of server to transfer from>
    zonefile: rpz.nlnetlabs.nl
```

The zone will now be transferred from the configured address and saved to a zonefile on disk. It is possible to have more than one policy zone in Unbound. Having multiple policy zones is as simple as having multiple **rpz:** clauses:

```
server:
    module-config: "respip validator iterator"
rpz:
    name: rpz.nlnetlabs.nl
    zonefile: rpz.nlnetlabs.nl
rpz:
    name: rpz2.nlnetlabs.nl
    zonefile: rpz2.nlnetlabs.nl
```

The policy zones will be applied in the configured order. In the example, Unbound will only look at the `rpz2.nlnetlabs.nl` policies if there is no match in the `rpz.nlnetlabs.nl` zone. If there is no match in any of the configured zones Unbound will continue to resolve the domain by sending upstream queries. Note that a PASSTHRU action is considered a match, having that action in the first zone will therefore stop Unbound from looking further at other policy zones.

Unbound has the possibility to override the actions that will be used for policies in a zone that matches the zone's triggers. This can be done using the *rpz-action-override:* configuration option. The possible values for the option are: `nxdomain`, `nodata`, `passthru`, `drop`, `disabled`, and `cname`. The first four options of this list will do the same as the RPZ actions with the same name.

The `cname` override option will make it possible to apply a local data action using a CNAME for all matching triggers in the policy zone. The CNAME to use in the answer can be configured using the *rpz-cname-override:* configuration option. Using these overrides is nice if you use an external feed to get a list of triggers, but would like to redirect all your users to your own domain:

```
RPZ zone (rpz.nlnetlabs.nl):
$ORIGIN rpz.nlnetlabs.nl.
drop.example.com.rpz.nlnetlabs.nl. CNAME rpz-drop.
32.34.216.184.93.rpz-ip.rpz.nlnetlabs.nl. A 192.0.2.1
```

This also requires a change in the Unbound configuration:

```
server:
    module-config: "respip validator iterator"

rpz:
    name: rpz.nlnetlabs.nl
    zonefile: rpz.nlnetlabs.nl
    rpz-action-override: cname
    rpz-cname-override: "example.nl."
```

The `disabled` option will stop Unbound from applying any of the actions in the zone. This, combined with the `rpz-log` option, is a nice way to test what would happen to your traffic when a policy will be enabled, without directly impacting your users. The difference between `disabled` and `passthru` is that disabled is not considered to be a valid match and will therefore not stop Unbound from looking at the next configured policy zone.

When *rpz-log:* is set to yes, Unbound will log all applied actions for a policy zone. With `rpz-log` enabled you can specify a name for the log using *rpz-log-name:*, this way you can easily find all matches for a specific zone. It is also

possible to get statistics per applied RPZ action using *unbound-control stats* or *unbound-control stats_noreset*. This requires the *extended-statistics:* to be enabled.

Unbound's RPZ implementation works together with the tags functionality. This makes it possible to enable (some of) the policy zones only for a subset of users. To do this, the tags need to be defined using *define-tag:*, the correct tags need to be matched either with the client IP prefix using *access-control-tag:* or the clients on a listening interface using *interface-tag:*, and the tags need to be specified for the policy zones for which they apply.

```
server:
    module-config: "respip validator iterator"
    interface: eth0
    define-tag: "malware social"

    # Per client IP ...
    access-control-tag: 127.0.0.10/32 "social"
    access-control-tag: 127.0.0.20/32 "social malware"
    access-control-tag: 127.0.0.30/32 "malware"
    # ... and/or per listening interface
    interface-tag: eth0 "social"

rpz:
    name: malware.rpz.example.com
    zonefile: malware.rpz.example.com
    tags: "malware"

rpz:
    name: social.rpz.example.com
    zonefile: social.rpz.example.com
    tags: "social"
```

Queries from 127.0.0.1 will not be filtered. For queries coming from 127.0.0.10 or the eth0 interface, only the policies from the social.rpz.example.com zone will be used. For queries coming from 127.0.0.30 only the policies from the malware.rpz.example.com zone will be used. Queries coming from 127.0.0.20 will be subjected to the policies from both zones.

See also:

Response Policy Zone Options, *module-config*, *define-tag*, *access-control-tag*, and *extended-statistics* in the *unbound.conf(5)* manpage.

CHAPTER

THIRTEEN

UNBOUND LIBRARY TUTORIAL

This is the tutorial for the *unbound library*. Unbound can run as a server, as a daemon in the background, answering DNS queries from the network. Alternatively, it can link to an application as a library `-lunbound`, and answer DNS queries for the application. This tutorial explains how to use the library API.

13.1 Resolve a Name

First, obtain ldns and *Unbound*, compile and install them. To compile a program with its library use this command, assuming unbound was installed in `/usr/local`:

```
gcc -o program program.c -I/usr/local/include -L/usr/local/lib -lunbound
```

First a basic example that shows how to create a context and resolve a host address.

```
#include <stdio.h>      /* for printf */
#include <arpa/inet.h>  /* for inet_ntoa */
#include <unbound.h>    /* unbound API */

int main(void)
{
        struct ub_ctx* ctx;
        struct ub_result* result;
        int retval;

        /* create context */
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }

        /* query for webserver */
        retval = ub_resolve(ctx, "www.nlnetlabs.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return 1;
        }
```

(continues on next page)

(continued from previous page)

```
        /* show first result */
        if(result->havedata)
                printf("The address is %s\n",
                        inet_ntoa(*(struct in_addr*)result->data[0]));

        ub_resolve_free(result);
        ub_ctx_delete(ctx);
        return 0;
}
```

Invocation of this program yields the following:

```
$ example_1
The address is 213.154.224.1
```

The code starts by including system header files and unbound.h. Then, the main routine creates the context using the `ub_ctx_create()` function. If this returns NULL, the program prints an error and exits.

Then, the domain name `www.nlnetlabs.nl` is resolved using the function `ub_resolve()`. The `ub_resolve` invocation takes as arguments the context that was just created, the domain name string and the type and class to lookup. Results are returned in the `ub_result` structure, unless an error occurs. If an error occurs, retval contains an error code and `ub_strerror` converts the error code into a readable string, that is printed and the program exits.

If the resolve succeeds, then the results are printed. In this example, the results are not examined in detail, but only if there is data, the first element of data is printed. The `result->havedata` boolean indicates whether the resolver found data, and `result->data[0]` is a pointer to the first element of data. The standard C library routine `inet_ntoa` is used to print out the IPv4 address.

Note that this example program neglects to examine `result->len[0]` for simplicity. For security, such untrusted data from the internet should be checked. That value should have been 4 (bytes), the length of IPv4 addresses.

At the end of the main routine, the results are freed with `ub_resolve_free(result)` and the context is deleted with `ub_ctx_delete`. If you perform multiple lookups, it is good to keep the context around, it performs caching and that will speed up your responses.

13.2 Setup the Context

In the second example we set additional useful options on the context, to enhance performance and utility. It is a modification of the example program from the *Resolve a Name* section.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unbound.h>

int main(void)
{
        struct ub_ctx* ctx;
        struct ub_result* result;
        int retval;

        /* create context */
```

(continues on next page)

(continued from previous page)

```
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }
        /* read /etc/resolv.conf for DNS proxy settings (from DHCP) */
        if( (retval=ub_ctx_resolvconf(ctx, "/etc/resolv.conf")) != 0) {
                printf("error reading resolv.conf: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }
        /* read /etc/hosts for locally supplied host addresses */
        if( (retval=ub_ctx_hosts(ctx, "/etc/hosts")) != 0) {
                printf("error reading hosts: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }

        /* query for webserver */
        retval = ub_resolve(ctx, "www.nlnetlabs.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return 1;
        }

        /* show first result */
        if(result->havedata)
                printf("The address is %s\n",
                        inet_ntoa(*(struct in_addr*)result->data[0]));

        ub_resolve_free(result);
        ub_ctx_delete(ctx);
        return 0;
}
```

Invocation of this program yields the following:

```
$ example_2
The address is 213.154.224.1
```

As said, the code is a modification of the *first example*. The context is set up, a single name is looked up, and the results and context are freed. The difference is that local settings are applied.

The local DNS server settings (acquired from DHCP perhaps) are read from `/etc/resolv.conf` with `ub_ctx_resolvconf`. Without reading this, Unbound will use built-in root hints, this is a lot slower than using the DNS servers from `resolv.conf`. It makes a large difference, for me `time example1` takes about 0.25 seconds, and `time example2` takes about 0.05 seconds.

The difference is caused because the DNS proxy in `resolv.conf` has a cache of often used data, and thus can resolve queries much faster. If you perform many queries (and keep the unbound context around between calls to resolve) the time difference will grow smaller over time, since a cache of data is kept inside the context as well.

When you use `ub_ctx_resolvconf` libunbound becomes a stub resolver, not going to the internet itself, but relying on the servers listed. Without the call, by default, libunbound contacts the servers on the internet itself. A reason to not use the servers from resolv.conf is because you do not trust them, or because they lack support for DNSSEC, and you want to use DNSSEC validation.

Note: Some people have complained about DNSSEC validation changing between secure and bogus, randomly. Often these are because they read a `resolv.conf` that contains nameservers where some support DNSSEC and some do not. If unbound detects that signatures are stripped from the answer, it returns bogus.

The function `ub_ctx_set_fwd(ctx, "192.168.0.1")` (not shown in the example program) can be used to set an explicit IPv4 or IPv6 address for the DNS server to use. You can use this function to set DNS caching proxy server addresses that are not listed in `/etc/resolv.conf`.

If you wish to provide your own root-hints file, to override the built-in values, you can use the power-user interface `ub_ctx_set_option(ctx, "root-hints:", "my-hints.root")`, and the file `my-hints.root` is read in before the first name resolution.

The function `ub_ctx_hosts` is used to read `/etc/hosts`. This allows unbound to (very quickly) return addresses for hosts that are configured in `/etc/hosts`. If you do not trust the `/etc/hosts` file, you can avoid loading it. The addresses listed in the hosts file lack DNSSEC signatures, which may affect their validation status later on. The hosts file is a very useful configuration file to load, as it allows users to list addresses that are often used, or addresses for hosts on their local network.

If you do not want your program to fail if `/etc/resolv.conf` or `/etc/hosts` do not exist at all, you can check if `errno == ENOENT` when the reading functions fail, and act accordingly.

13.3 Examine the Results

In the third example, the results returned are examined in detail. In addition, the program is modified to accept an argument, the name to look up. It is a modification of the example program from the *Setup the Context* section.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unbound.h>

/* examine the result structure in detail */
void examine_result(char* query, struct ub_result* result)
{
        int i;
        int num;

        printf("The query is for: %s\n", query);
        printf("The result has:\n");
        printf("qname: %s\n", result->qname);
        printf("qtype: %d\n", result->qtype);
        printf("qclass: %d\n", result->qclass);
        if(result->canonname)
                printf("canonical name: %s\n",
                        result->canonname);
        else    printf("canonical name: <none>\n");
```

(continues on next page)

(continued from previous page)

```
        if(result->havedata)
                printf("has data\n");
        else    printf("has no data\n");

        if(result->nxdomain)
                printf("nxdomain (name does not exist)\n");
        else    printf("not an nxdomain (name exists)\n");

        if(result->secure)
                printf("validated to be secure\n");
        else    printf("not validated as secure\n");

        if(result->bogus)
                printf("a security failure! (bogus)\n");
        else    printf("not a security failure (not bogus)\n");

        printf("DNS rcode: %d\n", result->rcode);
        printf("\n");

        num = 0;
        for(i=0; result->data[i]; i++) {
                printf("result data element %d has length %d\n",
                        i, result->len[i]);
                printf("result data element %d is: %s\n",
                        i, inet_ntoa(*(struct in_addr*)result->data[i]));
                num++;
        }
        printf("result has %d data element(s)\n", num);
}

int main(int argc, char** argv)
{
        struct ub_ctx* ctx;
        struct ub_result* result;
        int retval;

        if(argc != 2) {
                printf("usage: <hostname>\n");
                return 1;
        }

        /* create context */
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }
        /* read /etc/resolv.conf for DNS proxy settings (from DHCP) */
        if( (retval=ub_ctx_resolvconf(ctx, "/etc/resolv.conf")) != 0) {
                printf("error reading resolv.conf: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
```

(continues on next page)

(continued from previous page)

```
                return 1;
        }
        /* read /etc/hosts for locally supplied host addresses */
        if( (retval=ub_ctx_hosts(ctx, "/etc/hosts")) != 0) {
                printf("error reading hosts: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }

        /* query for webserver */
        retval = ub_resolve(ctx, argv[1],
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return 1;
        }
        examine_result(argv[1], result);

        ub_resolve_free(result);
        ub_ctx_delete(ctx);
        return 0;
}
```

Invocation of this program yields the following:

```
$ example_3 www.nlnetlabs.nl
The query is for: www.nlnetlabs.nl
The result has:
qname: www.nlnetlabs.nl
qtype: 1
qclass: 1
canonical name: <none>
has data
not an nxdomain (name exists)
not validated as secure
not a security failure (not bogus)
DNS rcode: 0

result data element 0 has length 4
result data element 0 is: 213.154.224.1
result has 1 data element(s)
```

This example add the option to specify the name too lookup from the commandline, and this name is found in `argv[1]`. The name is looked up and `examine_result` is called to printout a detailed account of the results.

The `qname`, `qtype` and `qclass` fields show the question that was asked to `ub_resolve`.

The canonical name may be set if you query for an alias, in that case the alternate name for the host is set here.

The boolean value `hasdata` is true when at least one data element is available.

The boolean value `nxdomain` is true, when no data is available because the name queried for does not exist.

The boolean value `secure` is true when public key signatures on the answer are are valid. It is also possible for

responses without data to be secure.

The boolean value `bogus` is true when security checks failed. The authenticity of the content, and the absence or presence of it, failed security checks. This happens when, for example, you use the wrong public keys for validation, or if the data was altered in transit.

If both `secure` and `bogus` are false this indicates there was no security information for that domain name.

The `rcode` value indicates the exact DNS error code. If there is no data, it may explain why (the servers encountered errors). If there is no data and the name does not exist (so `nxdomain` is true), the `rcode` value is 3 (NXDOMAIN). If there is no data, and the name does exist (it does not have this type of data) the `rcode` is 0 (NOERROR). Other error codes indicate some sort of failure, mostly a failure at the DNS server.

The example prints all the data elements and their length.

Here are some other results that you can get. The first is an alias, with several addresses, and the second is a nonexistent name:

```
$ example_3 www.google.nl
The query is for: www.google.nl
The result has:
qname: www.google.nl
qtype: 1
qclass: 1
canonical name: www.l.google.com.
has data
not an nxdomain (name exists)
not validated as secure
not a security failure (not bogus)
DNS rcode: 0

result data element 0 has length 4
result data element 0 is: 64.233.183.99
result data element 1 has length 4
result data element 1 is: 64.233.183.104
result data element 2 has length 4
result data element 2 is: 64.233.183.147
result has 3 data element(s)

$ example_3 bla.bla.nl
The query is for: bla.bla.nl
The result has:
qname: bla.bla.nl
qtype: 1
qclass: 1
canonical name: <none>
has no data
nxdomain (name does not exist)
not validated as secure
not a security failure (not bogus)
DNS rcode: 3

result has 0 data element(s)
```

13.4 Asynchronous Lookup

This example performs the name lookup in the background. The original program keeps running, while the name is resolved. It is a modification of the example program from the *Resolve a Name* section.

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <unbound.h>

/* This is called when resolution is completed */
void mycallback(void* mydata, int err, struct ub_result* result)
{
        int* done = (int*)mydata;
        *done = 1;
        if(err != 0) {
                printf("resolve error: %s\n", ub_strerror(err));
                return;
        }
        /* show first result */
        if(result->havedata)
                printf("The address of %s is %s\n", result->qname,
                        inet_ntoa(*(struct in_addr*)result->data[0]));

        ub_resolve_free(result);
}

int main(void)
{
        struct ub_ctx* ctx;
        volatile int done = 0;
        int retval;
        int i = 0;

        /* create context */
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }

        /* asynchronous query for webserver */
        retval = ub_resolve_async(ctx, "www.nlnetlabs.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */,
                (void*)&done, mycallback, NULL);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return 1;
        }

        /* we keep running, lets do something while waiting */
        while(!done) {
```

(continues on next page)

(continued from previous page)

```
                usleep(100000); /* wait 1/10 of a second */
                printf("time passed (%d) ..\n", i++);
                retval = ub_process(ctx);
                if(retval != 0) {
                        printf("resolve error: %s\n", ub_strerror(retval));
                        return 1;
                }
        }
        printf("done\n");

        ub_ctx_delete(ctx);
        return 0;
}
```

Invocation of this program yields the following:

```
$ example_4
time passed (0) ..
time passed (1) ..
time passed (2) ..
The address of www.nlnetlabs.nl is 213.154.224.1
done
```

If resolution takes longer or shorter, the output can vary.

The context is created. Then an asynchronous resolve is performed. This performs the name resolution work in the background, allowing your application to continue to perform tasks (like showing a GUI to the user).

The function to start a background, asynchronous, resolve is `ub_resolve_async`. It takes the usual context, name, type and class as arguments. Additionally it takes a user argument, callback function and an id as arguments. In the example, the user argument is a reference to the variable done. It can be any pointer you like, or NULL if you don't care. The callback function is a pointer to a function, like `mycallback` in the example, that is invoked when the lookup is done.

The optional id argument is omitted in the example by passing NULL. If you pass an int*, an identifier is returned to you, that allows subsequent cancellation of the outstanding resolve request. The function `ub_cancel` can be used while the asynchronous lookup has not completed yet to cancel it (not shown in the example).

After requesting the lookup the main function continues with a while loop, that prints time increments. Every time increment `ub_process` is called. This function processes pending lookup results and an application has to call `ub_process` somewhere to be able to receive results from asynchronous queries. The function `ub_process` does not block. The callback function is called from within `ub_process`.

The callback is called after some time, in the example it is called `mycallback`. This function receives as its first argument the same value you passed as user argument to `ub_resolve_async`. It also receives the error code and a result structure. If the error code is not 0 (an error happened), the result is NULL. The result structure contains the lookup information.

The example callback uses its first argument to set done to true, to signal the main function that lookup has completed. It then checks if an error happened, and prints it if so. If there was no error it prints the first data element of the result. (It doesn't check the result very closely, this is only an example).

When the main function sees that after a call to `ub_process` the variable done is true, it exits the waiting loop, and deletes the context. The delete of the context also stops the background resolution process and removes the cached data from memory.

You do not have to call `ub_process` all the time. The function `ub_poll` (not shown in example) returns true when new

data is available (without calling any callbacks). The function `ub_fd` (not shown in example) returns a file descriptor that becomes readable when new data is available (for use with `select()` or similar system calls).

The function `ub_wait` (not shown in example) can be used to wait for the asynchronous lookups to complete. For example, when the main program continues to set up a user GUI after starting the lookup, then if it runs out of work before the result arrives, it can use `ub_wait` to block until data arrives.

13.5 Lookup from Threads

This example shows how to use libunbound from a threaded program. It is a modification of the example program from the *Setup the Context* section. It creates two threads and resolves from both threads.

This example uses `pthreads`, and assumes that libunbound was compiled with threading enabled (which is the default, if `pthreads` can be found). To compile the example pass the compiler the option `-lpthread`.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unbound.h>

#include <pthread.h>

/* The main function of the first thread */
void* thread_one(void* threadarg)
{
        struct ub_ctx* ctx = (struct ub_ctx*)threadarg;
        struct ub_result* result;
        int retval;
        /* query for webserver */
        retval = ub_resolve(ctx, "www.nlnetlabs.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return NULL;
        }

        /* show first result */
        if(result->havedata)
                printf("Thread1: address of %s is %s\n", result->qname,
                        inet_ntoa(*(struct in_addr*)result->data[0]));

        /* exit thread */
        ub_resolve_free(result);
        return NULL;
}

/* The main function of the second thread */
void* thread_two(void* threadarg)
{
        struct ub_ctx* ctx = (struct ub_ctx*)threadarg;
        struct ub_result* result;
```

(continues on next page)

(continued from previous page)

```
        int retval;
        /* query for webserver */
        retval = ub_resolve(ctx, "www.google.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return NULL;
        }

        /* show first result */
        if(result->havedata)
                printf("Thread2: address of %s is %s\n", result->qname,
                        inet_ntoa(*(struct in_addr*)result->data[0]));

        /* exit thread */
        ub_resolve_free(result);
        return NULL;
}

int main(void)
{
        struct ub_ctx* ctx;
        int retval;
        pthread_t t1, t2;

        /* create context */
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }
        /* read /etc/resolv.conf for DNS proxy settings (from DHCP) */
        if( (retval=ub_ctx_resolvconf(ctx, "/etc/resolv.conf")) != 0) {
                printf("error reading resolv.conf: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }
        /* read /etc/hosts for locally supplied host addresses */
        if( (retval=ub_ctx_hosts(ctx, "/etc/hosts")) != 0) {
                printf("error reading hosts: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }

        /* start two threads, uses pthreads */
        pthread_create(&t1, NULL, thread_one, ctx);
        pthread_create(&t2, NULL, thread_two, ctx);
        /* wait for both threads to complete */
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
```

(continues on next page)

(continued from previous page)

```
        ub_ctx_delete(ctx);
        return 0;
}
```

Invocation of this program yields the following:

```
$ example_5
Thread1: address of www.nlnetlabs.nl is 213.154.224.1
Thread2: address of www.google.nl is 64.233.183.147
```

Sometimes, the result from thread 2 is printed first.

The example starts at the `main` program function. The unbound context is created and `resolv.conf` and `/etc/hosts` are read in. Then, two threads are started using `pthread_create`. The main program continues with waiting for those two threads to finish.

The first thread, `thread_one`, starts by obtaining a pointer to the unbound context from the thread argument. Then, www.nlnetlabs.nl is resolved, using the regular `ub_resolve`. The result is printed, and freed and the thread exits with `return NULL`.

The second thread, `thread_two`, does the same as the first thread, but looks up www.google.nl instead.

Using threads is easy when the context is created with `ub_ctx_create`. In this example, when both threads start resolving, they act as a 2-threaded resolver, and share results, validation outcomes and data. When one of the threads finishes its lookup, the other thread continues as a 1-threaded resolver. When the resolver is created with `ub_ctx_create_event` or `ub_ctx_create_ub_event`, with an event base, then it can only be accessed from one thread, usually the one that is running that event loop.

This example uses blocking resolution for both threads. You can use asynchronous resolution in threaded programs too. The function `ub_resolve_async` is used to perform a background lookup. The calling thread continues executing while the background lookup is in progress.

The application can decide if it wants the background lookup to be performed from a (forked) process or from a (newly created) thread, by setting `ub_ctx_async`. The default is to fork. The asynchronous resolution process or thread is deleted when `ub_ctx_delete` is called.

Callbacks from asynchronous lookups are performed when `ub_process` is called, just like in a single-threaded program. The thread from which the callbacks are called is the thread from which `ub_process` has been called. It is the responsibility of the application to signal other threads that lookup results are available.

It is possible to have a thread wait for the file descriptor from `ub_ctx_fd` (a pipe) to become readable, and process any pending lookup results with `ub_process`.

13.6 DNSSEC Validate

This example program performs DNSSEC validation of a lookup. It is a modification of the example program from the *Setup the Context* section.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <unbound.h>

int main(void)
```

(continues on next page)

(continued from previous page)

```
{
        struct ub_ctx* ctx;
        struct ub_result* result;
        int retval;

        /* create context */
        ctx = ub_ctx_create();
        if(!ctx) {
                printf("error: could not create unbound context\n");
                return 1;
        }
        /* read /etc/resolv.conf for DNS proxy settings (from DHCP) */
        if( (retval=ub_ctx_resolvconf(ctx, "/etc/resolv.conf")) != 0) {
                printf("error reading resolv.conf: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }
        /* read /etc/hosts for locally supplied host addresses */
        if( (retval=ub_ctx_hosts(ctx, "/etc/hosts")) != 0) {
                printf("error reading hosts: %s. errno says: %s\n",
                        ub_strerror(retval), strerror(errno));
                return 1;
        }

        /* read public keys for DNSSEC verification */
        if( (retval=ub_ctx_add_ta_file(ctx, "keys")) != 0) {
                printf("error adding keys: %s\n", ub_strerror(retval));
                return 1;
        }

        /* query for webserver */
        retval = ub_resolve(ctx, "www.nlnetlabs.nl",
                1 /* TYPE A (IPv4 address) */,
                1 /* CLASS IN (internet) */, &result);
        if(retval != 0) {
                printf("resolve error: %s\n", ub_strerror(retval));
                return 1;
        }

        /* show first result */
        if(result->havedata)
                printf("The address is %s\n",
                        inet_ntoa(*(struct in_addr*)result->data[0]));
        /* show security status */
        if(result->secure)
                printf("Result is secure\n");
        else if(result->bogus)
                printf("Result is bogus: %s\n", result->why_bogus);
        else    printf("Result is insecure\n");

        ub_resolve_free(result);
        ub_ctx_delete(ctx);
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

Invocation of this program yields the following:

```
First testrun
    $ touch keys
    $ example_6
    The address is 213.154.224.1
    Result is insecure
```

The first testrun uses an empty keyfile, and since there is no security configured for nlnetlabs.nl, the result is insecure. For a secure result, DNSSEC security must be configured on both the server and the client, and in this example run, the nlnetlabs.nl server has security configured, but the key file is empty on the client.

```
Second testrun
    $ dig nlnetlabs.nl DNSKEY > keys
    $ example_6
    The address is 213.154.224.1
    Result is secure
```

The second testrun obtains the current DNSKEY information for `nlnetlabs.nl` using `dig` (from the `named` utilities).

Note: This is not a secure method to obtain keys, check keys carefully before you trust them and enter them into your application (for example RIPE distributes key files with added PGP signatures).

But it is very easy, and useful for this tutorial. The lookup result is secure, because it is signed with the correct keys.

```
Third testrun
    $ echo 'nlnetlabs.nl. 3528 IN DNSKEY ( 256 3 5
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
        AAAAAA== )' > keys
    $ example_6
    The address is 213.154.224.1
    Result is bogus: validation failure <www.nlnetlabs.nl. A IN>:
    signatures from unknown keys from 213.154.224.254 for trust anchor
    nlnetlabs.nl. while building chain of trust
```

The third example puts a key into the keyfile that is not going to match any signatures. The `echo` command is wrapped onto multiple lines on this page for presentation, put the text onto one line. Because the key and the signatures on the data do not match, verification fails and the result is bogus.

The program starts like in the *Setup the Context* section of the tutorial, creates the unbound context and reads in `/etc/resolv.conf` and `/etc/hosts`. Then it adds the contents of the keys file from the current directory as trusted keys. It continues to resolve www.nlnetlabs.nl and prints the result. It also prints the security status of the result.

The function `ub_ctx_add_ta_file` adds trusted keys. The keys file contains text in the zone file format (output from `dig` or `drill` tools, or a copy and paste from the DNS zone file). It can contain DNSKEY and DS entries, for any number of domain names. If any of the keys matches the signatures on lookup results, the `result->secure` is set true.

The function ub_ctx_add_ta (not shown in example) can be used to add a trusted key from a string. A single DNSKEY or DS key entry, on a single line, is expected. Multiple keys can be given with multiple calls to ub_ctx_add_ta. For example:

```
if( (retval=ub_ctx_add_ta(ctx, "jelte.nlnetlabs.nl. DS 31560 "
    "5 1 1CFED84787E6E19CCF9372C1187325972FE546CD")) != 0)
{ /* print error */ }
```

It is also possible to read in named (BIND-style) key config files. These files contain trusted-key{} clauses. The function ub_ctx_trustedkeys (not shown in example) adds the keys from a bind-style config file. ub_ctx_set_option(ctx, "auto-trust-anchor-file:", "keys") (not shown in example) can be used to use auto-updated keys (with RFC5011), the file is read from and written to when the keys change. The probes have to be frequent enough to not lose track, about every 15 days.

It is worth noting that with DNSSEC it is possible to verify nonexistence of data. So, if the example above is modified to query for foobar.nlnetlabs.nl and with correct keys in the keys file, the output is no data, but the result is secure.

DNSSEC has complicated verification procedures. The result is distilled into two booleans, secure and bogus. Either the result is secure, the result is bogus, or the result is neither of the two, called insecure. Insecure happens when no DNSSEC security is configured for the domain name (or you simply forgot to add the trusted key). Secure means that one of the trusted keys verifies the signatures on the data. Bogus (security failed) can have many reasons, DNSSEC protects against alteration of the data in transit, signatures can expire, the trusted keys can be rolled over to fresh trusted keys, and many others. The functions ub_ctx_debugout (sets a stream to log to) and ub_ctx_debuglevel (try level 2) can give more information about a security failure. The why_bogus string as printed in the example above attempts to give a detailed reason for the failure. An e-commerce application can simply look at result->secure for its shopping server, and only continue if the result is secure.

CHAPTER

FOURTEEN

UNBOUND FOR PYTHON

14.1 Pyunbound

Unbound supports bindings for Python which we call 'pyunbound'. This functionality can be enabled in the **configure** step of the installation using the following option:

```
./configure --with-pyunbound
```

Documentation for pyunbound will then also be included when building Unbound's documentation with:

```
make doc
```

This command will generate the relevant pyunbound documentation in `doc/html/pyunbound`, which can be browsed in a web browser by opening the `index.html` file in that directory.

The pyunbound documentation can also be solely generated without the need to configure/compile Unbound by invoking **sphinx-build** directly with:

```
sphinx-build -b html libunbound/python/doc doc/html/pyunbound/
```

14.2 Pythonmod

Unbound contains a module that executes python code called 'pythonmod'. The supplied Python code has to follow module operation semantics. This module is enabled in the **configure** step of the installation using the following option:

```
./configure --with-pythonmodule
```

Documentation for pythonmod will then also be included when building Unbound's documentation with:

```
make doc
```

This command will generate the relevant pythonmod documentation in `doc/html/pythonmod`, which can be browsed in a web browser by opening the `index.html` file in that directory.

The pythonmod documentation can also be solely generated without the need to configure/compile Unbound by invoking **sphinx-build** directly with:

```
sphinx-build -b html pythonmod/doc/ doc/html/pythonmod/
```

CHAPTER

FIFTEEN

SOURCE CODE DOCS

The automatically generated documentation of the Unbound source code is available in on the NLnet Labs website.

CHAPTER

SIXTEEN

UNBOUND(8)

16.1 Synopsis

unbound [-hdpv] [-c <cfgfile>]

16.2 Description

unbound is a caching DNS resolver.

It uses a built in list of authoritative nameservers for the root zone (.), the so called root hints. On receiving a DNS query it will ask the root nameservers for an answer and will in almost all cases receive a delegation to a top level domain (TLD) authoritative nameserver. It will then ask that nameserver for an answer. It will recursively continue until an answer is found or no answer is available (NXDOMAIN). For performance and efficiency reasons that answer is cached for a certain time (the answer's time-to-live or TTL). A second query for the same name will then be answered from the cache. Unbound can also do DNSSEC validation.

To use a locally running Unbound for resolving put:

```
nameserver 127.0.0.1
```

into *resolv.conf(5)*.

If authoritative DNS is needed as well using nsd(8), careful setup is required because authoritative nameservers and resolvers are using the same port number (53).

The available options are:

-h

Show the version number and commandline option help, and exit.

-c <cfgfile>

Set the config file with settings for unbound to read instead of reading the file at the default location, /usr/local/etc/unbound/unbound.conf. The syntax is described in *unbound.conf(5)*.

-d

Debug flag: do not fork into the background, but stay attached to the console. This flag will also delay writing to the log file until the thread-spawn time, so that most config and setup errors appear on stderr. If given twice or more, logging does not switch to the log file or to syslog, but the log messages are printed to stderr all the time.

-p

Don't use a pidfile. This argument should only be used by supervision systems which can ensure that only one instance of Unbound will run concurrently.

-v

Increase verbosity. If given multiple times, more information is logged. This is in addition to the verbosity (if any) from the config file.

-V

Show the version number and build options, and exit.

16.3 See Also

unbound.conf(5), *unbound-checkconf(8)*, nsd(8).

CHAPTER

SEVENTEEN

UNBOUND-CHECKCONF(8)

17.1 Synopsis

unbound-checkconf [-hf] [-o option] [cfgfile]

17.2 Description

unbound-checkconf checks the configuration file for the *unbound(8)* DNS resolver for syntax and other errors. The config file syntax is described in *unbound.conf(5)*.

The available options are:

-h

Show the version and commandline option help.

-f

Print full pathname, with chroot applied to it. Use with the *-o* option.

-o <option>

If given, after checking the config file the value of this option is printed to stdout. For "" (disabled) options an empty line is printed.

cfgfile

The config file to read with settings for Unbound. It is checked. If omitted, the config file at the default location is checked.

17.3 Exit Code

The unbound-checkconf program exits with status code 1 on error, 0 for a correct config file.

17.4 Files

/usr/local/etc/unbound/unbound.conf
Unbound configuration file.

17.5 See Also

unbound.conf(5), *unbound(8)*.

CHAPTER

EIGHTEEN

UNBOUND.CONF(5)

18.1 Synopsis

unbound.conf

18.2 Description

unbound.conf is used to configure *unbound(8)*. The file format has attributes and values. Some attributes have attributes inside them. The notation is: `attribute: value`.

Comments start with # and last to the end of line. Empty lines are ignored as is whitespace at the beginning of a line.

The utility *unbound-checkconf(8)* can be used to check `unbound.conf` prior to usage.

18.3 Example

An example config file is shown below. Copy this to `/etc/unbound/unbound.conf` and start the server with:

```
$ unbound -c /etc/unbound/unbound.conf
```

Most settings are the defaults. Stop the server with:

```
$ kill `cat /etc/unbound/unbound.pid`
```

Below is a minimal config file. The source distribution contains an extensive `example.conf` file with all the options.

```
# unbound.conf(5) config file for unbound(8).
server:
directory: "/etc/unbound"
username: unbound
# make sure unbound can access entropy from inside the chroot.
# e.g. on linux the use these commands (on BSD, devfs(8) is used):
#      mount --bind -n /dev/urandom /etc/unbound/dev/urandom
# and  mount --bind -n /dev/log /etc/unbound/dev/log
chroot: "/etc/unbound"
# logfile: "/etc/unbound/unbound.log"  #uncomment to use logfile.
pidfile: "/etc/unbound/unbound.pid"
# verbosity: 1      # uncomment and increase to get more logging.
# listen on all interfaces, answer queries from the local subnet.
```

(continues on next page)

(continued from previous page)

```
interface: 0.0.0.0
interface: ::0
access-control: 10.0.0.0/8 allow
access-control: 2001:DB8::/64 allow
```

18.4 File Format

There must be whitespace between keywords. Attribute keywords end with a colon `':'`. An attribute is followed by a value, or its containing attributes in which case it is referred to as a clause. Clauses can be repeated throughout the file (or included files) to group attributes under the same clause.

Files can be included using the **include:** directive. It can appear anywhere, it accepts a single file name as argument. Processing continues as if the text from the included file was copied into the config file at that point. If also using *chroot:*, using full path names for the included files works, relative pathnames for the included names work if the directory where the daemon is started equals its chroot/working directory or is specified before the include statement with *directory: dir*. Wildcards can be used to include multiple files, see *glob(7)*.

For a more structural include option, the **include-toplevel:** directive can be used. This closes whatever clause is currently active (if any) and forces the use of clauses in the included files and right after this directive.

18.4.1 Server Options

These options are part of the **server:** clause.

verbosity: <*number*>
The verbosity level.

Level 0
No verbosity, only errors.

Level 1
Gives operational information.

Level 2
Gives detailed operational information including short information per query.

Level 3
Gives query level information, output per query.

Level 4
Gives algorithm level information.

Level 5
Logs client identification for cache misses.

The verbosity can also be increased from the command line and during run time via remote control. See *unbound(8)* and *unbound-control(8)* respectively.

Default: 1

statistics-interval: <*seconds*>
The number of seconds between printing statistics to the log for every thread. Disable with value `0` or `""`. The histogram statistics are only printed if replies were sent during the statistics interval, requestlist statistics are printed for every interval (but can be 0). This is because the median calculation requires data to be present.

Default: 0 (disabled)

statistics-cumulative: *<yes or no>*
If enabled, statistics are cumulative since starting Unbound, without clearing the statistics counters after logging the statistics.

Default: no

extended-statistics: *<yes or no>*
If enabled, extended statistics are printed from *unbound-control(8)*. The counters are listed in *unbound-control(8)*. Keeping track of more statistics takes time.

Default: no

statistics-inhibit-zero: *<yes or no>*
If enabled, selected extended statistics with a value of 0 are inhibited from printing with *unbound-control(8)*. These are query types, query classes, query opcodes, answer rcodes (except NOERROR, FORMERR, SERVFAIL, NXDOMAIN, NOTIMPL, REFUSED) and PRZ actions.

Default: yes

num-threads: *<number>*
The number of threads to create to serve clients. Use 1 for no threading.

Default: 1

port: *<port number>*
The port number on which the server responds to queries.

Default: 53

interface: *<IP address or interface name[@port]>*
Interface to use to connect to the network. This interface is listened to for queries from clients, and answers to clients are given from it. Can be given multiple times to work on several interfaces. If none are given the default is to listen on localhost.

If an interface name is used instead of an IP address, the list of IP addresses on that interface are used. The interfaces are not changed on a reload (`kill -HUP`) but only on restart.

A port number can be specified with @port (without spaces between interface and port number), if not specified the default port (from *port:*) is used.

ip-address: *<IP address or interface name[@port]>*
Same as *interface:* (for ease of compatibility with nsd.conf(5)).

interface-automatic: *<yes or no>*
Listen on all addresses on all (current and future) interfaces, detect the source interface on UDP queries and copy them to replies. This is a lot like *ip-transparent:*, but this option services all interfaces whilst with *ip-transparent:* you can select which (future) interfaces Unbound provides service on. This feature is experimental, and needs support in your OS for particular socket options.

Default: no

interface-automatic-ports: *"<string>"*
List the port numbers that *interface-automatic:* listens on. If empty, the default port is listened on. The port numbers are separated by spaces in the string.

This can be used to have interface automatic to deal with the interface, and listen on the normal port number, by including it in the list, and also HTTPS or DNS-over-TLS port numbers by putting them in the list as well.

Default: ""

outgoing-interface: *<IPv4/IPv6 address or IPv6 netblock>*
Interface to use to connect to the network. This interface is used to send queries to authoritative servers and receive their replies. Can be given multiple times to work on several interfaces. If none are given the default (all)

is used. You can specify the same interfaces in *interface:* and *outgoing-interface:* lines, the interfaces are then used for both purposes. Outgoing queries are sent via a random outgoing interface to counter spoofing.

If an IPv6 netblock is specified instead of an individual IPv6 address, outgoing UDP queries will use a randomised source address taken from the netblock to counter spoofing. Requires the IPv6 netblock to be routed to the host running Unbound, and requires OS support for unprivileged non-local binds (currently only supported on Linux). Several netblocks may be specified with multiple *outgoing-interface:* options, but do not specify both an individual IPv6 address and an IPv6 netblock, or the randomisation will be compromised. Consider combining with *prefer-ip6:* yes to increase the likelihood of IPv6 nameservers being selected for queries. On Linux you need these two commands to be able to use the freebind socket option to receive traffic for the ip6 netblock:

```
ip -6 addr add mynetblock/64 dev lo && \
ip -6 route add local mynetblock/64 dev lo
```

outgoing-range: <*number*>
Number of ports to open. This number of file descriptors can be opened per thread. Must be at least 1. Default depends on compile options. Larger numbers need extra resources from the operating system. For performance a very large value is best, use libevent to make this possible.

Default: 4096 (libevent) / 960 (minievent) / 48 (windows)

outgoing-port-permit: <*port number or range*>
Permit Unbound to open this port or range of ports for use to send queries. A larger number of permitted outgoing ports increases resilience against spoofing attempts. Make sure these ports are not needed by other daemons. By default only ports above 1024 that have not been assigned by IANA are used. Give a port number or a range of the form "low-high", without spaces.

The *outgoing-port-permit:* and *outgoing-port-avoid:* statements are processed in the line order of the config file, adding the permitted ports and subtracting the avoided ports from the set of allowed ports. The processing starts with the non IANA allocated ports above 1024 in the set of allowed ports.

outgoing-port-avoid: <*port number or range*>
Do not permit Unbound to open this port or range of ports for use to send queries. Use this to make sure Unbound does not grab a port that another daemon needs. The port is avoided on all outgoing interfaces, both IPv4 and IPv6. By default only ports above 1024 that have not been assigned by IANA are used. Give a port number or a range of the form "low-high", without spaces.

outgoing-num-tcp: <*number*>
Number of outgoing TCP buffers to allocate per thread. If set to 0, or if *do-tcp: no*, no TCP queries to authoritative servers are done. For larger installations increasing this value is a good idea.

Default: 10

incoming-num-tcp: <*number*>
Number of incoming TCP buffers to allocate per thread. If set to 0, or if *do-tcp: no*, no TCP queries from clients are accepted. For larger installations increasing this value is a good idea.

Default: 10

edns-buffer-size: <*number*>
Number of bytes size to advertise as the EDNS reassembly buffer size. This is the value put into datagrams over UDP towards peers. The actual buffer size is determined by *msg-buffer-size:* (both for TCP and UDP). Do not set higher than that value. Setting to 512 bypasses even the most stringent path MTU problems, but is seen as extreme, since the amount of TCP fallback generated is excessive (probably also for this resolver, consider tuning *outgoing-num-tcp:*).

Default: 1232 (DNS Flag Day 2020 recommendation)

max-udp-size: <*number*>
Maximum UDP response size (not applied to TCP response). 65536 disables the UDP response size maximum,

and uses the choice from the client, always. Suggested values are 512 to 4096.

Default: 1232 (same as *edns-buffer-size:*)

stream-wait-size: <*number*>
Number of bytes size maximum to use for waiting stream buffers. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte). As TCP and TLS streams queue up multiple results, the amount of memory used for these buffers does not exceed this number, otherwise the responses are dropped. This manages the total memory usage of the server (under heavy use), the number of requests that can be queued up per connection is also limited, with further requests waiting in TCP buffers.

Default: 4m

msg-buffer-size: <*number*>
Number of bytes size of the message buffers. Default is 65552 bytes, enough for 64 Kb packets, the maximum DNS message size. No message larger than this can be sent or received. Can be reduced to use less memory, but some requests for DNS data, such as for huge resource records, will result in a SERVFAIL reply to the client.

Default: 65552

msg-cache-size: <*number*>
Number of bytes size of the message cache. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 4m

msg-cache-slabs: <*number*>
Number of slabs in the message cache. Slabs reduce lock contention by threads. Must be set to a power of 2. Setting (close) to the number of cpus is a reasonable guess.

Default: 4

num-queries-per-thread: <*number*>
The number of queries that every thread will service simultaneously. If more queries arrive that need servicing, and no queries can be jostled out (see *jostle-timeout:*), then the queries are dropped. This forces the client to resend after a timeout; allowing the server time to work on the existing queries. Default depends on compile options.

Default: 1024 (libevent) / 512 (minievent) / 24 (windows)

jostle-timeout: <*msec*>
Timeout used when the server is very busy. Set to a value that usually results in one roundtrip to the authority servers.

If too many queries arrive, then 50% of the queries are allowed to run to completion, and the other 50% are replaced with the new incoming query if they have already spent more than their allowed time. This protects against denial of service by slow queries or high query rates.

The effect is that the qps for long-lasting queries is about (numqueriesperthread / 2) / (average time for such long queries) qps. The qps for short queries can be about (numqueriesperthread / 2) / (jostletimeout in whole seconds) qps per thread, about (1024/2)*5 = 2560 qps by default.

Default: 200

delay-close: <*msec*>
Extra delay for timeouted UDP ports before they are closed, in msec. This prevents very delayed answer packets from the upstream (recursive) servers from bouncing against closed ports and setting off all sort of close-port counters, with eg. 1500 msec. When timeouts happen you need extra sockets, it checks the ID and remote IP of packets, and unwanted packets are added to the unwanted packet counter.

Default: 0 (disabled)

udp-connect: *<yes or no>*
Perform *connect(2)* for UDP sockets that mitigates ICMP side channel leakage.

Default: yes

unknown-server-time-limit: *<msec>*
The wait time in msec for waiting for an unknown server to reply. Increase this if you are behind a slow satellite link, to eg. 1128. That would then avoid re-querying every initial query because it times out.

Default: 376

so-rcvbuf: *<number>*
If not 0, then set the SO_RCVBUF socket option to get more buffer space on UDP port 53 incoming queries. So that short spikes on busy servers do not drop packets (see counter in `netstat -su`). Otherwise, the number of bytes to ask for, try "4m" on a busy server.

The OS caps it at a maximum, on linux Unbound needs root permission to bypass the limit, or the admin can use `sysctl net.core.rmem_max`.

On BSD change `kern.ipc.maxsockbuf` in `/etc/sysctl.conf`.

On OpenBSD change header and recompile kernel.

On Solaris `ndd -set /dev/udp udp_max_buf 8388608`.

Default: 0 (use system value)

so-sndbuf: *<number>*
If not 0, then set the SO_SNDBUF socket option to get more buffer space on UDP port 53 outgoing queries. This for very busy servers handles spikes in answer traffic, otherwise

```
send: resource temporarily unavailable
```

can get logged, the buffer overrun is also visible by `netstat -su`. Specify the number of bytes to ask for, try "4m" on a very busy server.

The OS caps it at a maximum, on linux Unbound needs root permission to bypass the limit, or the admin can use `sysctl net.core.wmem_max`.

On BSD, Solaris changes are similar to *so-rcvbuf:*.

Default: 0 (use system value)

so-reuseport: *<yes or no>*
If yes, then open dedicated listening sockets for incoming queries for each thread and try to set the SO_REUSEPORT socket option on each socket. May distribute incoming queries to threads more evenly.

On Linux it is supported in kernels >= 3.9.

On other systems, FreeBSD, OSX it may also work.

You can enable it (on any platform and kernel), it then attempts to open the port and passes the option if it was available at compile time, if that works it is used, if it fails, it continues silently (unless verbosity 3) without the option.

At extreme load it could be better to turn it off to distribute the queries evenly, reported for Linux systems (4.4.x).

Default: yes

ip-transparent: *<yes or no>*
If yes, then use IP_TRANSPARENT socket option on sockets where Unbound is listening for incoming traffic. Allows you to bind to non-local interfaces. For example for non-existent IP addresses that are going to exist later on, with host failover configuration.

This is a lot like *interface-automatic:*, but that one services all interfaces and with this option you can select which (future) interfaces Unbound provides service on.

This option needs Unbound to be started with root permissions on some systems. The option uses IP_BINDANY on FreeBSD systems and SO_BINDANY on OpenBSD systems.

Default: no

ip-freebind: <*yes or no*>
If yes, then use IP_FREEBIND socket option on sockets where Unbound is listening to incoming traffic. Allows you to bind to IP addresses that are nonlocal or do not exist, like when the network interface or IP address is down.

Exists only on Linux, where the similar *ip-transparent:* option is also available.

Default: no

ip-dscp: <*number*>
The value of the Differentiated Services Codepoint (DSCP) in the differentiated services field (DS) of the outgoing IP packet headers. The field replaces the outdated IPv4 Type-Of-Service field and the IPv6 traffic class field.

rrset-cache-size: <*number*>
Number of bytes size of the RRset cache. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 4m

rrset-cache-slabs: <*number*>
Number of slabs in the RRset cache. Slabs reduce lock contention by threads. Must be set to a power of 2.

Default: 4

cache-max-ttl: <*seconds*>
Time to live maximum for RRsets and messages in the cache. When the TTL expires, the cache item has expired. Can be set lower to force the resolver to query for data often, and not trust (very large) TTL values. Downstream clients also see the lower TTL.

Default: 86400 (1 day)

cache-min-ttl: <*seconds*>
Time to live minimum for RRsets and messages in the cache. If the minimum kicks in, the data is cached for longer than the domain owner intended, and thus less queries are made to look up the data. Zero makes sure the data in the cache is as the domain owner intended, higher values, especially more than an hour or so, can lead to trouble as the data in the cache does not match up with the actual data any more.

Default: 0 (disabled)

cache-max-negative-ttl: <*seconds*>
Time to live maximum for negative responses, these have a SOA in the authority section that is limited in time. This applies to NXDOMAIN and NODATA answers.

Default: 3600

infra-host-ttl: <*seconds*>
Time to live for entries in the host cache. The host cache contains roundtrip timing, lameness and EDNS support information.

Default: 900

infra-cache-slabs: <*number*>
Number of slabs in the infrastructure cache. Slabs reduce lock contention by threads. Must be set to a power of 2.

Default: 4

infra-cache-numhosts: *<number>*
Number of hosts for which information is cached.

Default: 10000

infra-cache-min-rtt: *<msec>*
Lower limit for dynamic retransmit timeout calculation in infrastructure cache. Increase this value if using forwarders needing more time to do recursive name resolution.

Default: 50

infra-cache-max-rtt: *<msec>*
Upper limit for dynamic retransmit timeout calculation in infrastructure cache.

Default: 120000 (2 minutes)

infra-keep-probing: *<yes or no>*
If enabled the server keeps probing hosts that are down, in the one probe at a time regime. Hosts that are down, eg. they did not respond during the one probe at a time period, are marked as down and it may take *infra-host-ttl:* time to get probed again.

Default: no

define-tag: *"<list of tags>"*
Define the tags that can be used with *local-zone:* and *access-control:*. Enclose the list between quotes ("") and put spaces between tags.

do-ip4: *<yes or no>*
Enable or disable whether IPv4 queries are answered or issued.

Default: yes

do-ip6: *<yes or no>*
Enable or disable whether IPv6 queries are answered or issued. If disabled, queries are not answered on IPv6, and queries are not sent on IPv6 to the internet nameservers. With this option you can disable the IPv6 transport for sending DNS traffic, it does not impact the contents of the DNS traffic, which may have IPv4 (A) and IPv6 (AAAA) addresses in it.

Default: yes

prefer-ip4: *<yes or no>*
If enabled, prefer IPv4 transport for sending DNS queries to internet nameservers. Useful if the IPv6 netblock the server has, the entire /64 of that is not owned by one operator and the reputation of the netblock /64 is an issue, using IPv4 then uses the IPv4 filters that the upstream servers have.

Default: no

prefer-ip6: *<yes or no>*
If enabled, prefer IPv6 transport for sending DNS queries to internet nameservers.

Default: no

do-udp: *<yes or no>*
Enable or disable whether UDP queries are answered or issued.

Default: yes

do-tcp: *<yes or no>*
Enable or disable whether TCP queries are answered or issued.

Default: yes

tcp-mss: <*number*>
Maximum segment size (MSS) of TCP socket on which the server responds to queries. Value lower than common MSS on Ethernet (1220 for example) will address path MTU problem. Note that not all platform supports socket option to set MSS (TCP_MAXSEG). Default is system default MSS determined by interface MTU and negotiation between server and client.

outgoing-tcp-mss: <*number*>
Maximum segment size (MSS) of TCP socket for outgoing queries (from Unbound to other servers). Value lower than common MSS on Ethernet (1220 for example) will address path MTU problem. Note that not all platform supports socket option to set MSS (TCP_MAXSEG). Default is system default MSS determined by interface MTU and negotiation between Unbound and other servers.

tcp-idle-timeout: <*msec*>
The period Unbound will wait for a query on a TCP connection. If this timeout expires Unbound closes the connection. When the number of free incoming TCP buffers falls below 50% of the total number configured, the option value used is progressively reduced, first to 1% of the configured value, then to 0.2% of the configured value if the number of free buffers falls below 35% of the total number configured, and finally to 0 if the number of free buffers falls below 20% of the total number configured. A minimum timeout of 200 milliseconds is observed regardless of the option value used.

Default: 30000 (30 seconds)

tcp-reuse-timeout: <*msec*>
The period Unbound will keep TCP persistent connections open to authority servers.

Default: 60000 (60 seconds)

max-reuse-tcp-queries: <*number*>
The maximum number of queries that can be sent on a persistent TCP connection.

Default: 200

tcp-auth-query-timeout: <*number*>
Timeout in milliseconds for TCP queries to auth servers.

Default: 3000 (3 seconds)

edns-tcp-keepalive: <*yes or no*>
Enable or disable EDNS TCP Keepalive.

Default: no

edns-tcp-keepalive-timeout: <*msec*>
The period Unbound will wait for a query on a TCP connection when EDNS TCP Keepalive is active. If this timeout expires Unbound closes the connection. If the client supports the EDNS TCP Keepalive option, Unbound sends the timeout value to the client to encourage it to close the connection before the server times out.

When the number of free incoming TCP buffers falls below 50% of the total number configured, the advertised timeout is progressively reduced to 1% of the configured value, then to 0.2% of the configured value if the number of free buffers falls below 35% of the total number configured, and finally to 0 if the number of free buffers falls below 20% of the total number configured. A minimum actual timeout of 200 milliseconds is observed regardless of the advertised timeout.

Default: 120000 (2 minutes)

sock-queue-timeout: <*sec*>
UDP queries that have waited in the socket buffer for a long time can be dropped. The time is set in seconds, 3 could be a good value to ignore old queries that likely the client does not need a reply for any more. This could happen if the host has not been able to service the queries for a while, i.e. Unbound is not running, and then is enabled again. It uses timestamp socket options.

Default: 0 (disabled)

tcp-upstream: *<yes or no>*
Enable or disable whether the upstream queries use TCP only for transport. Useful in tunneling scenarios. If set to no you can specify TCP transport only for selected forward or stub zones using *forward-tcp-upstream:* or *stub-tcp-upstream:* respectively.

Default: no

udp-upstream-without-downstream: *<yes or no>*
Enable UDP upstream even if *do-udp:* is no. Useful for TLS service providers, that want no UDP downstream but use UDP to fetch data upstream.

Default: no (no changes)

tls-upstream: *<yes or no>*
Enabled or disable whether the upstream queries use TLS only for transport. Useful in tunneling scenarios. The TLS contains plain DNS in TCP wireformat. The other server must support this (see *tls-service-key:*).

If you enable this, also configure a *tls-cert-bundle:* or use *tls-win-cert:* or *tls-system-cert:* to load CA certs, otherwise the connections cannot be authenticated.

This option enables TLS for all of them, but if you do not set this you can configure TLS specifically for some forward zones with *forward-tls-upstream:*. And also with *stub-tls-upstream:*.

Default: no

ssl-upstream: *<yes or no>*
Alternate syntax for *tls-upstream:*. If both are present in the config file the last is used.

tls-service-key: *<file>*
If enabled, the server provides DNS-over-TLS or DNS-over-HTTPS service on the TCP ports marked implicitly or explicitly for these services with *tls-port:* or *https-port:*. The file must contain the private key for the TLS session, the public certificate is in the *tls-service-pem:* file and it must also be specified if *tls-service-key:* is specified. Enabling or disabling this service requires a restart (a reload is not enough), because the key is read while root permissions are held and before chroot (if any). The ports enabled implicitly or explicitly via *tls-port:* and *https-port:* do not provide normal DNS TCP service.

Note: Unbound needs to be compiled with libnghttp2 in order to provide DNS-over-HTTPS.

Default: "" (disabled)

ssl-service-key: *<file>*
Alternate syntax for *tls-service-key:*.

tls-service-pem: *<file>*
The public key certificate pem file for the tls service.

Default: "" (disabled)

ssl-service-pem: *<file>*
Alternate syntax for *tls-service-pem:*.

tls-port: *<number>*
The port number on which to provide TCP TLS service. Only interfaces configured with that port number as @number get the TLS service.

Default: 853

ssl-port: *<number>*
Alternate syntax for *tls-port:*.

tls-cert-bundle: *<file>*
If null or "", no file is used. Set it to the certificate bundle file, for example `/etc/pki/tls/certs/ca-bundle.crt`. These certificates are used for authenticating connections made to outside peers. For example *auth-zone urls:*, and also DNS-over-TLS connections. It is read at start up before permission drop and chroot.

Default: "" (disabled)

ssl-cert-bundle: *<file>*
Alternate syntax for *tls-cert-bundle:*.

tls-win-cert: *<yes or no>*
Add the system certificates to the cert bundle certificates for authentication. If no cert bundle, it uses only these certificates. On windows this option uses the certificates from the cert store. Use the *tls-cert-bundle:* option on other systems. On other systems, this option enables the system certificates.

Default: no

tls-system-cert: *<yes or no>*
This the same attribute as the *tls-win-cert:* attribute, under a different name. Because it is not windows specific.

tls-additional-port: *<portnr>*
List port numbers as *tls-additional-port:*, and when interfaces are defined, eg. with the @port suffix, as this port number, they provide DNS-over-TLS service. Can list multiple, each on a new statement.

tls-session-ticket-keys: *<file>*
If not "", lists files with 80 bytes of random contents that are used to perform TLS session resumption for clients using the Unbound server. These files contain the secret key for the TLS session tickets. First key use to encrypt and decrypt TLS session tickets. Other keys use to decrypt only.

With this you can roll over to new keys, by generating a new first file and allowing decrypt of the old file by listing it after the first file for some time, after the wait clients are not using the old key any more and the old key can be removed. One way to create the file is:

```
dd if=/dev/random bs=1 count=80 of=ticket.dat
```

The first 16 bytes should be different from the old one if you create a second key, that is the name used to identify the key. Then there is 32 bytes random data for an AES key and then 32 bytes random data for the HMAC key.

Default: ""

tls-ciphers: *<string with cipher list>*
Set the list of ciphers to allow when serving TLS. Use "" for default ciphers.

Default: ""

tls-ciphersuites: *<string with ciphersuites list>*
Set the list of ciphersuites to allow when serving TLS. This is for newer TLS 1.3 connections. Use "" for default ciphersuites.

Default: ""

pad-responses: *<yes or no>*
If enabled, TLS serviced queries that contained an EDNS Padding option will cause responses padded to the closest multiple of the size specified in *pad-responses-block-size:*.

Default: yes

pad-responses-block-size: *<number>*
The block size with which to pad responses serviced over TLS. Only responses to padded queries will be padded.

Default: 468

pad-queries: *<yes or no>*
If enabled, all queries sent over TLS upstreams will be padded to the closest multiple of the size specified in *pad-queries-block-size:*.

Default: yes

pad-queries-block-size: *<number>*
The block size with which to pad queries sent over TLS upstreams.

Default: 128

tls-use-sni: *<yes or no>*
Enable or disable sending the SNI extension on TLS connections.

Note: Changing the value requires a reload.

Default: yes

https-port: *<number>*
The port number on which to provide DNS-over-HTTPS service. Only interfaces configured with that port number as @number get the HTTPS service.

Default: 443

http-endpoint: *<endpoint string>*
The HTTP endpoint to provide DNS-over-HTTPS service on.

Default: /dns-query

http-max-streams: *<number of streams>*
Number used in the SETTINGS_MAX_CONCURRENT_STREAMS parameter in the HTTP/2 SETTINGS frame for DNS-over-HTTPS connections.

Default: 100

http-query-buffer-size: *<size in bytes>*
Maximum number of bytes used for all HTTP/2 query buffers combined. These buffers contain (partial) DNS queries waiting for request stream completion. An RST_STREAM frame will be send to streams exceeding this limit. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 4m

http-response-buffer-size: *<size in bytes>*
Maximum number of bytes used for all HTTP/2 response buffers combined. These buffers contain DNS responses waiting to be written back to the clients. An RST_STREAM frame will be send to streams exceeding this limit. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 4m

http-nodelay: *<yes or no>*
Set TCP_NODELAY socket option on sockets used to provide DNS-over-HTTPS service. Ignored if the option is not available.

Default: yes

http-notls-downstream: *<yes or no>*
Disable use of TLS for the downstream DNS-over-HTTP connections. Useful for local back end servers.

Default: no

proxy-protocol-port: *<portnr>*
List port numbers as *proxy-protocol-port:*, and when interfaces are defined, eg. with the @port suffix, as this port number, they support and expect PROXYv2.

In this case the proxy address will only be used for the network communication and initial ACL (check if the proxy itself is denied/refused by configuration).

The proxied address (if any) will then be used as the true client address and will be used where applicable for logging, ACL, DNSTAP, RPZ and IP ratelimiting.

PROXYv2 is supported for UDP and TCP/TLS listening interfaces.

There is no support for PROXYv2 on a DoH or DNSCrypt listening interface.

Can list multiple, each on a new statement.

use-systemd: *<yes or no>*
Enable or disable systemd socket activation.

Default: no

do-daemonize: *<yes or no>*
Enable or disable whether the Unbound server forks into the background as a daemon. Set the value to no when Unbound runs as systemd service.

Default: yes

tcp-connection-limit: *<IP netblock> <limit>*
Allow up to limit simultaneous TCP connections from the given netblock. When at the limit, further connections are accepted but closed immediately. This option is experimental at this time.

Default: (disabled)

access-control: *<IP netblock> <action>*
Specify treatment of incoming queries from their originating IP address. Queries can be allowed to have access to this server that gives DNS answers, or refused, with other actions possible. The IP address range can be specified as a netblock, it is possible to give the statement several times in order to specify the treatment of different netblocks. The netblock is given as an IPv4 or IPv6 address with /size appended for a classless network block. The most specific netblock match is used, if none match *refuse* is used. The order of the access-control statements therefore does not matter. The action can be *deny*, *refuse*, *allow*, *allow_setrd*, *allow_snoop*, *allow_cookie*, *deny_non_local* or *refuse_non_local*.

deny
Stops queries from hosts from that netblock.

refuse
Stops queries too, but sends a DNS rcode REFUSED error message back.

allow
Gives access to clients from that netblock. It gives only access for recursion clients (which is what almost all clients need). Non-recursive queries are refused.

The *allow* action does allow non-recursive queries to access the local-data that is configured. The reason is that this does not involve the Unbound server recursive lookup algorithm, and static data is served in the reply. This supports normal operations where non-recursive queries are made for the authoritative data. For non-recursive queries any replies from the dynamic cache are refused.

allow_setrd
Ignores the recursion desired (RD) bit and treats all requests as if the recursion desired bit is set.

Note that this behavior violates **RFC 1034** which states that a name server should never perform recursive service unless asked via the RD bit since this interferes with trouble shooting of name servers and their databases. This prohibited behavior may be useful if another DNS server must forward requests for specific

zones to a resolver DNS server, but only supports stub domains and sends queries to the resolver DNS server with the RD bit cleared.

allow_snoop
Gives non-recursive access too. This gives both recursive and non recursive access. The name *allow_snoop* refers to cache snooping, a technique to use non-recursive queries to examine the cache contents (for malicious acts). However, non-recursive queries can also be a valuable debugging tool (when you want to examine the cache contents).

In that case use *allow_snoop* for your administration host.

allow_cookie
Allows access only to UDP queries that contain a valid DNS Cookie as specified in RFC 7873 and RFC 9018, when the *answer-cookie:* option is enabled. UDP queries containing only a DNS Client Cookie and no Server Cookie, or an invalid DNS Cookie, will receive a BADCOOKIE response including a newly generated DNS Cookie, allowing clients to retry with that DNS Cookie. The *allow_cookie* action will also accept requests over stateful transports, regardless of the presence of an DNS Cookie and regardless of the *answer-cookie:* setting. UDP queries without a DNS Cookie receive REFUSED responses with the TC flag set, that may trigger fall back to TCP for those clients.

deny_non_local, refuse_non_local
The *deny_non_local* and *refuse_non_local* actions are for hosts that are only allowed to query for the authoritative *local-data:*, they are not allowed full recursion but only the static data.

With *deny_non_local*, messages that are disallowed are dropped, with *refuse_non_local* they receive error code REFUSED.

By default only localhost is *allowed*, the rest is refused. The default is *refused*, because that is protocol-friendly. The DNS protocol is not designed to handle dropped packets due to policy, and dropping may result in (possibly excessive) retried queries.

access-control-tag: ***<IP netblock> "<list of tags>"***
Assign tags to *access-control:* elements. Clients using this access control element use localzones that are tagged with one of these tags.

Tags must be defined in *define-tag:*. Enclose list of tags in quotes ("") and put spaces between tags.

If *access-control-tag:* is configured for a netblock that does not have an *access-control:*, an access-control element with action *allow* is configured for this netblock.

access-control-tag-action: ***<IP netblock> <tag> <action>***
Set action for particular tag for given access control element. If you have multiple tag values, the tag used to lookup the action is the first tag match between *access-control-tag:* and *local-zone-tag:* where "first" comes from the order of the *define-tag:* values.

access-control-tag-data: ***<IP netblock> <tag> "<resource record string>"***
Set redirect data for particular tag for given access control element.

access-control-view: ***<IP netblock> <view name>***
Set view for given access control element.

interface-action: ***<ip address or interface name [@port]> <action>***
Similar to *access-control:* but for interfaces.

The action is the same as the ones defined under *access-control:*.

Default action for interfaces is *refuse*. By default only localhost (the IP netblock, not the loopback interface) is allowed through the default *access-control:* behavior.

Note: The interface needs to be already specified with *interface:* and that any **access-control*:** attribute

overrides all **interface-*:** attributes for targeted clients.

---

interface-tag: ***<ip address or interface name [@port]> <"list of tags">***
Similar to *access-control-tag:* but for interfaces.

Note: The interface needs to be already specified with *interface:* and that any **access-control*:** attribute overrides all **interface-*:** attributes for targeted clients.

---

interface-tag-action: ***<ip address or interface name [@port]> <tag> <action>***
Similar to *access-control-tag-action:* but for interfaces.

Note: The interface needs to be already specified with *interface:* and that any **access-control*:** attribute overrides all **interface-*:** attributes for targeted clients.

---

interface-tag-data: ***<ip address or interface name [@port]> <tag> <"resource record string">***
Similar to *access-control-tag-data:* but for interfaces.

Note: The interface needs to be already specified with *interface:* and that any **access-control*:** attribute overrides all **interface-*:** attributes for targeted clients.

---

interface-view: ***<ip address or interface name [@port]> <view name>***
Similar to *access-control-view:* but for interfaces.

Note: The interface needs to be already specified with *interface:* and that any **access-control*:** attribute overrides all **interface-*:** attributes for targeted clients.

---

chroot: ***<directory>***
If *chroot:* is enabled, you should pass the configfile (from the commandline) as a full path from the original root. After the chroot has been performed the now defunct portion of the config file path is removed to be able to reread the config after a reload.

All other file paths (working dir, logfile, roothints, and key files) can be specified in several ways: as an absolute path relative to the new root, as a relative path to the working directory, or as an absolute path relative to the original root. In the last case the path is adjusted to remove the unused portion.

The pidfile can be either a relative path to the working directory, or an absolute path relative to the original root. It is written just prior to chroot and dropping permissions. This allows the pidfile to be `/var/run/unbound.pid` and the chroot to be `/var/unbound`, for example. Note that Unbound is not able to remove the pidfile after termination when it is located outside of the chroot directory.

Additionally, Unbound may need to access `/dev/urandom` (for entropy) from inside the chroot.

If given, a *chroot(2)* is done to the given directory. If you give "" no *chroot(2)* is performed.

Default: /usr/local/etc/unbound

username: ***<name>***
If given, after binding the port the user privileges are dropped. If you give username: "" no user change is performed.

If this user is not capable of binding the port, reloads (by signal HUP) will still retain the opened ports. If you change the port number in the config file, and that new port number requires privileges, then a reload will fail; a restart is needed.

Default: unbound

directory: *<directory>*
Sets the working directory for the program. On Windows the string "%EXECUTABLE%" tries to change to the directory that **unbound.exe** resides in. If you give a *server: directory: <directory>* before *include:* file statements then those includes can be relative to the working directory.

Default: /usr/local/etc/unbound

logfile: *<filename>*
If "" is given, logging goes to stderr, or nowhere once daemonized. The logfile is appended to, in the following format:

```
[seconds since 1970] unbound[pid:tid]: type: message.
```

If this option is given, the *use-syslog:* attribute is set to "no". The logfile is reopened (for append) when the config file is reread, on SIGHUP.

Default: "" (disabled)

use-syslog: *<yes or no>*
Sets Unbound to send log messages to the syslogd, using *syslog(3)*. The log facility LOG_DAEMON is used, with identity "unbound". The logfile setting is overridden when *use-syslog:* is turned on.

Default: yes

log-identity: *<string>*
If "" is given, then the name of the executable, usually "unbound" is used to report to the log. Enter a string to override it with that, which is useful on systems that run more than one instance of Unbound, with different configurations, so that the logs can be easily distinguished against.

Default: ""

log-time-ascii: *<yes or no>*
Sets logfile lines to use a timestamp in UTC ASCII. No effect if using syslog, in that case syslog formats the timestamp printed into the log files.

Default: no (prints the seconds since 1970 in brackets)

log-queries: *<yes or no>*
Prints one line per query to the log, with the log timestamp and IP address, name, type and class. Note that it takes time to print these lines which makes the server (significantly) slower. Odd (nonprintable) characters in names are printed as `'?'`.

Default: no

log-replies: *<yes or no>*
Prints one line per reply to the log, with the log timestamp and IP address, name, type, class, return code, time to resolve, from cache and response size. Note that it takes time to print these lines which makes the server (significantly) slower. Odd (nonprintable) characters in names are printed as `'?'`.

Default: no

log-tag-queryreply: *<yes or no>*
Prints the word 'query' and 'reply' with *log-queries:* and *log-replies:*. This makes filtering logs easier.

Default: no (backwards compatible)

log-destaddr: *<yes or no>*
Prints the destination address, port and type in the *log-replies* output. This disambiguates what type of traffic, eg. UDP or TCP, and to what local port the traffic was sent to.

Default: no

log-local-actions: *<yes or no>*
Print log lines to inform about local zone actions. These lines are like the *local-zone type inform* print outs, but they are also printed for the other types of local zones.

Default: no

log-servfail: *<yes or no>*
Print log lines that say why queries return SERVFAIL to clients. This is separate from the verbosity debug logs, much smaller, and printed at the error level, not the info level of debug info from verbosity.

Default: no

pidfile: *<filename>*
The process id is written to the file. Default is "/usr/local/etc/unbound/unbound.pid". So,

```
kill -HUP `cat /usr/local/etc/unbound/unbound.pid`
```

triggers a reload,

```
kill -TERM `cat /usr/local/etc/unbound/unbound.pid`
```

gracefully terminates.

Default: /usr/local/etc/unbound/unbound.pid

root-hints: *<filename>*
Read the root hints from this file. Default is nothing, using builtin hints for the IN class. The file has the format of zone files, with root nameserver names and addresses only. The default may become outdated, when servers change, therefore it is good practice to use a root hints file.

Default: “”

hide-identity: *<yes or no>*
If enabled ‘id.server’ and ‘hostname.bind’ queries are REFUSED.

Default: no

identity: *<string>*
Set the identity to report. If set to "", then the hostname of the server is returned.

Default: “”

hide-version: *<yes or no>*
If enabled ‘version.server’ and ‘version.bind’ queries are REFUSED.

Default: no

version: *<string>*
Set the version to report. If set to "", then the package version is returned.

Default: “”

hide-http-user-agent: *<yes or no>*
If enabled the HTTP header User-Agent is not set. Use with caution as some webserver configurations may reject HTTP requests lacking this header. If needed, it is better to explicitly set the *http-user-agent:* below.

Default: no

http-user-agent: *<string>*
Set the HTTP User-Agent header for outgoing HTTP requests. If set to "", then the package name and version are used.

Default: “”

nsid: *<string>*
Add the specified nsid to the EDNS section of the answer when queried with an NSID EDNS enabled packet. As a sequence of hex characters or with 'ascii_' prefix and then an ASCII string.

Default: (disabled)

hide-trustanchor: *<yes or no>*
If enabled 'trustanchor.unbound' queries are REFUSED.

Default: no

target-fetch-policy: *<"list of numbers">*
Set the target fetch policy used by Unbound to determine if it should fetch nameserver target addresses opportunistically. The policy is described per dependency depth.

The number of values determines the maximum dependency depth that Unbound will pursue in answering a query. A value of -1 means to fetch all targets opportunistically for that dependency depth. A value of 0 means to fetch on demand only. A positive value fetches that many targets opportunistically.

Enclose the list between quotes ("") and put spaces between numbers. Setting all zeroes, "0 0 0 0 0" gives behaviour closer to that of BIND 9, while setting "-1 -1 -1 -1 -1" gives behaviour rumoured to be closer to that of BIND 8.

Default: "3 2 1 0 0"

harden-short-bufsize: *<yes or no>*
Very small EDNS buffer sizes from queries are ignored.

Default: on (as described in the standard)

harden-large-queries: *<yes or no>*
Very large queries are ignored. Default is off, since it is legal protocol wise to send these, and could be necessary for operation if TSIG or EDNS payload is very large.

Default: no

harden-glue: *<yes or no>*
Will trust glue only if it is within the servers authority.

Default: yes

harden-dnssec-stripped: *<yes or no>*
Require DNSSEC data for trust-anchored zones, if such data is absent, the zone becomes bogus. If turned off, and no DNSSEC data is received (or the DNSKEY data fails to validate), then the zone is made insecure, this behaves like there is no trust anchor. You could turn this off if you are sometimes behind an intrusive firewall (of some sort) that removes DNSSEC data from packets, or a zone changes from signed to unsigned to badly signed often. If turned off you run the risk of a downgrade attack that disables security for a zone.

Default: yes

harden-below-nxdomain: *<yes or no>*
From **RFC 8020** (with title "NXDOMAIN: There Really Is Nothing Underneath"), returns NXDOMAIN to queries for a name below another name that is already known to be NXDOMAIN. DNSSEC mandates NOERROR for empty nonterminals, hence this is possible. Very old software might return NXDOMAIN for empty nonterminals (that usually happen for reverse IP address lookups), and thus may be incompatible with this. To try to avoid this only DNSSEC-secure NXDOMAINs are used, because the old software does not have DNSSEC.

Note: The NXDOMAIN must be secure, this means NSEC3 with optout is insufficient.

Default: yes

harden-referral-path: *<yes or no>*
Harden the referral path by performing additional queries for infrastructure data. Validates the replies if trust anchors are configured and the zones are signed. This enforces DNSSEC validation on nameserver NS sets and the nameserver addresses that are encountered on the referral path to the answer. Default is off, because it burdens the authority servers, and it is not RFC standard, and could lead to performance problems because of the extra query load that is generated. Experimental option. If you enable it consider adding more numbers after the *target-fetch-policy:* to increase the max depth that is checked to.

Default: no

harden-algo-downgrade: *<yes or no>*
Harden against algorithm downgrade when multiple algorithms are advertised in the DS record. If no, allows the weakest algorithm to validate the zone. Zone signers must produce zones that allow this feature to work, but sometimes they do not, and turning this option off avoids that validation failure.

Default: no

harden-unknown-additional: *<yes or no>*
Harden against unknown records in the authority section and additional section. If no, such records are copied from the upstream and presented to the client together with the answer. If yes, it could hamper future protocol developments that want to add records.

Default: no

use-caps-for-id: *<yes or no>*
Use 0x20-encoded random bits in the query to foil spoof attempts. This perturbs the lowercase and uppercase of query names sent to authority servers and checks if the reply still has the correct casing. This feature is an experimental implementation of draft dns-0x20.

Default: no

caps-exempt: *<domain>*
Exempt the domain so that it does not receive caps-for-id perturbed queries. For domains that do not support 0x20 and also fail with fallback because they keep sending different answers, like some load balancers. Can be given multiple times, for different domains.

caps-whitelist: *<yes or no>*
Alternate syntax for *caps-exempt:*.

qname-minimisation: *<yes or no>*
Send minimum amount of information to upstream servers to enhance privacy. Only send minimum required labels of the QNAME and set QTYPE to A when possible. Best effort approach; full QNAME and original QTYPE will be sent when upstream replies with a RCODE other than NOERROR, except when receiving NXDOMAIN from a DNSSEC signed zone.

Default: yes

qname-minimisation-strict: *<yes or no>*
QNAME minimisation in strict mode. Do not fall-back to sending full QNAME to potentially broken nameservers. A lot of domains will not be resolvable when this option in enabled. Only use if you know what you are doing. This option only has effect when *qname-minimisation:* is enabled.

Default: no

aggressive-nsec: *<yes or no>*
Aggressive NSEC uses the DNSSEC NSEC chain to synthesize NXDOMAIN and other denials, using information from previous NXDOMAINs answers. It helps to reduce the query rate towards targets that get a very high nonexistent name lookup rate.

Default: yes

private-address: *<IP address or subnet>*
Give IPv4 of IPv6 addresses or classless subnets. These are addresses on your private network, and are not allowed to be returned for public internet names. Any occurrence of such addresses are removed from DNS answers. Additionally, the DNSSEC validator may mark the answers bogus. This protects against so-called DNS Rebinding, where a user browser is turned into a network proxy, allowing remote access through the browser to other parts of your private network.

Some names can be allowed to contain your private addresses, by default all the *local-data:* that you configured is allowed to, and you can specify additional names using *private-domain:*. No private addresses are enabled by default.

We consider to enable this for the **RFC 1918** private IP address space by default in later releases. That would enable private addresses for `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `169.254.0.0/16`, `fd00::/8` and `fe80::/10`, since the RFC standards say these addresses should not be visible on the public internet.

Turning on `127.0.0.0/8` would hinder many spamblocklists as they use that. Adding `::ffff:0:0/96` stops IPv4-mapped IPv6 addresses from bypassing the filter.

private-domain: *<domain name>*
Allow this domain, and all its subdomains to contain private addresses. Give multiple times to allow multiple domain names to contain private addresses.

Default: (none)

unwanted-reply-threshold: *<number>*
If set, a total number of unwanted replies is kept track of in every thread. When it reaches the threshold, a defensive action is taken and a warning is printed to the log. The defensive action is to clear the rrset and message caches, hopefully flushing away any poison. A value of 10 million is suggested.

Default: 0 (disabled)

do-not-query-address: *<IP address>*
Do not query the given IP address. Can be IPv4 or IPv6. Append /num to indicate a classless delegation netblock, for example like `10.2.3.4/24` or `2001::11/64`.

Default: (none)

do-not-query-localhost: *<yes or no>*
If yes, localhost is added to the *do-not-query-address:* entries, both IPv6 `::1` and IPv4 `127.0.0.1/8`. If no, then localhost can be used to send queries to.

Default: yes

prefetch: *<yes or no>*
If yes, message cache elements are prefetched before they expire to keep the cache up to date. Turning it on gives about 10 percent more traffic and load on the machine, but popular items do not expire from the cache.

Default: no

prefetch-key: *<yes or no>*
If yes, fetch the DNSKEYs earlier in the validation process, when a DS record is encountered. This lowers the latency of requests. It does use a little more CPU. Also if the cache is set to 0, it is no use.

Default: no

deny-any: *<yes or no>*
If yes, deny queries of type ANY with an empty response. If disabled, Unbound responds with a short list of resource records if some can be found in the cache and makes the upstream type ANY query if there are none.

Default: no

rrset-roundrobin: *<yes or no>*
If yes, Unbound rotates RRSet order in response (the random number is taken from the query ID, for speed and thread safety).

Default: yes

minimal-responses: *<yes or no>*
If yes, Unbound does not insert authority/additional sections into response messages when those sections are not required. This reduces response size significantly, and may avoid TCP fallback for some responses. This may cause a slight speedup.

The default is yes, even though the DNS protocol RFCs mandate these sections, and the additional content could be of use and save roundtrips for clients. Because they are not used, and the saved roundtrips are easier saved with prefetch, whilst this is faster.

Default: yes

disable-dnssec-lame-check: *<yes or no>*
If true, disables the DNSSEC lameness check in the iterator. This check sees if RRSIGs are present in the answer, when dnssec is expected, and retries another authority if RRSIGs are unexpectedly missing. The validator will insist in RRSIGs for DNSSEC signed domains regardless of this setting, if a trust anchor is loaded.

Default: no

module-config: *"<module names>"*
Module configuration, a list of module names separated by spaces, surround the string with quotes (""). The modules can be `respip`, `validator`, or `iterator` (and possibly more, see below).

Note: The ordering of the modules is significant, the order decides the order of processing.

Setting this to just "iterator" will result in a non-validating server. Setting this to "validator iterator" will turn on DNSSEC validation.

Note: You must also set trust-anchors for validation to be useful.

Adding `respip` to the front will cause RPZ processing to be done on all queries.

The default is "validator iterator".

When the server is built with EDNS client subnet support the default is "subnetcache validator iterator".

Most modules that need to be listed here have to be listed at the beginning of the line.

The `subnetcache` module has to be listed just before the iterator.

The `python` module can be listed in different places, it then processes the output of the module it is just before.

The `dynlib` module can be listed pretty much anywhere, it is only a very thin wrapper that allows dynamic libraries to run in its place.

trust-anchor-file: *<filename>*
File with trusted keys for validation. Both DS and DNSKEY entries can appear in the file. The format of the file is the standard DNS Zone file format.

Default: "" (no trust anchor file)

auto-trust-anchor-file: *<filename>*
File with trust anchor for one zone, which is tracked with **RFC 5011** probes. The probes are run several times per month, thus the machine must be online frequently. The initial file can be one with contents as described in *trust-anchor-file:*. The file is written to when the anchor is updated, so the Unbound user must have write

permission. Write permission to the file, but also to the directory it is in (to create a temporary file, which is necessary to deal with filesystem full events), it must also be inside the *chroot:* (if that is used).

Default: "" (no auto trust anchor file)

trust-anchor: ***"<Resource Record>"***
A DS or DNSKEY RR for a key to use for validation. Multiple entries can be given to specify multiple trusted keys, in addition to the *trust-anchor-file:*. The resource record is entered in the same format as *dig(1)* or *drill(1)* prints them, the same format as in the zone file. Has to be on a single line, with "" around it. A TTL can be specified for ease of cut and paste, but is ignored. A class can be specified, but class IN is default.

Default: (none)

trusted-keys-file: ***<filename>***
File with trusted keys for validation. Specify more than one file with several entries, one file per entry. Like *trust-anchor-file:* but has a different file format. Format is BIND-9 style format, the `trusted-keys { name flag proto algo "key"; };` clauses are read. It is possible to use wildcards with this statement, the wildcard is expanded on start and on reload.

Default: "" (no trusted keys file)

trust-anchor-signaling: ***<yes or no>***
Send **RFC 8145** key tag query after trust anchor priming.

Default: yes

root-key-sentinel: ***<yes or no>***
Root key trust anchor sentinel.

Default: yes

domain-insecure: ***<domain name>***
Sets *<domain name>* to be insecure, DNSSEC chain of trust is ignored towards the *<domain name>*. So a trust anchor above the domain name can not make the domain secure with a DS record, such a DS record is then ignored. Can be given multiple times to specify multiple domains that are treated as if unsigned. If you set trust anchors for the domain they override this setting (and the domain is secured).

This can be useful if you want to make sure a trust anchor for external lookups does not affect an (unsigned) internal domain. A DS record externally can create validation failures for that internal domain.

Default: (none)

val-override-date: ***<rrsig-style date spec>***

> **Warning:** Debugging feature!

If enabled by giving a RRSIG style date, that date is used for verifying RRSIG inception and expiration dates, instead of the current date. Do not set this unless you are debugging signature inception and expiration. The value -1 ignores the date altogether, useful for some special applications.

Default: 0 (disabled)

val-sig-skew-min: ***<seconds>***
Minimum number of seconds of clock skew to apply to validated signatures. A value of 10% of the signature lifetime (expiration - inception) is used, capped by this setting. Default is 3600 (1 hour) which allows for daylight savings differences. Lower this value for more strict checking of short lived signatures.

Default: 3600 (1 hour)

val-sig-skew-max: ***<seconds>***
Maximum number of seconds of clock skew to apply to validated signatures. A value of 10% of the signature

lifetime (expiration - inception) is used, capped by this setting. Default is 86400 (24 hours) which allows for time-zone setting problems in stable domains. Setting both min and max very low disables the clock skew allowances. Setting both min and max very high makes the validator check the signature timestamps less strictly.

Default: 86400 (24 hours)

val-max-restart: <*number*>
The maximum number the validator should restart validation with another authority in case of failed validation.

Default: 5

val-bogus-ttl: <*seconds*>
The time to live for bogus data. This is data that has failed validation; due to invalid signatures or other checks. The TTL from that data cannot be trusted, and this value is used instead. The time interval prevents repeated revalidation of bogus data.

Default: 60

val-clean-additional: <*yes or no*>
Instruct the validator to remove data from the additional section of secure messages that are not signed properly. Messages that are insecure, bogus, indeterminate or unchecked are not affected. Use this setting to protect the users that rely on this validator for authentication from potentially bad data in the additional section.

Default: yes

val-log-level: <*number*>
Have the validator print validation failures to the log. Regardless of the verbosity setting.

At 1, for every user query that fails a line is printed to the logs. This way you can monitor what happens with validation. Use a diagnosis tool, such as dig or drill, to find out why validation is failing for these queries.

At 2, not only the query that failed is printed but also the reason why Unbound thought it was wrong and which server sent the faulty data.

Default: 0 (disabled)

val-permissive-mode: <*yes or no*>
Instruct the validator to mark bogus messages as indeterminate. The security checks are performed, but if the result is bogus (failed security), the reply is not withheld from the client with SERVFAIL as usual. The client receives the bogus data. For messages that are found to be secure the AD bit is set in replies. Also logging is performed as for full validation.

Default: no

ignore-cd-flag: <*yes or no*>
Instruct Unbound to ignore the CD flag from clients and refuse to return bogus answers to them. Thus, the CD (Checking Disabled) flag does not disable checking any more. This is useful if legacy (w2008) servers that set the CD flag but cannot validate DNSSEC themselves are the clients, and then Unbound provides them with DNSSEC protection.

Default: no

disable-edns-do: <*yes or no*>
Disable the EDNS DO flag in upstream requests. It breaks DNSSEC validation for Unbound's clients. This results in the upstream name servers to not include DNSSEC records in their replies and could be helpful for devices that cannot handle DNSSEC information. When the option is enabled, clients that set the DO flag receive no EDNS record in the response to indicate the lack of support to them. If this option is enabled but Unbound is already configured for DNSSEC validation (i.e., the validator module is enabled; default) this option is implicitly turned off with a warning as to not break DNSSEC validation in Unbound.

Default: no

serve-expired: *<yes or no>*
If enabled, Unbound attempts to serve old responses from cache with a TTL of *serve-expired-reply-ttl:* in the response without waiting for the actual resolution to finish. The actual resolution answer ends up in the cache later on.

Default: no

serve-expired-ttl: *<seconds>*
Limit serving of expired responses to configured seconds after expiration. 0 disables the limit. This option only applies when *serve-expired:* is enabled. A suggested value per **RFC 8767** is between 86400 (1 day) and 259200 (3 days).

Default: 0

serve-expired-ttl-reset: *<yes or no>*
Set the TTL of expired records to the *serve-expired-ttl:* value after a failed attempt to retrieve the record from upstream. This makes sure that the expired records will be served as long as there are queries for it.

Default: no

serve-expired-reply-ttl: *<seconds>*
TTL value to use when replying with expired data. If *serve-expired-client-timeout:* is also used then it is RECOMMENDED to use 30 as the value (**RFC 8767**).

Default: 30

serve-expired-client-timeout: *<msec>*
Time in milliseconds before replying to the client with expired data. This essentially enables the serve-stale behavior as specified in **RFC 8767** that first tries to resolve before immediately responding with expired data. A recommended value per **RFC 8767** is 1800. Setting this to 0 will disable this behavior.

Default: 0

serve-original-ttl: *<yes or no>*
If enabled, Unbound will always return the original TTL as received from the upstream name server rather than the decrementing TTL as stored in the cache. This feature may be useful if Unbound serves as a front-end to a hidden authoritative name server.

Enabling this feature does not impact cache expiry, it only changes the TTL Unbound embeds in responses to queries.

Note: Enabling this feature implicitly disables enforcement of the configured minimum and maximum TTL, as it is assumed users who enable this feature do not want Unbound to change the TTL obtained from an upstream server.

Note: The values set using *cache-min-ttl:* and *cache-max-ttl:* are ignored.

Default: no

val-nsec3-keysize-iterations: *<"list of values">*
List of keysize and iteration count values, separated by spaces, surrounded by quotes. This determines the maximum allowed NSEC3 iteration count before a message is simply marked insecure instead of performing the many hashing iterations. The list must be in ascending order and have at least one entry. If you set it to "1024 65535" there is no restriction to NSEC3 iteration values.

Note: This table must be kept short; a very long list could cause slower operation.

Default: "1024 150 2048 150 4096 150"

zonemd-permissive-mode: <*yes or no*>
If enabled the ZONEMD verification failures are only logged and do not cause the zone to be blocked and only return servfail. Useful for testing out if it works, or if the operator only wants to be notified of a problem without disrupting service.

Default: no

add-holddown: <*seconds*>
Instruct the *auto-trust-anchor-file:* probe mechanism for **RFC 5011** autotrust updates to add new trust anchors only after they have been visible for this time.

Default: 2592000 (30 days as per the RFC)

del-holddown: <*seconds*>
Instruct the *auto-trust-anchor-file:* probe mechanism for **RFC 5011** autotrust updates to remove revoked trust anchors after they have been kept in the revoked list for this long.

Default: 2592000 (30 days as per the RFC)

keep-missing: <*seconds*>
Instruct the *auto-trust-anchor-file:* probe mechanism for **RFC 5011** autotrust updates to remove missing trust anchors after they have been unseen for this long. This cleans up the state file if the target zone does not perform trust anchor revocation, so this makes the auto probe mechanism work with zones that perform regular (non-5011) rollovers. The value 0 does not remove missing anchors, as per the RFC.

Default: 31622400 (366 days)

permit-small-holddown: <*yes or no*>
Debug option that allows the autotrust 5011 rollover timers to assume very small values.

Default: no

key-cache-size: <*number*>
Number of bytes size of the key cache. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 4m

key-cache-slabs: <*number*>
Number of slabs in the key cache. Slabs reduce lock contention by threads. Must be set to a power of 2. Setting (close) to the number of cpus is a reasonable guess.

Default: 4

neg-cache-size: <*number*>
Number of bytes size of the aggressive negative cache. A plain number is in bytes, append 'k', 'm' or 'g' for kilobytes, megabytes or gigabytes (1024*1024 bytes in a megabyte).

Default: 1m

unblock-lan-zones: <*yes or no*>
If enabled, then for private address space, the reverse lookups are no longer filtered. This allows Unbound when running as dns service on a host where it provides service for that host, to put out all of the queries for the 'lan' upstream. When enabled, only localhost, `127.0.0.1` reverse and `::1` reverse zones are configured with default local zones. Disable the option when Unbound is running as a (DHCP-) DNS network resolver for a group of machines, where such lookups should be filtered (RFC compliance), this also stops potential data leakage about the local network to the upstream DNS servers.

Default: no

insecure-lan-zones: *<yes or no>*
If enabled, then reverse lookups in private address space are not validated. This is usually required whenever *unblock-lan-zones:* is used.

Default: no

local-zone: *<zone>* *<type>*
Configure a local zone. The type determines the answer to give if there is no match from *local-data:*. The types are *deny*, *refuse*, *static*, *transparent*, *redirect*, *nodefault*, *typetransparent*, *inform*, *inform_deny*, *inform_redirect*, *always_transparent*, *block_a*, *always_refuse*, *always_nxdomain*, *always_null*, *noview*, and are explained below. After that the default settings are listed. Use *local-data:* to enter data into the local zone. Answers for local zones are authoritative DNS answers. By default the zones are class IN.

If you need more complicated authoritative data, with referrals, wildcards, CNAME/DNAME support, or DNSSEC authoritative service, setup a *stub-zone:* for it as detailed in the stub zone section below. A *stub-zone:* can be used to have unbound send queries to another server, an authoritative server, to fetch the information. With a *forward-zone:*, unbound sends queries to a server that is a recursive server to fetch the information. With an *auth-zone:* a zone can be loaded from file and used, it can be used like a local zone for users downstream, or the *auth-zone:* information can be used to fetch information from when resolving like it is an upstream server. The *forward-zone:* and *auth-zone:* options are described in their sections below. If you want to perform filtering of the information that the users can fetch, the *local-zone:* and *local-data:* statements allow for this, but also the *rpz:* functionality can be used, described in the RPZ section.

deny
Do not send an answer, drop the query. If there is a match from local data, the query is answered.

refuse
Send an error message reply, with rcode REFUSED. If there is a match from local data, the query is answered.

static
If there is a match from local data, the query is answered. Otherwise, the query is answered with NODATA or NXDOMAIN. For a negative answer a SOA is included in the answer if present as *local-data:* for the zone apex domain.

transparent
If there is a match from *local-data:*, the query is answered. Otherwise if the query has a different name, the query is resolved normally. If the query is for a name given in *local-data:* but no such type of data is given in localdata, then a NOERROR NODATA answer is returned. If no *local-zone:* is given *local-data:* causes a transparent zone to be created by default.

typetransparent
If there is a match from local data, the query is answered. If the query is for a different name, or for the same name but for a different type, the query is resolved normally. So, similar to *transparent* but types that are not listed in local data are resolved normally, so if an A record is in the local data that does not cause a NODATA reply for AAAA queries.

redirect
The query is answered from the local data for the zone name. There may be no local data beneath the zone name. This answers queries for the zone, and all subdomains of the zone with the local data for the zone. It can be used to redirect a domain to return a different address record to the end user, with

```
local-zone: "example.com." redirect
local-data: "example.com. A 127.0.0.1"
```

queries for `www.example.com` and `www.foo.example.com` are redirected, so that users with web browsers cannot access sites with suffix example.com.

inform
The query is answered normally, same as *transparent*. The client IP address (@portnumber) is printed to the logfile. The log message is:

```
timestamp, unbound-pid, info: zonename inform IP@port queryname type class.
```

This option can be used for normal resolution, but machines looking up infected names are logged, eg. to run antivirus on them.

inform_deny
The query is dropped, like *deny*, and logged, like *inform*. Ie. find infected machines without answering the queries.

inform_redirect
The query is redirected, like *redirect*, and logged, like *inform*. Ie. answer queries with fixed data and also log the machines that ask.

always_transparent
Like *transparent*, but ignores local data and resolves normally.

block_a
Like *transparent*, but ignores local data and resolves normally all query types excluding A. For A queries it unconditionally returns NODATA. Useful in cases when there is a need to explicitly force all apps to use IPv6 protocol and avoid any queries to IPv4.

always_refuse
Like *refuse*, but ignores local data and refuses the query.

always_nxdomain
Like *static*, but ignores local data and returns NXDOMAIN for the query.

always_nodata
Like *static*, but ignores local data and returns NODATA for the query.

always_deny
Like *deny*, but ignores local data and drops the query.

always_null
Always returns `0.0.0.0` or `::0` for every name in the zone. Like *redirect* with zero data for A and AAAA. Ignores local data in the zone. Used for some block lists.

noview
Breaks out of that view and moves towards the global local zones for answer to the query. If the *view-first:* is no, it'll resolve normally. If *view-first:* is enabled, it'll break perform that step and check the global answers. For when the view has view specific overrides but some zone has to be answered from global local zone contents.

nodefault
Used to turn off default contents for AS112 zones. The other types also turn off default contents for the zone. The *nodefault* option has no other effect than turning off default contents for the given zone. Use *nodefault* if you use exactly that zone, if you want to use a subzone, use *transparent*.

The default zones are localhost, reverse `127.0.0.1` and `::1`, the home.arpa, onion, test, invalid and the AS112 zones. The AS112 zones are reverse DNS zones for private use and reserved IP addresses for which the servers on the internet cannot provide correct answers. They are configured by default to give NXDOMAIN (no reverse information) answers.

The defaults can be turned off by specifying your own *local-zone:* of that name, or using the *nodefault* type. Below is a list of the default zone contents.

localhost
The IPv4 and IPv6 localhost information is given. NS and SOA records are provided for completeness and to satisfy some DNS update tools. Default content:

```
local-zone: "localhost." redirect
local-data: "localhost. 10800 IN NS localhost."
local-data: "localhost. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200␣
↪604800 10800"
local-data: "localhost. 10800 IN A 127.0.0.1"
local-data: "localhost. 10800 IN AAAA ::1"
```

reverse IPv4 loopback
Default content:

```
local-zone: "127.in-addr.arpa." static
local-data: "127.in-addr.arpa. 10800 IN NS localhost."
local-data: "127.in-addr.arpa. 10800 IN SOA localhost. nobody.invalid. 1 3600␣
↪1200 604800 10800"
local-data: "1.0.0.127.in-addr.arpa. 10800 IN PTR localhost."
```

reverse IPv6 loopback
Default content:

```
local-zone: "1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
↪ip6.arpa." static
local-data: "1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
↪ip6.arpa. 10800 IN NS localhost."
local-data: "1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
↪ip6.arpa. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200 604800 10800"
local-data: "1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
↪ip6.arpa. 10800 IN PTR localhost."
```

home.arpa (RFC 8375)
Default content:

```
local-zone: "home.arpa." static
local-data: "home.arpa. 10800 IN NS localhost."
local-data: "home.arpa. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200␣
↪604800 10800"
```

onion (RFC 7686)
Default content:

```
local-zone: "onion." static
local-data: "onion. 10800 IN NS localhost."
local-data: "onion. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200 604800␣
↪10800"
```

test (RFC 6761)
Default content:

```
local-zone: "test." static
local-data: "test. 10800 IN NS localhost."
local-data: "test. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200 604800␣
↪10800"
```

invalid (RFC 6761)
Default content:

```
local-zone: "invalid." static
local-data: "invalid. 10800 IN NS localhost."
local-data: "invalid. 10800 IN SOA localhost. nobody.invalid. 1 3600 1200␣
↪604800 10800"
```

reverse RFC 1918 local use zones
Reverse data for zones `10.in-addr.arpa`, `16.172.in-addr.arpa` to `31.172.in-addr.arpa`, `168.192.in-addr.arpa`. The *local-zone:* is set static and as *local-data:* SOA and NS records are provided.

reverse RFC 3330 IP4 this, link-local, testnet and broadcast
Reverse data for zones `0.in-addr.arpa`, `254.169.in-addr.arpa`, `2.0.192.in-addr.arpa` (TEST NET 1), `100.51.198.in-addr.arpa` (TEST NET 2), `113.0.203.in-addr.arpa` (TEST NET 3), `255.255.255.255.in-addr.arpa`. And from `64.100.in-addr.arpa` to `127.100.in-addr.arpa` (Shared Address Space).

reverse RFC 4291 IPv6 unspecified
Reverse data for zone `0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.ip6.arpa.`

reverse RFC 4193 IPv6 Locally Assigned Local Addresses
Reverse data for zone `D.F.ip6.arpa`.

reverse RFC 4291 IPv6 Link Local Addresses
Reverse data for zones `8.E.F.ip6.arpa` to `B.E.F.ip6.arpa`.

reverse IPv6 Example Prefix
Reverse data for zone `8.B.D.0.1.0.0.2.ip6.arpa`. This zone is used for tutorials and examples. You can remove the block on this zone with:

```
local-zone: 8.B.D.0.1.0.0.2.ip6.arpa. nodefault
```

You can also selectively unblock a part of the zone by making that part transparent with a *local-zone:* statement. This also works with the other default zones.

local-data: ***"<resource record string>"***
Configure local data, which is served in reply to queries for it. The query has to match exactly unless you configure the *local-zone:* as redirect. If not matched exactly, the *local-zone:* type determines further processing. If *local-data:* is configured that is not a subdomain of a *local-zone:*, a *transparent local-zone* is configured. For record types such as TXT, use single quotes, as in:

```
local-data: 'example. TXT "text"'
```

Note: If you need more complicated authoritative data, with referrals, wildcards, CNAME/DNAME support, or DNSSEC authoritative service, setup a *stub-zone:* for it as detailed in the stub zone section below.

---

local-data-ptr: ***"IPaddr name"***
Configure local data shorthand for a PTR record with the reversed IPv4 or IPv6 address and the host name. For example `"192.0.2.4 www.example.com"`. TTL can be inserted like this: `"2001:DB8::4 7200 www.example.com"`

local-zone-tag: ***<zone> <"list of tags">***
Assign tags to local zones. Tagged localzones will only be applied when the used *access-control:* element has a matching tag. Tags must be defined in *define-tag:*. Enclose list of tags in quotes (`""`) and put spaces between

tags. When there are multiple tags it checks if the intersection of the list of tags for the query and *local-zone-tag:* is non-empty.

local-zone-override: <*zone*> <*IP netblock*> <*type*>
Override the local zone type for queries from addresses matching netblock. Use this localzone type, regardless the type configured for the local zone (both tagged and untagged) and regardless the type configured using *access-control-tag-action:*.

response-ip: <*IP-netblock*> <*action*>
This requires use of the `respip` module.

If the IP address in an AAAA or A RR in the answer section of a response matches the specified IP netblock, the specified action will apply. <*action*> has generally the same semantics as that for *access-control-tag-action:*, but there are some exceptions.

Actions for *response-ip:* are different from those for *local-zone:* in that in case of the former there is no point of such conditions as "the query matches it but there is no local data". Because of this difference, the semantics of *response-ip:* actions are modified or simplified as follows: The *static*, *refuse*, *transparent*, *typetransparent*, and *nodefault* actions are invalid for *response-ip*. Using any of these will cause the configuration to be rejected as faulty. The *deny* action is non-conditional, i.e. it always results in dropping the corresponding query. The resolution result before applying the *deny* action is still cached and can be used for other queries.

response-ip-data: <*IP-netblock*> <*"resource record string"*>
This requires use of the `respip` module.

This specifies the action data for *response-ip:* with action being to redirect as specified by <*"resource record string"*>. <*"Resource record string"*> is similar to that of *access-control-tag-action:*, but it must be of either AAAA, A or CNAME types. If the <*IP-netblock*> is an IPv6/IPv4 prefix, the record must be AAAA/A respectively, unless it is a CNAME (which can be used for both versions of IP netblocks). If it is CNAME there must not be more than one *response-ip-data:* for the same <*IP-netblock*>. Also, CNAME and other types of records must not coexist for the same <*IP-netblock*>, following the normal rules for CNAME records. The textual domain name for the CNAME does not have to be explicitly terminated with a dot ("."); the root name is assumed to be the origin for the name.

response-ip-tag: <*IP-netblock*> <*"list of tags"*>
This requires use of the `respip` module.

Assign tags to response <*IP-netblock*>. If the IP address in an AAAA or A RR in the answer section of a response matches the specified <*IP-netblock*>, the specified tags are assigned to the IP address. Then, if an *access-control-tag:* is defined for the client and it includes one of the tags for the response IP, the corresponding *access-control-tag-action:* will apply. Tag matching rule is the same as that for *access-control-tag:* and *local-zone:*. Unlike *local-zone-tag:*, *response-ip-tag:* can be defined for an <*IP-netblock*> even if no *response-ip:* is defined for that netblock. If multiple *response-ip-tag:* options are specified for the same <*IP-netblock*> in different statements, all but the first will be ignored. However, this will not be flagged as a configuration error, but the result is probably not what was intended.

Actions specified in an *access-control-tag-action:* that has a matching tag with *response-ip-tag:* can be those that are "invalid" for *response-ip:* listed above, since *access-control-tag-action:* can be shared with local zones. For these actions, if they behave differently depending on whether local data exists or not in case of local zones, the behavior for *response-ip-data:* will generally result in NOERROR/NODATA instead of NXDOMAIN, since the *response-ip:* data are inherently type specific, and non-existence of data does not indicate anything about the existence or non-existence of the qname itself. For example, if the matching tag action is static but there is no data for the corresponding *response-ip:* configuration, then the result will be NOERROR/NODATA. The only case where NXDOMAIN is returned is when an *always_nxdomain* action applies.

ratelimit: <*number or 0*>
Enable ratelimiting of queries sent to nameserver for performing recursion. 0 disables the feature. This option is experimental at this time.

The ratelimit is in queries per second that are allowed. More queries are turned away with an error (SERVFAIL). Cached responses are not ratelimited by this setting.

This stops recursive floods, eg. random query names, but not spoofed reflection floods. The zone of the query is determined by examining the nameservers for it, the zone name is used to keep track of the rate. For example, 1000 may be a suitable value to stop the server from being overloaded with random names, and keeps unbound from sending traffic to the nameservers for those zones.

Note: Configured forwarders are excluded from ratelimiting.

Default: 0

ratelimit-size: ***<memory size>***
Give the size of the data structure in which the current ongoing rates are kept track in. In bytes or use m(mega), k(kilo), g(giga). The ratelimit structure is small, so this data structure likely does not need to be large.

Default: 4m

ratelimit-slabs: ***<number>***
Give power of 2 number of slabs, this is used to reduce lock contention in the ratelimit tracking data structure. Close to the number of CPUs is a fairly good setting.

Default: 4

ratelimit-factor: ***<number>***
Set the amount of queries to rate limit when the limit is exceeded. If set to 0, all queries are dropped for domains where the limit is exceeded. If set to another value, 1 in that number is allowed through to complete. Default is 10, allowing 1/10 traffic to flow normally. This can make ordinary queries complete (if repeatedly queried for), and enter the cache, whilst also mitigating the traffic flow by the factor given.

Default: 10

ratelimit-backoff: ***<yes or no>***
If enabled, the ratelimit is treated as a hard failure instead of the default maximum allowed constant rate. When the limit is reached, traffic is ratelimited and demand continues to be kept track of for a 2 second rate window. No traffic is allowed, except for *ratelimit-factor:*, until demand decreases below the configured ratelimit for a 2 second rate window. Useful to set *ratelimit:* to a suspicious rate to aggressively limit unusually high traffic.

Default: no

ratelimit-for-domain: ***<domain>*** ***<number qps or 0>***
Override the global *ratelimit:* for an exact match domain name with the listed number. You can give this for any number of names. For example, for a top-level-domain you may want to have a higher limit than other names. A value of 0 will disable ratelimiting for that domain.

ratelimit-below-domain: ***<domain>*** ***<number qps or 0>***
Override the global *ratelimit:* for a domain name that ends in this name. You can give this multiple times, it then describes different settings in different parts of the namespace. The closest matching suffix is used to determine the qps limit. The rate for the exact matching domain name is not changed, use *ratelimit-for-domain:* to set that, you might want to use different settings for a top-level-domain and subdomains. A value of 0 will disable ratelimiting for domain names that end in this name.

ip-ratelimit: ***<number or 0>***
Enable global ratelimiting of queries accepted per ip address. This option is experimental at this time. The ratelimit is in queries per second that are allowed. More queries are completely dropped and will not receive a reply, SERVFAIL or otherwise. IP ratelimiting happens before looking in the cache. This may be useful for mitigating amplification attacks. Clients with a valid DNS Cookie will bypass the ratelimit. If a ratelimit for such clients is still needed, *ip-ratelimit-cookie* can be used instead.

Default: 0 (disabled)

ip-ratelimit-cookie: *<number or 0>*
Enable global ratelimiting of queries accepted per IP address with a valid DNS Cookie. This option is experimental at this time. The ratelimit is in queries per second that are allowed. More queries are completely dropped and will not receive a reply, SERVFAIL or otherwise. IP ratelimiting happens before looking in the cache. This option could be useful in combination with *allow_cookie*, in an attempt to mitigate other amplification attacks than UDP reflections (e.g., attacks targeting Unbound itself) which are already handled with DNS Cookies. If used, the value is suggested to be higher than *ip-ratelimit:* e.g., tenfold.

Default: 0 (disabled)

ip-ratelimit-size: *<memory size>*
Give the size of the data structure in which the current ongoing rates are kept track in. In bytes or use m(mega), k(kilo), g(giga). The IP ratelimit structure is small, so this data structure likely does not need to be large.

Default: 4m

ip-ratelimit-slabs: *<number>*
Give power of 2 number of slabs, this is used to reduce lock contention in the IP ratelimit tracking data structure. Close to the number of cpus is a fairly good setting.

Default: 4

ip-ratelimit-factor: *<number>*
Set the amount of queries to rate limit when the limit is exceeded. If set to 0, all queries are dropped for addresses where the limit is exceeded. If set to another value, 1 in that number is allowed through to complete. Default is 10, allowing 1/10 traffic to flow normally. This can make ordinary queries complete (if repeatedly queried for), and enter the cache, whilst also mitigating the traffic flow by the factor given.

Default: 10

ip-ratelimit-backoff: *<yes or no>*
If enabled, the rate limit is treated as a hard failure instead of the default maximum allowed constant rate. When the limit is reached, traffic is ratelimited and demand continues to be kept track of for a 2 second rate window. No traffic is allowed, except for *ip-ratelimit-factor:*, until demand decreases below the configured ratelimit for a 2 second rate window. Useful to set *ip-ratelimit:* to a suspicious rate to aggressively limit unusually high traffic.

Default: no

outbound-msg-retry: *<number>*
The number of retries, per upstream nameserver in a delegation, that Unbound will attempt in case a throwaway response is received. No response (timeout) contributes to the retry counter. If a forward/stub zone is used, this is the number of retries per nameserver in the zone.

Default: 5

max-sent-count: *<number>*
Hard limit on the number of outgoing queries Unbound will make while resolving a name, making sure large NS sets do not loop. Results in SERVFAIL when reached. It resets on query restarts (e.g., CNAME) and referrals.

Default: 32

max-query-restarts: *<number>*
Hard limit on the number of times Unbound is allowed to restart a query upon encountering a CNAME record. Results in SERVFAIL when reached. Changing this value needs caution as it can allow long CNAME chains to be accepted, where Unbound needs to verify (resolve) each link individually.

Default: 11

fast-server-permil: *<number>*
Specify how many times out of 1000 to pick from the set of fastest servers. 0 turns the feature off. A value

of 900 would pick from the fastest servers 90 percent of the time, and would perform normal exploration of random servers for the remaining time. When *prefetch:* is enabled (or *serve-expired:*), such prefetches are not sped up, because there is no one waiting for it, and it presents a good moment to perform server exploration. The *fast-server-num:* option can be used to specify the size of the fastest servers set.

Default: 0

fast-server-num: ***<number>***
Set the number of servers that should be used for fast server selection. Only use the fastest specified number of servers with the *fast-server-permil:* option, that turns this on or off.

Default: 3

answer-cookie: ***<yes or no>***
If enabled, Unbound will answer to requests containing DNS Cookies as specified in RFC 7873 and RFC 9018.

Default: no

cookie-secret: ***"<128 bit hex string>"***
Server's secret for DNS Cookie generation. Useful to explicitly set for servers in an anycast deployment that need to share the secret in order to verify each other's Server Cookies. An example hex string would be "000102030405060708090a0b0c0d0e0f".

Default: 128 bits random secret generated at startup time

edns-client-string: ***<IP netblock>*** ***<string>***
Include an EDNS0 option containing configured ASCII string in queries with destination address matching the configured *<IP netblock>*. This configuration option can be used multiple times. The most specific match will be used.

edns-client-string-opcode: ***<opcode>***
EDNS0 option code for the *edns-client-string:* option, from 0 to 65535. A value from the 'Reserved for Local/Experimental' range (65001-65534) should be used.

Default: 65001

ede: ***<yes or no>***
If enabled, Unbound will respond with Extended DNS Error codes (**RFC 8914**). These EDEs attach informative error messages to a response for various errors.

When the *val-log-level:* option is also set to 2, responses with Extended DNS Errors concerning DNSSEC failures that are not served from cache, will also contain a descriptive text message about the reason for the failure.

Default: no

ede-serve-expired: ***<yes or no>***
If enabled, Unbound will attach an Extended DNS Error (**RFC 8914**) *Code 3 - Stale Answer* as EDNS0 option to the expired response.

Note: This will not attach the EDE code without setting *ede: yes* as well.

Default: no

18.4.2 Remote Control Options

In the **remote-control:** clause are the declarations for the remote control facility. If this is enabled, the *unbound-control(8)* utility can be used to send commands to the running Unbound server. The server uses these clauses to setup TLSv1 security for the connection. The *unbound-control(8)* utility also reads the **remote-control:** section for options. To setup the correct self-signed certificates use the *unbound-control-setup(8)* utility.

control-enable: *<yes or no>*
The option is used to enable remote control. If turned off, the server does not listen for control commands.

Default: no

control-interface: *<IP address or interface name or path>*
Give IPv4 or IPv6 addresses or local socket path to listen on for control commands. If an interface name is used instead of an IP address, the list of IP addresses on that interface are used.

By default localhost (`127.0.0.1` and `::1`) is listened to. Use `0.0.0.0` and `::0` to listen to all interfaces. If you change this and permissions have been dropped, you must restart the server for the change to take effect.

If you set it to an absolute path, a unix domain socket is used. This socket does not use the certificates and keys, so those files need not be present. To restrict access, Unbound sets permissions on the file to the user and group that is configured, the access bits are set to allow the group members to access the control socket file. Put users that need to access the socket in the that group. To restrict access further, create a directory to put the control socket in and restrict access to that directory.

control-port: *<port number>*
The port number to listen on for IPv4 or IPv6 control interfaces.

Note: If you change this and permissions have been dropped, you must restart the server for the change to take effect.

Default: 8953

control-use-cert: *<yes or no>*
For localhost *control-interface:* you can disable the use of TLS by setting this option to "no". For local sockets, TLS is disabled and the value of this option is ignored.

Default: yes

server-key-file: *<private key file>*
Path to the server private key. This file is generated by the *unbound-control-setup(8)* utility. This file is used by the Unbound server, but not by *unbound-control(8)*.

Default: unbound_server.key

server-cert-file: *<certificate file.pem>*
Path to the server self signed certificate. This file is generated by the *unbound-control-setup(8)* utility. This file is used by the Unbound server, and also by *unbound-control(8)*.

Default: unbound_server.pem

control-key-file: *<private key file>*
Path to the control client private key. This file is generated by the *unbound-control-setup(8)* utility. This file is used by *unbound-control(8)*.

Default: unbound_control.key

control-cert-file: *<certificate file.pem>*
Path to the control client certificate. This certificate has to be signed with the server certificate. This file is generated by the *unbound-control-setup(8)* utility. This file is used by *unbound-control(8)*.

Default: unbound_control.pem

18.4.3 Stub Zone Options

There may be multiple **stub-zone:** clauses. Each with a *name:* and zero or more hostnames or IP addresses. For the stub zone this list of nameservers is used. Class IN is assumed. The servers should be authority servers, not recursors; Unbound performs the recursive processing itself for stub zones.

The stub zone can be used to configure authoritative data to be used by the resolver that cannot be accessed using the public internet servers. This is useful for company-local data or private zones. Setup an authoritative server on a different host (or different port). Enter a config entry for Unbound with:

```
stub-addr: <ip address of host[@port]>
```

The Unbound resolver can then access the data, without referring to the public internet for it.

This setup allows DNSSEC signed zones to be served by that authoritative server, in which case a trusted key entry with the public key can be put in config, so that Unbound can validate the data and set the AD bit on replies for the private zone (authoritative servers do not set the AD bit). This setup makes Unbound capable of answering queries for the private zone, and can even set the AD bit ('authentic'), but the AA ('authoritative') bit is not set on these replies.

Consider adding *server:* statements for *domain-insecure:* and for *local-zone: <name> nodefault* for the zone if it is a locally served zone. The insecure clause stops DNSSEC from invalidating the zone. The *local-zone: nodefault* (or *transparent*) clause makes the (reverse-) zone bypass Unbound's filtering of **RFC 1918** zones.

name: <*domain name*>
Name of the stub zone. This is the full domain name of the zone.

stub-host: <*domain name*>
Name of stub zone nameserver. Is itself resolved before it is used.

To use a non-default port for DNS communication append `'@'` with the port number.

If TLS is enabled, then you can append a `'#'` and a name, then it'll check the TLS authentication certificates with that name.

If you combine the `'@'` and `'#'`, the `'@'` comes first. If only `'#'` is used the default port is the configured *tls-port:*.

stub-addr: <*IP address*>
IP address of stub zone nameserver. Can be IPv4 or IPv6.

To use a non-default port for DNS communication append `'@'` with the port number.

If TLS is enabled, then you can append a `'#'` and a name, then it'll check the tls authentication certificates with that name.

If you combine the `'@'` and `'#'`, the `'@'` comes first. If only `'#'` is used the default port is the configured *tls-port:*.

stub-prime: <*yes or no*>
If enabled it performs NS set priming, which is similar to root hints, where it starts using the list of nameservers currently published by the zone. Thus, if the hint list is slightly outdated, the resolver picks up a correct list online.

Default: no

stub-first: <*yes or no*>
If enabled, a query is attempted without the stub clause if it fails. The data could not be retrieved and would have caused SERVFAIL because the servers are unreachable, instead it is tried without this clause.

Default: no

stub-tls-upstream: *<yes or no>*
Enabled or disable whether the queries to this stub use TLS for transport.

Default: no

stub-ssl-upstream: *<yes or no>*
Alternate syntax for *stub-tls-upstream:*.

stub-tcp-upstream: *<yes or no>*
If it is set to "yes" then upstream queries use TCP only for transport regardless of global flag *tcp-upstream:*.

Default: no

stub-no-cache: *<yes or no>*
If enabled, data inside the stub is not cached. This is useful when you want immediate changes to be visible.

Default: no

18.4.4 Forward Zone Options

There may be multiple **forward-zone:** clauses. Each with a *name:* and zero or more hostnames or IP addresses. For the forward zone this list of nameservers is used to forward the queries to. The servers listed as *forward-host:* and *forward-addr:* have to handle further recursion for the query. Thus, those servers are not authority servers, but are (just like Unbound is) recursive servers too; Unbound does not perform recursion itself for the forward zone, it lets the remote server do it. Class IN is assumed. CNAMEs are chased by Unbound itself, asking the remote server for every name in the indirection chain, to protect the local cache from illegal indirect referenced items. A *forward-zone:* entry with name `"."` and a *forward-addr:* target will forward all queries to that other server (unless it can answer from the cache).

name: *<domain name>*
Name of the forward zone. This is the full domain name of the zone.

forward-host: *<domain name>*
Name of server to forward to. Is itself resolved before it is used.

To use a non-default port for DNS communication append `'@'` with the port number.

If TLS is enabled, then you can append a `'#'` and a name, then it'll check the TLS authentication certificates with that name.

If you combine the `'@'` and `'#'`, the `'@'` comes first. If only `'#'` is used the default port is the configured *tls-port:*.

forward-addr: *<IP address>*
IP address of server to forward to. Can be IPv4 or IPv6.

To use a non-default port for DNS communication append `'@'` with the port number.

If TLS is enabled, then you can append a `'#'` and a name, then it'll check the tls authentication certificates with that name.

If you combine the `'@'` and `'#'`, the `'@'` comes first. If only `'#'` is used the default port is the configured *tls-port:*.

At high verbosity it logs the TLS certificate, with TLS enabled. If you leave out the `'#'` and auth name from the *forward-addr:*, any name is accepted. The cert must also match a CA from the *tls-cert-bundle:*.

forward-first: *<yes or no>*
If a forwarded query is met with a SERVFAIL error, and this option is enabled, Unbound will fall back to normal recursive resolution for this query as if no query forwarding had been specified.

Default: no

forward-tls-upstream: *<yes or no>*
Enabled or disable whether the queries to this forwarder use TLS for transport. If you enable this, also configure a *tls-cert-bundle:* or use *tls-win-cert:* to load CA certs, otherwise the connections cannot be authenticated.

Default: no

forward-ssl-upstream: *<yes or no>*
Alternate syntax for *forward-tls-upstream:*.

forward-tcp-upstream: *<yes or no>*
If it is set to "yes" then upstream queries use TCP only for transport regardless of global flag *tcp-upstream:*.

Default: no

forward-no-cache: *<yes or no>*
If enabled, data inside the forward is not cached. This is useful when you want immediate changes to be visible.

Default: no

18.4.5 Authority Zone Options

Authority zones are configured with **auth-zone:**, and each one must have a *name:*. There can be multiple ones, by listing multiple auth-zone clauses, each with a different name, pertaining to that part of the namespace. The authority zone with the name closest to the name looked up is used. Authority zones can be processed on two distinct, non-exclusive, configurable stages.

With *for-downstream: yes* (default), authority zones are processed after **local-zones** and before cache. When used in this manner, Unbound responds like an authority server with no further processing other than returning an answer from the zone contents. A notable example, in this case, is CNAME records which are returned verbatim to downstream clients without further resolution.

With *for-upstream: yes* (default), authority zones are processed after the cache lookup, just before going to the network to fetch information for recursion. When used in this manner they provide a local copy of an authority server that speeds up lookups for that data during resolving.

If both options are enabled (default), client queries for an authority zone are answered authoritatively from Unbound, while internal queries that require data from the authority zone consult the local zone data instead of going to the network.

An interesting configuration is *for-downstream: no*, *for-upstream: yes* that allows for hyperlocal behavior where both client and internal queries consult the local zone data while resolving. In this case, the aforementioned CNAME example will result in a thoroughly resolved answer.

Authority zones can be read from zonefile. And can be kept updated via AXFR and IXFR. After update the zonefile is rewritten. The update mechanism uses the SOA timer values and performs SOA UDP queries to detect zone changes.

If the update fetch fails, the timers in the SOA record are used to time another fetch attempt. Until the SOA expiry timer is reached. Then the zone is expired. When a zone is expired, queries are SERVFAIL, and any new serial number is accepted from the primary (even if older), and if fallback is enabled, the fallback activates to fetch from the upstream instead of the SERVFAIL.

name: *<zone name>*
Name of the authority zone.

primary: *<IP address or host name>*
Where to download a copy of the zone from, with AXFR and IXFR. Multiple primaries can be specified. They are all tried if one fails.

To use a non-default port for DNS communication append `'@'` with the port number.

You can append a '#' and a name, then AXFR over TLS can be used and the TLS authentication certificates will be checked with that name.

If you combine the '@' and '#', the '@' comes first. If you point it at another Unbound instance, it would not work because that does not support AXFR/IXFR for the zone, but if you used *url:* to download the zonefile as a text file from a webserver that would work.

If you specify the hostname, you cannot use the domain from the zonefile, because it may not have that when retrieving that data, instead use a plain IP address to avoid a circular dependency on retrieving that IP address.

master: *<IP address or host name>*
Alternate syntax for *primary:*.

url: *<URL to zone file>*
Where to download a zonefile for the zone. With HTTP or HTTPS. An example for the url is:

```
http://www.example.com/example.org.zone
```

Multiple url statements can be given, they are tried in turn.

If only urls are given the SOA refresh timer is used to wait for making new downloads. If also primaries are listed, the primaries are first probed with UDP SOA queries to see if the SOA serial number has changed, reducing the number of downloads. If none of the urls work, the primaries are tried with IXFR and AXFR.

For HTTPS, the *tls-cert-bundle:* and the hostname from the url are used to authenticate the connection.

If you specify a hostname in the URL, you cannot use the domain from the zonefile, because it may not have that when retrieving that data, instead use a plain IP address to avoid a circular dependency on retrieving that IP address.

Avoid dependencies on name lookups by using a notation like "http://192.0.2.1/unbound-primaries/example.com.zone", with an explicit IP address.

allow-notify: *<IP address or host name or netblockIP/prefix>*
With *allow-notify:* you can specify additional sources of notifies. When notified, the server attempts to first probe and then zone transfer. If the notify is from a primary, it first attempts that primary. Otherwise other primaries are attempted. If there are no primaries, but only urls, the file is downloaded when notified.

Note: The primaries from *primary:* and *url:* statements are allowed notify by default.

---

fallback-enabled: *<yes or no>*
If enabled, Unbound falls back to querying the internet as a resolver for this zone when lookups fail. For example for DNSSEC validation failures.

Default: no

for-downstream: *<yes or no>*
If enabled, Unbound serves authority responses to downstream clients for this zone. This option makes Unbound behave, for the queries with names in this zone, like one of the authority servers for that zone.

Turn it off if you want Unbound to provide recursion for the zone but have a local copy of zone data.

If *for-downstream: no* and *for-upstream: yes*, then Unbound will DNSSEC validate the contents of the zone before serving the zone contents to clients and store validation results in the cache.

Default: yes

for-upstream: *<yes or no>*
If enabled, Unbound fetches data from this data collection for answering recursion queries. Instead of sending queries over the internet to the authority servers for this zone, it'll fetch the data directly from the zone data.

Turn it on when you want Unbound to provide recursion for downstream clients, and use the zone data as a local copy to speed up lookups.

Default: yes

zonemd-check: *<yes or no>*
Enable this option to check ZONEMD records in the zone. The ZONEMD record is a checksum over the zone data. This includes glue in the zone and data from the zone file, and excludes comments from the zone file. When there is a DNSSEC chain of trust, DNSSEC signatures are checked too.

Default: no

zonemd-reject-absence: *<yes or no>*
Enable this option to reject the absence of the ZONEMD record. Without it, when ZONEMD is not there it is not checked.

It is useful to enable for a non-DNSSEC signed zone where the operator wants to require the verification of a ZONEMD, hence a missing ZONEMD is a failure.

The action upon failure is controlled by the *zonemd-permissive-mode:* option, for log only or also block the zone.

Without the option, absence of a ZONEMD is only a failure when the zone is DNSSEC signed, and we have a trust anchor, and the DNSSEC verification of the absence of the ZONEMD fails. With the option enabled, the absence of a ZONEMD is always a failure, also for nonDNSSEC signed zones.

Default: no

zonefile: *<filename>*
The filename where the zone is stored. If not given then no zonefile is used. If the file does not exist or is empty, Unbound will attempt to fetch zone data (eg. from the primary servers).

18.4.6 View Options

There may be multiple **view:** clauses. Each with a *name:* and zero or more *local-zone:* and *local-data:* attributes. Views can also contain *view-first:*, *response-ip:*, *response-ip-data:* and *local-data-ptr:* attributes. View can be mapped to requests by specifying the view name in an *access-control-view:* attribute. Options from matching views will override global options. Global options will be used if no matching view is found, or when the matching view does not have the option specified.

name: *<view name>*
Name of the view. Must be unique. This name is used in the *access-control-view:* attribute.

local-zone: *<zone>* *<type>*
View specific local zone elements. Has the same types and behaviour as the global *local-zone:* elements. When there is at least one *local-zone:* specified and *view-first: no*, the default local-zones will be added to this view. Defaults can be disabled using the nodefault type. When *view-first: yes* or when a view does not have a *local-zone:*, the global *local-zone:* will be used including it's default zones.

local-data: *"<resource record string>"*
View specific local data elements. Has the same behaviour as the global *local-data:* elements.

local-data-ptr: *"IPaddr name"*
View specific local-data-ptr elements. Has the same behaviour as the global *local-data-ptr:* elements.

view-first: *<yes or no>*
If enabled, it attempts to use the global *local-zone:* and *local-data:* if there is no match in the view specific options.

Default: no

18.4.7 Python Module Options

The **python:** clause gives the settings for the *python(1)* script module. This module acts like the iterator and validator modules do, on queries and answers. To enable the script module it has to be compiled into the daemon, and the word `python` has to be put in the *module-config:* option (usually first, or between the validator and iterator). Multiple instances of the python module are supported by adding the word `python` more than once.

If the *chroot:* option is enabled, you should make sure Python's library directory structure is bind mounted in the new root environment, see *mount(8)*. Also the *python-script:* path should be specified as an absolute path relative to the new root, or as a relative path to the working directory.

python-script: ***<python file>***
The script file to load. Repeat this option for every python module instance added to the *module-config:* option.

18.4.8 Dynamic Library Module Options

The **dynlib:** clause gives the settings for the `dynlib` module. This module is only a very small wrapper that allows dynamic modules to be loaded on runtime instead of being compiled into the application. To enable the dynlib module it has to be compiled into the daemon, and the word `dynlib` has to be put in the *module-config:* attribute. Multiple instances of dynamic libraries are supported by adding the word `dynlib` more than once.

The *dynlib-file:* path should be specified as an absolute path relative to the new path set by *chroot:*, or as a relative path to the working directory.

dynlib-file: ***<dynlib file>***
The dynamic library file to load. Repeat this option for every dynlib module instance added to the *module-config:* option.

18.4.9 DNS64 Module Options

The `dns64` module must be configured in the *module-config:* directive, e.g.:

```
module-config: "dns64 validator iterator"
```

and be compiled into the daemon to be enabled.

Note: These settings go in the *server:* section.

dns64-prefix: ***<IPv6 prefix>***
This sets the DNS64 prefix to use to synthesize AAAA records with. It must be /96 or shorter.

Default: 64:ff9b::/96

dns64-synthall: ***<yes or no>***

Warning: Debugging feature!

If enabled, synthesize all AAAA records despite the presence of actual AAAA records.

Default: no

dns64-ignore-aaaa: ***<domain name>***
List domain for which the AAAA records are ignored and the A record is used by DNS64 processing instead. Can

be entered multiple times, list a new domain for which it applies, one per line. Applies also to names underneath the name given.

18.4.10 NAT64 Operation

NAT64 operation allows using a NAT64 prefix for outbound requests to IPv4-only servers. It is controlled by two options in the *server:* section:

do-nat64: *<yes or no>*
Use NAT64 to reach IPv4-only servers. Consider also enabling *prefer-ip6:* to prefer native IPv6 connections to nameservers.

Default: no

nat64-prefix: *<IPv6 prefix>*
Use a specific NAT64 prefix to reach IPv4-only servers. The prefix length must be one of /32, /40, /48, /56, /64 or /96.

Default: 64:ff9b::/96 (same as *dns64-prefix:*)

18.4.11 DNSCrypt Options

The **dnscrypt:** clause gives the settings of the dnscrypt channel. While those options are available, they are only meaningful if Unbound was compiled with `--enable-dnscrypt`. Currently certificate and secret/public keys cannot be generated by Unbound. You can use dnscrypt-wrapper to generate those: https://github.com/cofyc/dnscrypt-wrapper/blob/master/README.md#usage

dnscrypt-enable: *<yes or no>*
Whether or not the dnscrypt config should be enabled. You may define configuration but not activate it.

Default: no

dnscrypt-port: *<port number>*
On which port should dnscrypt should be activated.

Note: There should be a matching interface option defined in the *server:* section for this port.

---

dnscrypt-provider: *<provider name>*
The provider name to use to distribute certificates. This is of the form: `2.dnscrypt-cert.example.com.`. The name *MUST* end with a dot.

dnscrypt-secret-key: *<path to secret key file>*
Path to the time limited secret key file. This option may be specified multiple times.

dnscrypt-provider-cert: *<path to cert file>*
Path to the certificate related to the *dnscrypt-secret-key:*. This option may be specified multiple times.

dnscrypt-provider-cert-rotated: *<path to cert file>*
Path to a certificate that we should be able to serve existing connection from but do not want to advertise over *dnscrypt-provider:* 's TXT record certs distribution.

A typical use case is when rotating certificates, existing clients may still use the client magic from the old cert in their queries until they fetch and update the new cert. Likewise, it would allow one to prime the new cert/key without distributing the new cert yet, this can be useful when using a network of servers using anycast and on which the configuration may not get updated at the exact same time.

By priming the cert, the servers can handle both old and new certs traffic while distributing only one.

This option may be specified multiple times.

dnscrypt-shared-secret-cache-size: <*memory size*>
Give the size of the data structure in which the shared secret keys are kept in. In bytes or use m(mega), k(kilo), g(giga). The shared secret cache is used when a same client is making multiple queries using the same public key. It saves a substantial amount of CPU.

Default: 4m

dnscrypt-shared-secret-cache-slabs: <*number*>
Give power of 2 number of slabs, this is used to reduce lock contention in the dnscrypt shared secrets cache. Close to the number of cpus is a fairly good setting.

Default: 4

dnscrypt-nonce-cache-size: <*memory size*>
Give the size of the data structure in which the client nonces are kept in. In bytes or use m(mega), k(kilo), g(giga). The nonce cache is used to prevent dnscrypt message replaying. Client nonce should be unique for any pair of client pk/server sk.

Default: 4m

dnscrypt-nonce-cache-slabs: <*number*>
Give power of 2 number of slabs, this is used to reduce lock contention in the dnscrypt nonce cache. Close to the number of cpus is a fairly good setting.

Default: 4

18.4.12 EDNS Client Subnet Module Options

The ECS module must be configured in the *module-config:* directive, e.g.:

```
module-config: "subnetcache validator iterator"
```

and be compiled into the daemon to be enabled.

Note: These settings go in the *server:* section.

If the destination address is allowed in the configuration Unbound will add the EDNS0 option to the query containing the relevant part of the client's address. When an answer contains the ECS option the response and the option are placed in a specialized cache. If the authority indicated no support, the response is stored in the regular cache.

Additionally, when a client includes the option in its queries, Unbound will forward the option when sending the query to addresses that are explicitly allowed in the configuration using *send-client-subnet:*. The option will always be forwarded, regardless the allowed addresses, when *client-subnet-always-forward: yes*. In this case the lookup in the regular cache is skipped.

The maximum size of the ECS cache is controlled by *msg-cache-size:* in the configuration file. On top of that, for each query only 100 different subnets are allowed to be stored for each address family. Exceeding that number, older entries will be purged from cache.

This module does not interact with the *serve-expired*:* and *prefetch:* options.

send-client-subnet: <*IP address*>
Send client source address to this authority. Append /num to indicate a classless delegation netblock, for example like `10.2.3.4/24` or `2001::11/64`. Can be given multiple times. Authorities not listed will not receive edns-subnet information, unless domain in query is specified in *client-subnet-zone:*.

client-subnet-zone: *<domain>*
Send client source address in queries for this domain and its subdomains. Can be given multiple times. Zones not listed will not receive edns-subnet information, unless hosted by authority specified in *send-client-subnet:*.

client-subnet-always-forward: *<yes or no>*
Specify whether the ECS address check (configured using *send-client-subnet:*) is applied for all queries, even if the triggering query contains an ECS record, or only for queries for which the ECS record is generated using the querier address (and therefore did not contain ECS data in the client query). If enabled, the address check is skipped when the client query contains an ECS record. And the lookup in the regular cache is skipped.

Default: no

max-client-subnet-ipv6: *<number>*
Specifies the maximum prefix length of the client source address we are willing to expose to third parties for IPv6.

Default: 56

max-client-subnet-ipv4: *<number>*
Specifies the maximum prefix length of the client source address we are willing to expose to third parties for IPv4.

Default: 24

min-client-subnet-ipv6: *<number>*
Specifies the minimum prefix length of the IPv6 source mask we are willing to accept in queries. Shorter source masks result in REFUSED answers. Source mask of 0 is always accepted.

Default: 0

min-client-subnet-ipv4: *<number>*
Specifies the minimum prefix length of the IPv4 source mask we are willing to accept in queries. Shorter source masks result in REFUSED answers. Source mask of 0 is always accepted. Default: 0

max-ecs-tree-size-ipv4: *<number>*
Specifies the maximum number of subnets ECS answers kept in the ECS radix tree. This number applies for each qname/qclass/qtype tuple.

Default: 100

max-ecs-tree-size-ipv6: *<number>*
Specifies the maximum number of subnets ECS answers kept in the ECS radix tree. This number applies for each qname/qclass/qtype tuple.

Default: 100

18.4.13 Opportunistic IPsec Support Module Options

The IPsec module must be configured in the *module-config:* directive, e.g.:

```
module-config: "ipsecmod validator iterator"
```

and be compiled into Unbound by using `--enable-ipsecmod` to be enabled.

Note: These settings go in the *server:* section.

When Unbound receives an A/AAAA query that is not in the cache and finds a valid answer, it will withhold returning the answer and instead will generate an IPSECKEY subquery for the same domain name. If an answer was found, Unbound will call an external hook passing the following arguments:

QNAME
Domain name of the A/AAAA and IPSECKEY query. In string format.

IPSECKEY TTL
TTL of the IPSECKEY RRset.

A/AAAA
String of space separated IP addresses present in the A/AAAA RRset. The IP addresses are in string format.

IPSECKEY
String of space separated IPSECKEY RDATA present in the IPSECKEY RRset. The IPSECKEY RDATA are in DNS presentation format.

The A/AAAA answer is then cached and returned to the client. If the external hook was called the TTL changes to ensure it doesn't surpass *ipsecmod-max-ttl:*.

The same procedure is also followed when *prefetch: yes* is used, but the A/AAAA answer is given to the client before the hook is called. *ipsecmod-max-ttl:* ensures that the A/AAAA answer given from cache is still relevant for opportunistic IPsec.

ipsecmod-enabled: ***<yes or no>***
Specifies whether the IPsec module is enabled or not. The IPsec module still needs to be defined in the *module-config:* directive. This option facilitates turning on/off the module without restarting/reloading Unbound.

Default: yes

ipsecmod-hook: ***<filename>***
Specifies the external hook that Unbound will call with *system(3)*. The file can be specified as an absolute/relative path. The file needs the proper permissions to be able to be executed by the same user that runs Unbound. It must be present when the IPsec module is defined in the *module-config:* directive.

ipsecmod-strict: ***<yes or no>***
If enabled Unbound requires the external hook to return a success value of 0. Failing to do so Unbound will reply with SERVFAIL. The A/AAAA answer will also not be cached.

Default: no

ipsecmod-max-ttl: ***<seconds>***
Time to live maximum for A/AAAA cached records after calling the external hook.

Default: 3600

ipsecmod-ignore-bogus: ***<yes or no>***
Specifies the behaviour of Unbound when the IPSECKEY answer is bogus. If set to yes, the hook will be called and the A/AAAA answer will be returned to the client. If set to no, the hook will not be called and the answer to the A/AAAA query will be SERVFAIL. Mainly used for testing.

Default: no

ipsecmod-allow: ***<domain>***
Allow the IPsec module functionality for the domain so that the module logic will be executed. Can be given multiple times, for different domains. If the option is not specified, all domains are treated as being allowed (default).

ipsecmod-whitelist: ***<yes or no>***
Alternate syntax for *ipsecmod-allow:*.

18.4.14 Cache DB Module Options

The Cache DB module must be configured in the *module-config:* directive, e.g.:

```
module-config: "validator cachedb iterator"
```

and be compiled into the daemon with `--enable-cachedb`.

If this module is enabled and configured, the specified backend database works as a second level cache; when Unbound cannot find an answer to a query in its built-in in-memory cache, it consults the specified backend. If it finds a valid answer in the backend, Unbound uses it to respond to the query without performing iterative DNS resolution. If Unbound cannot even find an answer in the backend, it resolves the query as usual, and stores the answer in the backend.

This module interacts with the *serve-expired-** options and will reply with expired data if Unbound is configured for that. Currently the use of *serve-expired-client-timeout:* and *serve-expired-reply-ttl:* is not consistent for data originating from the external cache as these will result in a reply with 0 TTL without trying to update the data first, ignoring the configured values.

If Unbound was built with `--with-libhiredis` on a system that has installed the hiredis C client library of Redis, then the `redis` backend can be used. This backend communicates with the specified Redis server over a TCP connection to store and retrieve cache data. It can be used as a persistent and/or shared cache backend.

Note: Unbound never removes data stored in the Redis server, even if some data have expired in terms of DNS TTL or the Redis server has cached too much data; if necessary the Redis server must be configured to limit the cache size, preferably with some kind of least-recently-used eviction policy.

Additionally, the *redis-expire-records:* option can be used in order to set the relative DNS TTL of the message as timeout to the Redis records; keep in mind that some additional memory is used per key and that the expire information is stored as absolute Unix timestamps in Redis (computer time must be stable).

This backend uses synchronous communication with the Redis server based on the assumption that the communication is stable and sufficiently fast. The thread waiting for a response from the Redis server cannot handle other DNS queries. Although the backend has the ability to reconnect to the server when the connection is closed unexpectedly and there is a configurable timeout in case the server is overly slow or hangs up, these cases are assumed to be very rare. If connection close or timeout happens too often, Unbound will be effectively unusable with this backend. It's the administrator's responsibility to make the assumption hold.

The **cachedb:** clause gives custom settings of the cache DB module.

backend: *<backend name>*
: Specify the backend database name. The default database is the in-memory backend named `testframe`, which, as the name suggests, is not of any practical use. Depending on the build-time configuration, `redis` backend may also be used as described above.

  Default: testframe

secret-seed: *"<secret string>"*
: Specify a seed to calculate a hash value from query information. This value will be used as the key of the corresponding answer for the backend database and can be customized if the hash should not be predictable operationally. If the backend database is shared by multiple Unbound instances, all instances must use the same secret seed.

  Default: "default"

cachedb-no-store: *<yes or no>*
: If the backend should be read from, but not written to. This makes this instance not store dns messages in the backend. But if data is available it is retrieved.

  Default: no

The following **cachedb:** options are specific to the `redis` backend.

redis-server-host: *<server address or name>*
The IP (either v6 or v4) address or domain name of the Redis server. In general an IP address should be specified as otherwise Unbound will have to resolve the name of the server every time it establishes a connection to the server.

Default: 127.0.0.1

redis-server-port: *<port number>*
The TCP port number of the Redis server.

Default: 6379

redis-server-path: *<unix socket path>*
The unix socket path to connect to the redis server. Unix sockets may have better throughput than the IP address option.

Default: "" (disabled)

redis-server-password: *"<password>"*
The Redis AUTH password to use for the redis server. Only relevant if Redis is configured for client password authorisation.

Default: "" (disabled)

redis-timeout: *<msec>*
The period until when Unbound waits for a response from the Redis sever. If this timeout expires Unbound closes the connection, treats it as if the Redis server does not have the requested data, and will try to re-establish a new connection later.

Default: 100

redis-expire-records: *<yes or no>*
If Redis record expiration is enabled. If yes, Unbound sets timeout for Redis records so that Redis can evict keys that have expired automatically. If Unbound is configured with *serve-expired:* and *serve-expired-ttl: 0*, this option is internally reverted to "no".

Note: Redis SETEX support is required for this option (Redis >= 2.0.0).

Default: no

redis-logical-db: *<logical database index>*
The logical database in Redis to use. These are databases in the same Redis instance sharing the same configuration and persisted in the same RDB/AOF file. If unsure about using this option, Redis documentation (https://redis.io/commands/select/) suggests not to use a single Redis instance for multiple unrelated applications. The default database in Redis is 0 while other logical databases need to be explicitly SELECT'ed upon connecting.

Default: 0

18.4.15 DNSTAP Logging Options

DNSTAP support, when compiled in by using `--enable-dnstap`, is enabled in the **dnstap:** section. This starts an extra thread (when compiled with threading) that writes the log information to the destination. If Unbound is compiled without threading it does not spawn a thread, but connects per-process to the destination.

dnstap-enable: *<yes or no>*
: If dnstap is enabled. If yes, it connects to the DNSTAP server and if any of the *dnstap-log-..-messages:* options is enabled it sends logs for those messages to the server.

  Default: no

dnstap-bidirectional: *<yes or no>*
: Use frame streams in bidirectional mode to transfer DNSTAP messages.

  Default: yes

dnstap-socket-path: *<file name>*
: Sets the unix socket file name for connecting to the server that is listening on that socket.

  Default: “”

dnstap-ip: *<IPaddress[@port]>*
: If "", the unix socket is used, if set with an IP address (IPv4 or IPv6) that address is used to connect to the server.

  Default: “”

dnstap-tls: *<yes or no>*
: Set this to use TLS to connect to the server specified in *dnstap-ip:*. If set to no, TCP is used to connect to the server.

  Default: yes

dnstap-tls-server-name: *<name of TLS authentication>*
: The TLS server name to authenticate the server with. Used when *dnstap-tls: yes*. If "" it is ignored.

  Default: “”

dnstap-tls-cert-bundle: *<file name of cert bundle>*
: The pem file with certs to verify the TLS server certificate. If "" the server default cert bundle is used, or the windows cert bundle on windows.

  Default: “”

dnstap-tls-client-key-file: *<file name>*
: The client key file for TLS client authentication. If "" client authentication is not used.

  Default: “”

dnstap-tls-client-cert-file: *<file name>*
: The client cert file for TLS client authentication.

  Default: “”

dnstap-send-identity: *<yes or no>*
: If enabled, the server identity is included in the log messages.

  Default: no

dnstap-send-version: *<yes or no>*
: If enabled, the server version if included in the log messages.

  Default: no

dnstap-identity: *<string>*
The identity to send with messages, if "" the hostname is used.

Default: “”

dnstap-version: *<string>*
The version to send with messages, if "" the package version is used.

Default: “”

dnstap-log-resolver-query-messages: *<yes or no>*
Enable to log resolver query messages. These are messages from Unbound to upstream servers.

Default: no

dnstap-log-resolver-response-messages: *<yes or no>*
Enable to log resolver response messages. These are replies from upstream servers to Unbound.

Default: no

dnstap-log-client-query-messages: *<yes or no>*
Enable to log client query messages. These are client queries to Unbound.

Default: no

dnstap-log-client-response-messages: *<yes or no>*
Enable to log client response messages. These are responses from Unbound to clients.

Default: no

dnstap-log-forwarder-query-messages: *<yes or no>*
Enable to log forwarder query messages.

Default: no

dnstap-log-forwarder-response-messages: *<yes or no>*
Enable to log forwarder response messages.

Default: no

18.4.16 Response Policy Zone Options

Response Policy Zones are configured with **rpz:**, and each one must have a *name:*. There can be multiple ones, by listing multiple rpz clauses, each with a different name. RPZ clauses are applied in order of configuration. The respip module needs to be added to the *module-config*, e.g.:

```
module-config: "respip validator iterator"
```

QNAME, Response IP Address, nsdname, nsip and clientip triggers are supported. Supported actions are: NXDOMAIN, NODATA, PASSTHRU, DROP, Local Data, tcp-only and drop. RPZ QNAME triggers are applied after any *local-zone:* and before any *auth-zone:*.

The RPZ zone is formatted with a SOA start record as usual. The items in the zone are entries, that specify what to act on (the trigger) and what to do (the action). The trigger to act on is recorded in the name, the action to do is recorded as the resource record. The names all end in the zone name, so you could type the trigger names without a trailing dot in the zonefile.

An example RPZ record, that answers `example.com` with `NXDOMAIN`:

```
example.com CNAME .
```

The triggers are encoded in the name on the left

```
name                          query name
netblock.rpz-client-ip        client IP address
netblock.rpz-ip               response IP address in the answer
name.rpz-nsdname              nameserver name
netblock.rpz-nsip             nameserver IP address
```

The netblock is written as <netblocklen>.<ip address in reverse>. For IPv6 use 'zz' for '::'. Specify individual addresses with scope length of 32 or 128. For example, 24.10.100.51.198.rpz-ip is 198.51.100.10/24 and 32.10.zz.db8.2001.rpz-ip is 2001:db8:0:0:0:0:0:10/32.

The actions are specified with the record on the right

```
CNAME .                       nxdomain reply
CNAME *.                      nodata reply
CNAME rpz-passthru.           do nothing, allow to continue
CNAME rpz-drop.               the query is dropped
CNAME rpz-tcp-only.           answer over TCP
A 192.0.2.1                   answer with this IP address
```

Other records like AAAA, TXT and other CNAMEs (not rpz-..) can also be used to answer queries with that content.

The RPZ zones can be configured in the config file with these settings in the **rpz:** block.

name: ***<zone name>***
Name of the authority zone.

primary: ***<IP address or host name>***
Where to download a copy of the zone from, with AXFR and IXFR. Multiple primaries can be specified. They are all tried if one fails.

To use a non-default port for DNS communication append '@' with the port number.

You can append a '#' and a name, then AXFR over TLS can be used and the TLS authentication certificates will be checked with that name.

If you combine the '@' and '#', the '@' comes first. If you point it at another Unbound instance, it would not work because that does not support AXFR/IXFR for the zone, but if you used *url:* to download the zonefile as a text file from a webserver that would work.

If you specify the hostname, you cannot use the domain from the zonefile, because it may not have that when retrieving that data, instead use a plain IP address to avoid a circular dependency on retrieving that IP address.

master: ***<IP address or host name>***
Alternate syntax for *primary:*.

url: ***<url to zonefile>***
Where to download a zonefile for the zone. With HTTP or HTTPS. An example for the url is:

```
http://www.example.com/example.org.zone
```

Multiple url statements can be given, they are tried in turn.

If only urls are given the SOA refresh timer is used to wait for making new downloads. If also primaries are listed, the primaries are first probed with UDP SOA queries to see if the SOA serial number has changed, reducing the number of downloads. If none of the URLs work, the primaries are tried with IXFR and AXFR.

For HTTPS, the *tls-cert-bundle:* and the hostname from the url are used to authenticate the connection.

allow-notify: ***<IP address or host name or netblockIP / prefix>***
With *allow-notify:* you can specify additional sources of notifies. When notified, the server attempts to first probe

and then zone transfer. If the notify is from a primary, it first attempts that primary. Otherwise other primaries are attempted. If there are no primaries, but only urls, the file is downloaded when notified.

Note: The primaries from *primary:* and *url:* statements are allowed notify by default.

zonefile: <*filename*>
The filename where the zone is stored. If not given then no zonefile is used. If the file does not exist or is empty, Unbound will attempt to fetch zone data (eg. from the primary servers).

rpz-action-override: <*action*>
Always use this RPZ action for matching triggers from this zone. Possible actions are: *nxdomain*, *nodata*, *passthru*, *drop*, *disabled* and *cname*.

rpz-cname-override: <*domain*>
The CNAME target domain to use if the cname action is configured for *rpz-action-override:*.

rpz-log: <*yes or no*>
Log all applied RPZ actions for this RPZ zone.

Default: no

rpz-log-name: <*name*>
Specify a string to be part of the log line, for easy referencing.

rpz-signal-nxdomain-ra: <*yes or no*>
Signal when a query is blocked by the RPZ with NXDOMAIN with an unset RA flag. This allows certain clients, like dnsmasq, to infer that the domain is externally blocked.

Default: no

for-downstream: <*yes or no*>
If enabled the zone is authoritatively answered for and queries for the RPZ zone information are answered to downstream clients. This is useful for monitoring scripts, that can then access the SOA information to check if the RPZ information is up to date.

Default: no

tags: *"<list of tags>"*
Limit the policies from this RPZ clause to clients with a matching tag.

Tags need to be defined in *define-tag:* and can be assigned to client addresses using *access-control-tag:* or *interface-tag:*. Enclose list of tags in quotes ("") and put spaces between tags.

If no tags are specified the policies from this clause will be applied for all clients.

18.5 Memory Control Example

In the example config settings below memory usage is reduced. Some service levels are lower, notable very large data and a high TCP load are no longer supported. Very large data and high TCP loads are exceptional for the DNS. DNSSEC validation is enabled, just add trust anchors. If you do not have to worry about programs using more than 3 Mb of memory, the below example is not for you. Use the defaults to receive full service, which on BSD-32bit tops out at 30-40 Mb after heavy usage.

```
# example settings that reduce memory usage
server:
num-threads: 1
outgoing-num-tcp: 1 # this limits TCP service, uses less buffers.
```

(continues on next page)

(continued from previous page)

```
incoming-num-tcp: 1
outgoing-range: 60  # uses less memory, but less performance.
msg-buffer-size: 8192   # note this limits service, 'no huge stuff'.
msg-cache-size: 100k
msg-cache-slabs: 1
rrset-cache-size: 100k
rrset-cache-slabs: 1
infra-cache-numhosts: 200
infra-cache-slabs: 1
key-cache-size: 100k
key-cache-slabs: 1
neg-cache-size: 10k
num-queries-per-thread: 30
target-fetch-policy: "2 1 0 0 0 0"
harden-large-queries: "yes"
harden-short-bufsize: "yes"
```

18.6 Files

/usr/local/etc/unbound
: default Unbound working directory.

/usr/local/etc/unbound
: default *chroot(2)* location.

/usr/local/etc/unbound/unbound.conf
: Unbound configuration file.

/usr/local/etc/unbound/unbound.pid
: default Unbound pidfile with process ID of the running daemon.

unbound.log
: Unbound log file. Default is to log to *syslog(3)*.

18.7 See Also

unbound(8), *unbound-checkconf(8)*.

CHAPTER

NINETEEN

UNBOUND-HOST(1)

19.1 Synopsis

unbound-host [-C configfile] [-vdhr46D] [-c class] [-t type] [-y key] [-f keyfile] [-F namedkeyfile] hostname

19.2 Description

unbound-host uses the Unbound validating resolver to query for the hostname and display results. With the *-v* option it displays validation status: secure, insecure, bogus (security failure).

By default it reads no configuration file whatsoever. It attempts to reach the internet root servers. With *-C* an unbound config file and with *-r* resolv.conf can be read.

The available options are:

hostname
: This name is resolved (looked up in the DNS). If a IPv4 or IPv6 address is given, a reverse lookup is performed.

-h
: Show the version and commandline option help.

-v
: Enable verbose output and it shows validation results, on every line. Secure means that the NXDOMAIN (no such domain name), nodata (no such data) or positive data response validated correctly with one of the keys. Insecure means that that domain name has no security set up for it. Bogus (security failure) means that the response failed one or more checks, it is likely wrong, outdated, tampered with, or broken.

-d
: Enable debug output to stderr. One *-d* shows what the resolver and validator are doing and may tell you what is going on. More times, *-d -d*, gives a lot of output, with every packet sent and received.

-c <class>
: Specify the class to lookup for, the default is IN the internet class.

-t <type>
: Specify the type of data to lookup. The default looks for IPv4, IPv6 and mail handler data, or domain name pointers for reverse queries.

-y <key>
: Specify a public key to use as trust anchor. This is the base for a chain of trust that is built up from the trust anchor to the response, in order to validate the response message. Can be given as a DS or DNSKEY record. For example:

```
-y "example.com DS 31560 5 1 1CFED84787E6E19CCF9372C1187325972FE546CD"
```

-D

Enables DNSSEC validation. Reads the root anchor from the default configured root anchor at the default location, `/usr/local/etc/unbound/root.key`.

-f <keyfile>

Reads keys from a file. Every line has a DS or DNSKEY record, in the format as for *-y*. The zone file format, the same as `dig` and `drill` produce.

-F <namedkeyfile>

Reads keys from a BIND-style `named.conf` file. Only the `trusted-key {};` entries are read.

-C <configfile>

Uses the specified unbound.conf to prime *libunbound(3)*. Pass it as first argument if you want to override some options from the config file with further arguments on the commandline.

-r

Read `/etc/resolv.conf`, and use the forward DNS servers from there (those could have been set by DHCP). More info in *resolv.conf(5)*. Breaks validation if those servers do not support DNSSEC.

-4

Use solely the IPv4 network for sending packets.

-6

Use solely the IPv6 network for sending packets.

19.3 Examples

Some examples of use. The keys shown below are fakes, thus a security failure is encountered.

```
$ unbound-host www.example.com

$ unbound-host -v -y "example.com DS 31560 5 1 1CFED84787E6E19CCF9372C1187325972FE546CD"␣
↪www.example.com

$ unbound-host -v -y "example.com DS 31560 5 1 1CFED84787E6E19CCF9372C1187325972FE546CD"␣
↪192.0.2.153
```

19.4 Exit Code

The `unbound-host` program exits with status code 1 on error, 0 on no error. The data may not be available on exit code 0, exit code 1 means the lookup encountered a fatal error.

19.5 See Also

unbound.conf(5), *unbound(8)*.

CHAPTER

TWENTY

LIBUNBOUND(3)

20.1 Synopsis

20.2 Description

Unbound is an implementation of a DNS resolver, that does caching and DNSSEC validation. This is the library API, for using the `-lunbound` library. The server daemon is described in *unbound(8)*. The library works independent from a running unbound server, and can be used to convert hostnames to ip addresses, and back, and obtain other information from the DNS. The library performs public-key validation of results with DNSSEC.

The library uses a variable of type *struct ub_ctx* to keep context between calls. The user must maintain it, creating it with **ub_ctx_create** and deleting it with **ub_ctx_delete**. It can be created and deleted at any time. Creating it anew removes any previous configuration (such as trusted keys) and clears any cached results.

The functions are thread-safe, and a context can be used in a threaded (as well as in a non-threaded) environment. Also resolution (and validation) can be performed blocking and non-blocking (also called asynchronous). The async method returns from the call immediately, so that processing can go on, while the results become available later.

The functions are discussed in turn below.

20.3 Functions

ub_ctx_create
: Create a new context, initialised with defaults. The information from `/etc/resolv.conf` and `/etc/hosts` is not utilised by default. Use **ub_ctx_resolvconf** and **ub_ctx_hosts** to read them. Before you call this, use the openssl functions **CRYPTO_set_id_callback** and **CRYPTO_set_locking_callback** to set up asynchronous operation if you use lib openssl (the application calls these functions once for initialisation). Openssl 1.0.0 or later uses the **CRYPTO_THREADID_set_callback** function.

ub_ctx_delete
: Delete validation context and free associated resources. Outstanding async queries are killed and callbacks are not called for them.

ub_ctx_set_option
: A power-user interface that lets you specify one of the options from the config file format, see *unbound.conf(5)*. Not all options are relevant. For some specific options, such as adding trust anchors, special routines exist. Pass the option name with the trailing `':'`.

ub_ctx_get_option
: A power-user interface that gets an option value. Some options cannot be gotten, and others return a newline separated list. Pass the option name without trailing `':'`. The returned value must be free(2)d by the caller.

ub_ctx_config
A power-user interface that lets you specify an unbound config file, see *unbound.conf(5)*, which is read for configuration. Not all options are relevant. For some specific options, such as adding trust anchors, special routines exist. This function is thread-safe only if a single instance of **ub_ctx*** exists in the application. If several instances exist the application has to ensure that **ub_ctx_config** is not called in parallel by the different instances.

ub_ctx_set_fwd
Set machine to forward DNS queries to, the caching resolver to use. IP4 or IP6 address. Forwards all DNS requests to that machine, which is expected to run a recursive resolver. If the proxy is not DNSSEC capable, validation may fail. Can be called several times, in that case the addresses are used as backup servers. At this time it is only possible to set configuration before the first resolve is done.

ub_ctx_set_stub
Set a stub zone, authoritative dns servers to use for a particular zone. IP4 or IP6 address. If the address is NULL the stub entry is removed. Set isprime true if you configure root hints with it. Otherwise similar to the stub zone item from unbound's config file. Can be called several times, for different zones, or to add multiple addresses for a particular zone. At this time it is only possible to set configuration before the first resolve is done.

ub_ctx_set_tls
Enable DNS over TLS (DoT) for machines set with **ub_ctx_set_fwd**. At this time it is only possible to set configuration before the first resolve is done.

ub_ctx_resolvconf
By default the root servers are queried and full resolver mode is used, but you can use this call to read the list of nameservers to use from the filename given. Usually `"/etc/resolv.conf"`. Uses those nameservers as caching proxies. If they do not support DNSSEC, validation may fail. Only nameservers are picked up, the searchdomain, ndots and other settings from *resolv.conf(5)* are ignored. If fname NULL is passed, `"/etc/resolv.conf"` is used (if on Windows, the system-wide configured nameserver is picked instead). At this time it is only possible to set configuration before the first resolve is done.

ub_ctx_hosts
Read list of hosts from the filename given. Usually `"/etc/hosts"`. When queried for, these addresses are not marked DNSSEC secure. If fname NULL is passed, `"/etc/hosts"` is used (if on Windows, `etc/hosts` from WINDIR is picked instead). At this time it is only possible to set configuration before the first resolve is done.

ub_ctx_add_ta
Add a trust anchor to the given context. At this time it is only possible to add trusted keys before the first resolve is done. The format is a string, similar to the zone-file format, **[domainname] [type] [rdata contents]**. Both DS and DNSKEY records are accepted.

ub_ctx_add_ta_autr
Add filename with automatically tracked trust anchor to the given context. Pass name of a file with the managed trust anchor. You can create this file with *unbound-anchor(8)* for the root anchor. You can also create it with an initial file with one line with a DNSKEY or DS record. If the file is writable, it is updated when the trust anchor changes. At this time it is only possible to add trusted keys before the first resolve is done.

ub_ctx_add_ta_file
Add trust anchors to the given context. Pass name of a file with DS and DNSKEY records in zone file format. At this time it is only possible to add trusted keys before the first resolve is done.

ub_ctx_trustedkeys
Add trust anchors to the given context. Pass the name of a bind-style config file with `trusted-keys{}`. At this time it is only possible to add trusted keys before the first resolve is done.

ub_ctx_debugout
Set debug and error log output to the given stream. Pass NULL to disable output. Default is stderr. File-names or using syslog can be enabled using config options, this routine is for using your own stream.

ub_ctx_debuglevel
Set debug verbosity for the context. Output is directed to stderr. Higher debug level gives more output.

ub_ctx_async
Set a context behaviour for asynchronous action. if set to true, enables threading and a call to **ub_resolve_async** creates a thread to handle work in the background. If false, a process is forked to handle work in the background. Changes to this setting after **ub_resolve_async** calls have been made have no effect (delete and re-create the context to change).

ub_poll
Poll a context to see if it has any new results. Do not poll in a loop, instead extract the **fd** below to poll for readiness, and then check, or wait using the wait routine. Returns 0 if nothing to read, or nonzero if a result is available. If nonzero, call **ub_process** to do callbacks.

ub_wait
Wait for a context to finish with results. Calls **ub_process** after the wait for you. After the wait, there are no more outstanding asynchronous queries.

ub_fd
Get file descriptor. Wait for it to become readable, at this point answers are returned from the asynchronous validating resolver. Then call the **ub_process** to continue processing.

ub_process
Call this routine to continue processing results from the validating resolver (when the **fd** becomes readable). Will perform necessary callbacks.

ub_resolve
Perform resolution and validation of the target name. The name is a domain name in a zero terminated text string. The rrtype and rrclass are DNS type and class codes. The result structure is newly allocated with the resulting data.

ub_resolve_async
Perform asynchronous resolution and validation of the target name. Arguments mean the same as for **ub_resolve** except no data is returned immediately, instead a callback is called later. The callback receives a copy of the mydata pointer, that you can use to pass information to the callback. The callback type is a function pointer to a function declared as:

```
void my_callback_function(void* my_arg, int err,
                struct ub_result* result);
```

The **async_id** is returned so you can (at your option) decide to track it and cancel the request if needed. If you pass a NULL pointer the **async_id** is not returned.

ub_cancel
Cancel an async query in progress. This may return an error if the query does not exist, or the query is already being delivered, in that case you may still get a callback for the query.

ub_resolve_free
Free struct **ub_result** contents after use.

ub_strerror
Convert error value from one of the unbound library functions to a human readable string.

ub_ctx_print_local_zones
Debug printout the local authority information to debug output.

ub_ctx_zone_add
Add new zone to local authority info, like local-zone *unbound.conf(5)* statement.

ub_ctx_zone_remove
Delete zone from local authority info.

ub_ctx_data_add
Add resource record data to local authority info, like local-data *unbound.conf(5)* statement.

ub_ctx_data_remove
Delete local authority data from the name given.

20.4 Result Data structure

The result of the DNS resolution and validation is returned as *struct ub_result*. The result structure contains the following entries:

```
struct ub_result {
     char* qname;          /* text string, original question */
     int qtype;            /* type code asked for */
     int qclass;           /* class code asked for */
     char** data;          /* array of rdata items, NULL terminated*/
     int* len;             /* array with lengths of rdata items */
     char* canonname;      /* canonical name of result */
     int rcode;            /* additional error code in case of no data */
     void* answer_packet; /* full network format answer packet */
     int answer_len;       /* length of packet in octets */
     int havedata;         /* true if there is data */
     int nxdomain;         /* true if nodata because name does not exist */
     int secure;           /* true if result is secure */
     int bogus;            /* true if a security failure happened */
     char* why_bogus;      /* string with error if bogus */
     int was_ratelimited; /* true if the query was ratelimited (SERVFAIL) by unbound */
     int ttl;              /* number of seconds the result is valid */
};
```

If both secure and bogus are false, security was not enabled for the domain of the query. Else, they are not both true, one of them is true.

20.5 Return Values

Many routines return an error code. The value 0 (zero) denotes no error happened. Other values can be passed to **ub_strerror** to obtain a readable error string. **ub_strerror** returns a zero terminated string. **ub_ctx_create** returns NULL on an error (a malloc failure). **ub_poll** returns true if some information may be available, false otherwise. **ub_fd** returns a file descriptor or -1 on error. **ub_ctx_config** and **ub_ctx_resolvconf** attempt to leave errno informative on a function return with file read failure.

20.6 See Also

unbound.conf(5), *unbound(8)*.

CHAPTER

TWENTYONE

UNBOUND-CONTROL(8)

21.1 Synopsis

unbound-control [-hq] [-c cfgfile] [-s server] command

21.2 Description

unbound-control performs remote administration on the *unbound(8)* DNS server. It reads the configuration file, contacts the Unbound server over TLS sends the command and displays the result.

The available options are:

-h
: Show the version and commandline option help.

-c <cfgfile>
: The config file to read with settings. If not given the default config file /usr/local/etc/unbound/unbound.conf is used.

-s <server[@port]>
: IPv4 or IPv6 address of the server to contact. If not given, the address is read from the config file.

-q
: Quiet, if the option is given it does not print anything if it works ok.

21.3 Commands

There are several commands that the server understands.

start
: Start the server. Simply execs *unbound(8)*. The unbound executable is searched for in the **PATH** set in the environment. It is started with the config file specified using *-c* or the default config file.

stop
: Stop the server. The server daemon exits.

reload
: Reload the server. This flushes the cache and reads the config file fresh.

reload_keep_cache
: Reload the server but try to keep the RRset and message cache if (re)configuration allows for it. That means the caches sizes and the number of threads must not change between reloads.

verbosity *number*
Change verbosity value for logging. Same values as the **verbosity:** keyword in *unbound.conf(5)*. This new setting lasts until the server is issued a reload (taken from config file again), or the next verbosity control command.

log_reopen
Reopen the logfile, close and open it. Useful for logrotation to make the daemon release the file it is logging to. If you are using syslog it will attempt to close and open the syslog (which may not work if chrooted).

stats
Print statistics. Resets the internal counters to zero, this can be controlled using the **statistics-cumulative:** config statement. Statistics are printed with one `[name]: [value]` per line.

stats_noreset
Peek at statistics. Prints them like the stats command does, but does not reset the internal counters to zero.

status
Display server status. Exit code 3 if not running (the connection to the port is refused), 1 on error, 0 if running.

local_zone *name type*
Add new local zone with name and type. Like local-zone config statement. If the zone already exists, the type is changed to the given argument.

local_zone_remove *name*
Remove the local zone with the given name. Removes all local data inside it. If the zone does not exist, the command succeeds.

local_data *RR data...*
Add new local data, the given resource record. Like **local-data:** keyword, except for when no covering zone exists. In that case this remote control command creates a transparent zone with the same name as this record.

local_data_remove *name*
Remove all RR data from local name. If the name already has no items, nothing happens. Often results in NXDOMAIN for the name (in a static zone), but if the name has become an empty nonterminal (there is still data in domain names below the removed name), NOERROR nodata answers are the result for that name.

local_zones
Add local zones read from stdin of unbound-control. Input is read per line, with name space type on a line. For bulk additions.

local_zones_remove
Remove local zones read from stdin of unbound-control. Input is one name per line. For bulk removals.

local_datas
Add local data RRs read from stdin of unbound-control. Input is one RR per line. For bulk additions.

local_datas_remove
Remove local data RRs read from stdin of unbound-control. Input is one name per line. For bulk removals.

dump_cache
The contents of the cache is printed in a text format to stdout. You can redirect it to a file to store the cache in a file.

load_cache
The contents of the cache is loaded from stdin. Uses the same format as dump_cache uses. Loading the cache with old, or wrong data can result in old or wrong data returned to clients. Loading data into the cache in this way is supported in order to aid with debugging.

lookup *name*
Print to stdout the name servers that would be used to look up the name specified.

flush *name*
Remove the name from the cache. Removes the types A, AAAA, NS, SOA, CNAME, DNAME, MX, PTR, SRV,

NAPTR, SVCB and HTTPS. Because that is fast to do. Other record types can be removed using **flush_type** or **flush_zone**.

flush_type *name type*
Remove the name, type information from the cache.

flush_zone name
Remove all information at or below the name from the cache. The rrsets and key entries are removed so that new lookups will be performed. This needs to walk and inspect the entire cache, and is a slow operation. The entries are set to expired in the implementation of this command (so, with serve-expired enabled, it'll serve that information but schedule a prefetch for new information).

flush_bogus
Remove all bogus data from the cache.

flush_negative
Remove all negative data from the cache. This is nxdomain answers, nodata answers and servfail answers. Also removes bad key entries (which could be due to failed lookups) from the dnssec key cache, and iterator last-resort lookup failures from the rrset cache.

flush_stats
Reset statistics to zero.

flush_requestlist
Drop the queries that are worked on. Stops working on the queries that the server is working on now. The cache is unaffected. No reply is sent for those queries, probably making those users request again later. Useful to make the server restart working on queries with new settings, such as a higher verbosity level.

dump_requestlist
Show what is worked on. Prints all queries that the server is currently working on. Prints the time that users have been waiting. For internal requests, no time is printed. And then prints out the module status. This prints the queries from the first thread, and not queries that are being serviced from other threads.

flush_infra *all|IP*
If all then entire infra cache is emptied. If a specific IP address, the entry for that address is removed from the cache. It contains EDNS, ping and lameness data.

dump_infra
Show the contents of the infra cache.

set_option *opt: val*
Set the option to the given value without a reload. The cache is therefore not flushed. The option must end with a `':'` and whitespace must be between the option and the value. Some values may not have an effect if set this way, the new values are not written to the config file, not all options are supported. This is different from the set_option call in libunbound, where all values work because Unbound has not been initialized.

The values that work are: statistics-interval, statistics-cumulative, do-not-query-localhost, harden-short-bufsize, harden-large-queries, harden-glue, harden-dnssec-stripped, harden-below-nxdomain, harden-referral-path, prefetch, prefetch-key, log-queries, hide-identity, hide-version, identity, version, val-log-level, val-log-squelch, ignore-cd-flag, add-holddown, del-holddown, keep-missing, tcp-upstream, ssl-upstream, max-udp-size, ratelimit, ip-ratelimit, cache-max-ttl, cache-min-ttl, cache-max-negative-ttl.

get_option *opt*
Get the value of the option. Give the option name without a trailing `':'`. The value is printed. If the value is `""`, nothing is printed and the connection closes. On error `'error ...'` is printed (it gives a syntax error on unknown option). For some options a list of values, one on each line, is printed. The options are shown from the config file as modified with set_option. For some options an override may have been taken that does not show up with this command, not results from e.g. the verbosity and forward control commands. Not all options work, see list_stubs, list_forwards, list_local_zones and list_local_data for those.

list_stubs
List the stub zones in use. These are printed one by one to the output. This includes the root hints in use.

list_forwards
List the forward zones in use. These are printed zone by zone to the output.

list_insecure
List the zones with domain-insecure.

list_local_zones
List the local zones in use. These are printed one per line with zone type.

list_local_data
List the local data RRs in use. The resource records are printed.

insecure_add *zone*
Add a domain-insecure for the given zone, like the statement in unbound.conf. Adds to the running Unbound without affecting the cache contents (which may still be bogus, use flush_zone to remove it), does not affect the config file.

insecure_remove *zone*
Removes domain-insecure for the given zone.

forward_add [+i] *zone addr ...*
Add a new forward zone to running Unbound. With `+i` option also adds a domain-insecure for the zone (so it can resolve insecurely if you have a DNSSEC root trust anchor configured for other names). The addr can be IP4, IP6 or nameserver names, like forward-zone config in unbound.conf.

forward_remove [+i] *zone*
Remove a forward zone from running Unbound. The `+i` also removes a domain-insecure for the zone.

stub_add [+ip] *zone addr ...*
Add a new stub zone to running Unbound. With `+i` option also adds a domain-insecure for the zone. With `+p` the stub zone is set to prime, without it it is set to notprime. The addr can be IP4, IP6 or nameserver names, like the **stub-zone:** config in unbound.conf.

stub_remove [+i] *zone*
Remove a stub zone from running Unbound. The `+i` also removes a domain-insecure for the zone.

forward [*off* **|** *addr ...* **]**
Setup forwarding mode. Configures if the server should ask other upstream nameservers, should go to the internet root nameservers itself, or show the current config. You could pass the nameservers after a DHCP update.

Without arguments the current list of addresses used to forward all queries to is printed. On startup this is from the forward-zone `"."` configuration. Afterwards it shows the status. It prints off when no forwarding is used.

If off is passed, forwarding is disabled and the root nameservers are used. This can be used to avoid to avoid buggy or non-DNSSEC supporting nameservers returned from DHCP. But may not work in hotels or hotspots.

If one or more IPv4 or IPv6 addresses are given, those are then used to forward queries to. The addresses must be separated with spaces. With `'@port'` the port number can be set explicitly (default port is 53 (DNS)).

By default the forwarder information from the config file for the root `"."` is used. The config file is not changed, so after a reload these changes are gone. Other forward zones from the config file are not affected by this command.

ratelimit_list [+a]
List the domains that are ratelimited. Printed one per line with current estimated qps and qps limit from config. With `+a` it prints all domains, not just the ratelimited domains, with their estimated qps. The ratelimited domains return an error for uncached (new) queries, but cached queries work as normal.

ip_ratelimit_list [+a]
List the ip addresses that are ratelimited. Printed one per line with current estimated qps and qps limit from

config. With +a it prints all ips, not just the ratelimited ips, with their estimated qps. The ratelimited ips are dropped before checking the cache.

list_auth_zones
List the auth zones that are configured. Printed one per line with a status, indicating if the zone is expired and current serial number. Configured RPZ zones are included.

auth_zone_reload ***zone***
Reload the auth zone (or RPZ zone) from zonefile. The zonefile is read in overwriting the current contents of the zone in memory. This changes the auth zone contents itself, not the cache contents. Such cache contents exists if you set Unbound to validate with **for-upstream: yes** and that can be cleared with **flush_zone** *zone*.

auth_zone_transfer ***zone***
Transfer the auth zone (or RPZ zone) from master. The auth zone probe sequence is started, where the masters are probed to see if they have an updated zone (with the SOA serial check). And then the zone is transferred for a newer zone version.

rpz_enable ***zone***
Enable the RPZ zone if it had previously been disabled.

rpz_disable ***zone***
Disable the RPZ zone.

view_list_local_zones ***view***
list_local_zones for given view.

view_local_zone ***view name type***
local_zone for given view.

view_local_zone_remove ***view name***
local_zone_remove for given view.

view_list_local_data ***view***
list_local_data for given view.

view_local_data ***view RR data...***
local_data for given view.

view_local_data_remove ***view name***
local_data_remove for given view.

view_local_datas_remove ***view***
Remove a list of *local_data* for given view from stdin. Like *local_datas_remove*.

view_local_datas ***view***
Add a list of *local_data* for given view from stdin. Like *local_datas*.

21.4 Exit Code

The `unbound-control` program exits with status code 1 on error, 0 on success.

21.5 Set Up

The setup requires a self-signed certificate and private keys for both the server and client. The script `unbound-control-setup` generates these in the default run directory, or with `-d` in another directory. If you change the access control permissions on the key files you can decide who can use `unbound-control`, by default owner and group but not all users. Run the script under the same username as you have configured in `unbound.conf` or as root, so that the daemon is permitted to read the files, for example with:

```
sudo -u unbound unbound-control-setup
```

If you have not configured a username in `unbound.conf`, the keys need read permission for the user credentials under which the daemon is started. The script preserves private keys present in the directory. After running the script as root, turn on *control-enable:* in `unbound.conf`.

21.6 Statistic Counters

The *stats* and *stats_noreset* commands show a number of statistic counters:

threadX.num.queries
number of queries received by thread

threadX.num.queries_ip_ratelimited
number of queries rate limited by thread

threadX.num.queries_cookie_valid
number of queries with a valid DNS Cookie by thread

threadX.num.queries_cookie_client
number of queries with a client part only DNS Cookie by thread

threadX.num.queries_cookie_invalid
number of queries with an invalid DNS Cookie by thread

threadX.num.cachehits
number of queries that were successfully answered using a cache lookup

threadX.num.cachemiss
number of queries that needed recursive processing

threadX.num.dnscrypt.crypted
number of queries that were encrypted and successfully decapsulated by dnscrypt.

threadX.num.dnscrypt.cert
number of queries that were requesting dnscrypt certificates.

threadX.num.dnscrypt.cleartext
number of queries received on dnscrypt port that were cleartext and not a request for certificates.

threadX.num.dnscrypt.malformed
number of request that were neither cleartext, not valid dnscrypt messages.

threadX.num.prefetch
number of cache prefetches performed. This number is included in cachehits, as the original query had the unprefetched answer from cache, and resulted in recursive processing, taking a slot in the requestlist. Not part of the recursivereplies (or the histogram thereof) or cachemiss, as a cache response was sent.

threadX.num.expired
number of replies that served an expired cache entry.

threadX.num.queries_timed_out
number of queries that are dropped because they waited in the UDP socket buffer for too long.

threadX.query.queue_time_us.max
The maximum wait time for packets in the socket buffer, in microseconds. This is only reported when *sock-queue-timeout:* is enabled.

threadX.num.recursivereplies
The number of replies sent to queries that needed recursive processing. Could be smaller than threadX.num.cachemiss if due to timeouts no replies were sent for some queries.

threadX.requestlist.avg
The average number of requests in the internal recursive processing request list on insert of a new incoming recursive processing query.

threadX.requestlist.max
Maximum size attained by the internal recursive processing request list.

threadX.requestlist.overwritten
Number of requests in the request list that were overwritten by newer entries. This happens if there is a flood of queries that recursive processing and the server has a hard time.

threadX.requestlist.exceeded
Queries that were dropped because the request list was full. This happens if a flood of queries need recursive processing, and the server can not keep up.

threadX.requestlist.current.all
Current size of the request list, includes internally generated queries (such as priming queries and glue lookups).

threadX.requestlist.current.user
Current size of the request list, only the requests from client queries.

threadX.recursion.time.avg
Average time it took to answer queries that needed recursive processing. Note that queries that were answered from the cache are not in this average.

threadX.recursion.time.median
The median of the time it took to answer queries that needed recursive processing. The median means that 50% of the user queries were answered in less than this time. Because of big outliers (usually queries to non responsive servers), the average can be bigger than the median. This median has been calculated by interpolation from a histogram.

threadX.tcpusage
The currently held tcp buffers for incoming connections. A spot value on the time of the request. This helps you spot if the incoming-num-tcp buffers are full.

total.num.queries
summed over threads.

total.num.queries_ip_ratelimited
summed over threads.

total.num.queries_cookie_valid
summed over threads.

total.num.queries_cookie_client
summed over threads.

total.num.queries_cookie_invalid
summed over threads.

total.num.cachehits
summed over threads.

total.num.cachemiss
summed over threads.

total.num.dnscrypt.crypted
summed over threads.

total.num.dnscrypt.cert
summed over threads.

total.num.dnscrypt.cleartext
summed over threads.

total.num.dnscrypt.malformed
summed over threads.

total.num.prefetch
summed over threads.

total.num.expired
summed over threads.

total.num.queries_timed_out
summed over threads.

total.query.queue_time_us.max
the maximum of the thread values.

total.num.recursivereplies
summed over threads.

total.requestlist.avg
averaged over threads.

total.requestlist.max
the maximum of the thread requestlist.max values.

total.requestlist.overwritten
summed over threads.

total.requestlist.exceeded
summed over threads.

total.requestlist.current.all
summed over threads.

total.recursion.time.median
averaged over threads.

total.tcpusage
summed over threads.

time.now
current time in seconds since 1970.

time.up
uptime since server boot in seconds.

time.elapsed
time since last statistics printout, in seconds.

21.7 Extended Statistics

mem.cache.rrset
Memory in bytes in use by the RRset cache.

mem.cache.message
Memory in bytes in use by the message cache.

mem.cache.dnscrypt_shared_secret
Memory in bytes in use by the dnscrypt shared secrets cache.

mem.cache.dnscrypt_nonce
Memory in bytes in use by the dnscrypt nonce cache.

mem.mod.iterator
Memory in bytes in use by the iterator module.

mem.mod.validator
Memory in bytes in use by the validator module. Includes the key cache and negative cache.

mem.streamwait
Memory in bytes in used by the TCP and TLS stream wait buffers. These are answers waiting to be written back to the clients.

mem.http.query_buffer
Memory in bytes used by the HTTP/2 query buffers. Containing (partial) DNS queries waiting for request stream completion.

mem.http.response_buffer
Memory in bytes used by the HTTP/2 response buffers. Containing DNS responses waiting to be written back to the clients.

histogram.<sec>.<usec>.to.<sec>.<usec>
Shows a histogram, summed over all threads. Every element counts the recursive queries whose reply time fit between the lower and upper bound. Times larger or equal to the lowerbound, and smaller than the upper bound. There are 40 buckets, with bucket sizes doubling.

num.query.type.A
The total number of queries over all threads with query type A. Printed for the other query types as well, but only for the types for which queries were received, thus =0 entries are omitted for brevity.

num.query.type.other
Number of queries with query types 256-65535.

num.query.class.IN
The total number of queries over all threads with query class IN (internet). Also printed for other classes (such as CH (CHAOS) sometimes used for debugging), or NONE, ANY, used by dynamic update. num.query.class.other is printed for classes 256-65535.

num.query.opcode.QUERY
The total number of queries over all threads with query opcode QUERY. Also printed for other opcodes, UPDATE, ...

num.query.tcp
Number of queries that were made using TCP towards the Unbound server.

num.query.tcpout
Number of queries that the Unbound server made using TCP outgoing towards other servers.

num.query.udpout
Number of queries that the Unbound server made using UDP outgoing towards other servers.

num.query.tls
Number of queries that were made using TLS towards the Unbound server. These are also counted in num.query.tcp, because TLS uses TCP.

num.query.tls.resume
Number of TLS session resumptions, these are queries over TLS towards the Unbound server where the client negotiated a TLS session resumption key.

num.query.https
Number of queries that were made using HTTPS towards the Unbound server. These are also counted in num.query.tcp and num.query.tls, because HTTPS uses TLS and TCP.

num.query.ipv6
Number of queries that were made using IPv6 towards the Unbound server.

num.query.flags.RD
The number of queries that had the RD flag set in the header. Also printed for flags QR, AA, TC, RA, Z, AD, CD. Note that queries with flags QR, AA or TC may have been rejected because of that.

num.query.edns.present
number of queries that had an EDNS OPT record present.

num.query.edns.DO
number of queries that had an EDNS OPT record with the DO (DNSSEC OK) bit set. These queries are also included in the num.query.edns.present number.

num.query.ratelimited
The number of queries that are turned away from being send to nameserver due to ratelimiting.

num.query.dnscrypt.shared_secret.cachemiss
The number of dnscrypt queries that did not find a shared secret in the cache. This can be use to compute the shared secret hitrate.

num.query.dnscrypt.replay
The number of dnscrypt queries that found a nonce hit in the nonce cache and hence are considered a query replay.

num.answer.rcode.NXDOMAIN
The number of answers to queries, from cache or from recursion, that had the return code NXDOMAIN. Also printed for the other return codes.

num.answer.rcode.nodata
The number of answers to queries that had the pseudo return code nodata. This means the actual return code was NOERROR, but additionally, no data was carried in the answer (making what is called a NOERROR/NODATA answer). These queries are also included in the num.answer.rcode.NOERROR number. Common for AAAA lookups when an A record exists, and no AAAA.

num.answer.secure
Number of answers that were secure. The answer validated correctly. The AD bit might have been set in some of these answers, where the client signalled (with DO or AD bit in the query) that they were ready to accept the AD bit in the answer.

num.answer.bogus
Number of answers that were bogus. These answers resulted in SERVFAIL to the client because the answer failed validation.

num.rrset.bogus
The number of rrsets marked bogus by the validator. Increased for every RRset inspection that fails.

unwanted.queries
Number of queries that were refused or dropped because they failed the access control settings.

unwanted.replies
Replies that were unwanted or unsolicited. Could have been random traffic, delayed duplicates, very late answers, or could be spoofing attempts. Some low level of late answers and delayed duplicates are to be expected with the UDP protocol. Very high values could indicate a threat (spoofing).

msg.cache.count
The number of items (DNS replies) in the message cache.

rrset.cache.count
The number of RRsets in the rrset cache. This includes rrsets used by the messages in the message cache, but also delegation information.

infra.cache.count
The number of items in the infra cache. These are IP addresses with their timing and protocol support information.

key.cache.count
The number of items in the key cache. These are DNSSEC keys, one item per delegation point, and their validation status.

msg.cache.max_collisions
The maximum number of hash table collisions in the msg cache. This is the number of hashes that are identical when a new element is inserted in the hash table. If the value is very large, like hundreds, something is wrong with the performance of the hash table, hash values are incorrect or malicious.

rrset.cache.max_collisions
The maximum number of hash table collisions in the rrset cache. This is the number of hashes that are identical when a new element is inserted in the hash table. If the value is very large, like hundreds, something is wrong with the performance of the hash table, hash values are incorrect or malicious.

dnscrypt_shared_secret.cache.count
The number of items in the shared secret cache. These are precomputed shared secrets for a given client public key/server secret key pair. Shared secrets are CPU intensive and this cache allows Unbound to avoid recomputing the shared secret when multiple dnscrypt queries are sent from the same client.

dnscrypt_nonce.cache.count
The number of items in the client nonce cache. This cache is used to prevent dnscrypt queries replay. The client nonce must be unique for each client public key/server secret key pair. This cache should be able to host QPS * *replay window* interval keys to prevent replay of a query during *replay window* seconds.

num.query.authzone.up
The number of queries answered from auth-zone data, upstream queries. These queries would otherwise have been sent (with fallback enabled) to the internet, but are now answered from the auth zone.

num.query.authzone.down
The number of queries for downstream answered from auth-zone data. These queries are from downstream clients, and have had an answer from the data in the auth zone.

num.query.aggressive.NOERROR
The number of queries answered using cached NSEC records with NODATA RCODE. These queries would otherwise have been sent to the internet, but are now answered using cached data.

num.query.aggressive.NXDOMAIN
The number of queries answered using cached NSEC records with NXDOMAIN RCODE. These queries would otherwise have been sent to the internet, but are now answered using cached data.

num.query.subnet
Number of queries that got an answer that contained EDNS client subnet data.

num.query.subnet_cache
Number of queries answered from the edns client subnet cache. These are counted as cachemiss by the main counters, but hit the client subnet specific cache after getting processed by the edns client subnet module.

num.query.cachedb
Number of queries answered from the external cache of cachedb. These are counted as cachemiss by the main counters, but hit the cachedb external cache after getting processed by the cachedb module.

num.rpz.action.<rpz_action>
Number of queries answered using configured RPZ policy, per RPZ action type. Possible actions are: nxdomain, nodata, passthru, drop, tcp-only, local-data, disabled, and cname-override.

21.8 Files

/usr/local/etc/unbound/unbound.conf
Unbound configuration file.

/usr/local/etc/unbound
directory with private keys (`unbound_server.key` and `unbound_control.key`) and self-signed certificates (`unbound_server.pem` and `unbound_control.pem`).

21.9 See Also

unbound.conf(5), *unbound(8)*.

CHAPTER

TWENTYTWO

UNBOUND-ANCHOR(8)

22.1 Synopsis

unbound-anchor [opts]

22.2 Description

unbound-anchor performs setup or update of the root trust anchor for DNSSEC validation. The program fetches the trust anchor with the method from **RFC 7958** when regular **RFC 5011** update fails to bring it up to date. It can be run (as root) from the commandline, or run as part of startup scripts. Before you start the *unbound(8)* DNS server.

Suggested usage:

```
# in the init scripts.
# provide or update the root anchor (if necessary)
unbound-anchor -a "/usr/local/etc/unbound/root.key"
# Please note usage of this root anchor is at your own risk
# and under the terms of our LICENSE (see source).
#
# start validating resolver
# the unbound.conf contains:
# auto-trust-anchor-file: "/usr/local/etc/unbound/root.key"
unbound -c unbound.conf
```

This tool provides builtin default contents for the root anchor and root update certificate files.

It tests if the root anchor file works, and if not, and an update is possible, attempts to update the root anchor using the root update certificate. It performs a https fetch of root-anchors.xml and checks the results (**RFC 7958**); if all checks are successful, it updates the root anchor file. Otherwise the root anchor file is unchanged. It performs **RFC 5011** tracking if the DNSSEC information available via the DNS makes that possible.

It does not perform an update if the certificate is expired, if the network is down or other errors occur.

The available options are:

-a <file>

The root anchor key file, that is read in and written out. Default is /usr/local/etc/unbound/root.key. If the file does not exist, or is empty, a builtin root key is written to it.

-c <file>

The root update certificate file, that is read in. Default is /usr/local/etc/unbound/icannbundle.pem. If the file does not exist, or is empty, a builtin certificate is used.

-l

List the builtin root key and builtin root update certificate on stdout.

-u <name>

The server name, it connects to `https://name`. Specify without `https://` prefix. The default is "`data.iana.org`". It connects to the port specified with *-P*. You can pass an IPv4 address or IPv6 address (no brackets) if you want.

-S

Do not use SNI for the HTTPS connection. Default is to use SNI.

-b <address>

The source address to bind to for domain resolution and contacting the server on https. May be either an IPv4 address or IPv6 address (no brackets).

-x <path>

The pathname to the root-anchors.xml file on the server. (forms URL with *-u*). The default is `/root-anchors/root-anchors.xml`.

-s <path>

The pathname to the root-anchors.p7s file on the server. (forms URL with *-u*). The default is `/root-anchors/root-anchors.p7s`. This file has to be a PKCS7 signature over the xml file, using the pem file (*-c*) as trust anchor.

-n <name>

The emailAddress for the Subject of the signer's certificate from the p7s signature file. Only signatures from this name are allowed. The default is `dnssec@iana.org`. If you pass "" then the emailAddress is not checked.

-4

Use IPv4 for domain resolution and contacting the server on https. Default is to use IPv4 and IPv6 where appropriate.

-6

Use IPv6 for domain resolution and contacting the server on https. Default is to use IPv4 and IPv6 where appropriate.

-f <resolv.conf>

Use the given resolv.conf file. Not enabled by default, but you could try to pass `/etc/resolv.conf` on some systems. It contains the IP addresses of the recursive nameservers to use. However, since this tool could be used to bootstrap that very recursive nameserver, it would not be useful (since that server is not up yet, since we are bootstrapping it). It could be useful in a situation where you know an upstream cache is deployed (and running) and in captive portal situations.

-r <root.hints>

Use the given root.hints file (same syntax as the BIND and Unbound root hints file) to bootstrap domain resolution. By default a list of builtin root hints is used. unbound-anchor goes to the network itself for these roots, to resolve the server (*-u* option) and to check the root DNSKEY records. It does so, because the tool when used for bootstrapping the recursive resolver, cannot use that recursive resolver itself because it is bootstrapping that server.

-R

Allow fallback from *-f* <resolv.conf> file to direct root servers query. It allows you to prefer local resolvers, but fallback automatically to direct root query if they do not respond or do not support DNSSEC.

-v

More verbose. Once prints informational messages, multiple times may enable large debug amounts (such as full certificates or byte-dumps of downloaded files). By default it prints almost nothing. It also prints nothing

on errors by default; in that case the original root anchor file is simply left undisturbed, so that a recursive server can start right after it.

-C <unbound.conf>

Debug option to read <unbound.conf> into the resolver process used.

-P port

Set the port number to use for the https connection. The default is 443.

-F

Debug option to force update of the root anchor through downloading the xml file and verifying it with the certificate. By default it first tries to update by contacting the DNS, which uses much less bandwidth, is much faster (200 msec not 2 sec), and is nicer to the deployed infrastructure. With this option, it still attempts to do so (and may verbosely tell you), but then ignores the result and goes on to use the xml fallback method.

-h

Show the version and commandline option help.

22.3 Exit Code

This tool exits with value 1 if the root anchor was updated using the certificate or if the builtin root-anchor was used. It exits with code 0 if no update was necessary, if the update was possible with **RFC 5011** tracking, or if an error occurred.

You can check the exit value in this manner:

```
unbound-anchor -a "root.key" || logger "Please check root.key"
```

Or something more suitable for your operational environment.

22.4 Trust

The root keys and update certificate included in this tool are provided for convenience and under the terms of our license (see the LICENSE file in the source distribution or https://github.com/NLnetLabs/unbound/blob/master/LICENSE and might be stale or not suitable to your purpose.

By running *unbound-anchor -l* the keys and certificate that are configured in the code are printed for your convenience.

The built-in configuration can be overridden by providing a root-cert file and a rootkey file.

22.5 Files

/usr/local/etc/unbound/root.key

The root anchor file, updated with 5011 tracking, and read and written to. The file is created if it does not exist.

/usr/local/etc/unbound/icannbundle.pem

The trusted self-signed certificate that is used to verify the downloaded DNSSEC root trust anchor. You can update it by fetching it from https://data.iana.org/root-anchors/icannbundle.pem (and validate it). If the file does not exist or is empty, a builtin version is used.

https://data.iana.org/root-anchors/root-anchors.xml

Source for the root key information.

https://data.iana.org/root-anchors/root-anchors.p7s
Signature on the root key information.

22.6 See Also

unbound.conf(5), *unbound(8)*.

CHAPTER

TWENTYTHREE

RFC COMPLIANCE

Unbound strives to be a reference implementation for emerging standards in the Internet Engineering Task Force (IETF). The aim is to implement well-established Internet Drafts as a compile option and drafts in the final stage of open community review as an optional feature, that is disabled by default. Accepted RFCs are implemented in Unbound according to the described standard.

The following table provides an extensive overview of all the RFC standards and Internet drafts that have been implemented in Unbound.

RFC 1034	Domain Names – Concepts and Facilities
RFC 1035	Domain Names – Implementation and Specification
RFC 1101	DNS Encoding of Network Names and Other Types
RFC 1123	Requirements for Internet Hosts – Application and Support
RFC 1183	New DNS RR Definitions
RFC 1337	TIME-WAIT Assassination Hazards in TCP
RFC 1521	MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Int
RFC 1706	DNS NSAP Resource Records
RFC 1712	DNS Encoding of Geographical Location
RFC 1876	A Means for Expressing Location Information in the Domain Name System
RFC 1982	Serial Number Arithmetic
RFC 1995	Incremental Zone Transfer in DNS
RFC 1996	A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)
RFC 2163	Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM)
RFC 2181	Clarifications to the DNS Specification
RFC 2182	Selection and Operation of Secondary DNS Servers
RFC 2230	Key Exchange Delegation Record for the DNS
RFC 2253	Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names
RFC 2308	Negative Caching of DNS Queries (DNS NCACHE)
RFC 2535	Domain Name System Security Extensions
RFC 2536	DSA KEYs and SIGs in the Domain Name System (DNS)
RFC 2537	RSA/MD5 KEYs and SIGs in the Domain Name System (DNS)
RFC 2538	Storing Certificates in the Domain Name System (DNS)
RFC 2539	Storage of Diffie-Hellman Keys in the Domain Name System (DNS)
RFC 2606	Reserved Top Level DNS Names
RFC 2671	Extension Mechanisms for DNS (EDNS0)
RFC 2672	Non-Terminal DNS Name Redirection
RFC 2673	Binary Labels in the Domain Name System
RFC 2782	A DNS RR for specifying the location of services (DNS SRV)
RFC 2874	DNS Extensions to Support IPv6 Address Aggregation and Renumbering
RFC 2915	The Naming Authority Pointer (NAPTR) DNS Resource Record
RFC 2930	Secret Key Establishment for DNS (TKEY RR)

con

Table 1 – continued from previous page

RFC 3110	RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)
RFC 3123	A DNS RR Type for Lists of Address Prefixes (APL RR)
RFC 3225	Indicating Resolver Support of DNSSEC
RFC 3526	More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)
RFC 3597	Handling of Unknown DNS Resource Record (RR) Types
RFC 3779	X.509 Extensions for IP Addresses and AS Identifiers
RFC 4007	IPv6 Scoped Address Architecture
RFC 4025	A Method for Storing IPsec Keying Material in DNS
RFC 4033	DNS Security Introduction and Requirements
RFC 4034	Resource Records for the DNS Security Extensions
RFC 4035	Protocol Modifications for the DNS Security Extensions
RFC 4255	Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints
RFC 4343	Domain Name System (DNS) Case Insensitivity Clarification
RFC 4398	Storing Certificates in the Domain Name System (DNS)
RFC 4431	The DNSSEC Lookaside Validation (DLV) DNS Resource Record
RFC 4509	Use of SHA-256 in DNSSEC Delegation Signer (DS) Resource Records (RRs)
RFC 4592	The Role of Wildcards in the Domain Name System
RFC 4597	Conferencing Scenarios
RFC 4697	Observed DNS Resolution Misbehavior
RFC 4701	A DNS Resource Record (RR) for Encoding Dynamic Host Configuration Protocol (DHCP) Information (DHCID RR)
RFC 5001	DNS Name Server Identifier (NSID) Option
RFC 5011	Automated Updates of DNS Security (DNSSEC) Trust Anchors
RFC 5114	Additional Diffie-Hellman Groups for Use with IETF Standards
RFC 5155	DNS Security (DNSSEC) Hashed Authenticated Denial of Existence
RFC 5205	Host Identity Protocol (HIP) Domain Name System (DNS) Extension
RFC 5358	Preventing Use of Recursive Nameservers in Reflector Attacks
RFC 5452	Measures for Making DNS More Resilient against Forged Answers
RFC 5702	Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC
RFC 5933	Use of GOST Signature Algorithms in DNSKEY and RRSIG Resource Records for DNSSEC
RFC 6147	DNS64: DNS Extensions for Network Address Translation from IPv6 Clients to IPv4 Servers
RFC 6234	US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)
RFC 6303	Locally Served DNS Zones
RFC 6598	IANA-Reserved IPv4 Prefix for Shared Address Space
RFC 6604	xNAME RCODE and Status Bits Clarification
RFC 6605	Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC
RFC 6672	DNAME Redirection in the DNS
RFC 6698	The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA
RFC 6725	DNS Security (DNSSEC) DNSKEY Algorithm IANA Registry Updates
RFC 6742	DNS Resource Records for the Identifier-Locator Network Protocol (ILNP)
RFC 6761	Special-Use Domain Names
RFC 6840	Clarifications and Implementation Notes for DNS Security (DNSSEC)
RFC 6844	DNS Certification Authority Authorization (CAA) Resource Record
RFC 6891	Extension Mechanisms for DNS (EDNS(0))
RFC 6975	Signaling Cryptographic Algorithm Understanding in DNS Security Extensions (DNSSEC)
RFC 7043	Resource Records for EUI-48 and EUI-64 Addresses in the DNS
RFC 7344	Automating DNSSEC Delegation Trust Maintenance
RFC 7413	TCP Fast Open
RFC 7477	Child-to-Parent Synchronization in DNS
RFC 7553	The Uniform Resource Identifier (URI) DNS Resource Record
RFC 7646	Definition and Use of DNSSEC Negative Trust Anchors
RFC 7686	The ".onion" Special-Use Domain Name
RFC 7706	Decreasing Access Time to Root Servers by Running One on Loopback

con

Table 1 – continued from previous page

RFC 7830	The EDNS(0) Padding Option
RFC 7858	Specification for DNS over Transport Layer Security (TLS)
RFC 7871	Client Subnet in DNS Queries
RFC 7929	DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP
RFC 7958	DNSSEC Trust Anchor Publication for the Root Zone
RFC 8020	NXDOMAIN: There Really Is Nothing Underneath
RFC 8080	Edwards-Curve Digital Security Algorithm (EdDSA) for DNSSEC
RFC 8145	Signaling Trust Anchor Knowledge in DNS Security Extensions (DNSSEC)
RFC 8162	Using Secure DNS to Associate Certificates with Domain Names for S/MIME
RFC 8198	Aggressive Use of DNSSEC-Validated Cache
RFC 8310	Usage Profiles for DNS over TLS and DNS over DTLS
RFC 8375	Special-Use Domain 'home.arpa.'
RFC 8467	Padding Policies for Extension Mechanisms for DNS (EDNS(0))
RFC 8482	Providing Minimal-Sized Responses to DNS Queries That Have QTYPE=ANY
RFC 8484	DNS Queries over HTTPS (DoH)
RFC 8509	A Root Key Trust Anchor Sentinel for DNSSEC
RFC 8624	Algorithm Implementation Requirements and Usage Guidance for DNSSEC
RFC 8767	Serving Stale Data to Improve DNS Resiliency
RFC 8806	Running a Root Server Local to a Resolver
RFC 8914	Extended DNS Errors
RFC 8976	Message Digest for DNS Zones
RFC 9156	DNS Query Name Minimisation to Improve Privacy

CHAPTER

TWENTYFOUR

HISTORY

Unbound started out as a project to develop independent open-source DNS software and documentation. It was jointly funded by VeriSign, Inc., and the University of Southern California/Information Sciences Institute. The C implementation was based on a prototype written in Java. It was released in May 2008 with this `press release`.

This section contains several historic documents. There are also presentations about the initial Unbound design delivered at `IETF 67` and `RIPE 56`. The `Windows Vista install guide` is also available as a PDF.

24.1 Requirements for Recursive Caching Resolver

24.1.1 1. Introduction

This is the requirements document for a DNS name server and aims to document the goals and non-goals of the project. The DNS (the Domain Name System) is a global, replicated database that uses a hierarchical structure for queries.

Data in the DNS is stored in Resource Record sets (RR sets), and has a time to live (TTL). During this time the data can be cached. It is thus useful to cache data to speed up future lookups. A server that looks up data in the DNS for clients and caches previous answers to speed up processing is called a caching, recursive nameserver.

This project aims to develop such a nameserver in modular components, so that also DNSSEC (secure DNS) validation and stub-resolvers (that do not run as a server, but a linked into an application) are easily possible.

The main components are the Validator that validates the security fingerprints on data sets, the Iterator that sends queries to the hierarchical DNS servers that own the data and the Cache that stores data from previous queries. The networking and query management code then interface with the modules to perform the necessary processing.

In Section 2 the origins of the Unbound project are documented. Section 3 lists the goals, while Section 4 lists the explicit non-goals of the project. Section 5 discusses choices made during development.

24.1.2 2. History

The unbound resolver project started by Bill Manning, David Blacka, and Matt Larson (from the University of California and from Verisign), that created a Java based prototype resolver called Unbound. The basic design decisions of clean modules was executed.

The Java prototype worked very well, with contributions from Geoff Sisson and Roy Arends from Nominet. Around 2006 the idea came to create a full-fledged C implementation ready for deployed use. NLnet Labs volunteered to write this implementation.

24.1.3 3. Goals

- A validating recursive DNS resolver
- Code diversity in the DNS resolver monoculture
- Drop-in replacement for BIND apart from config
- DNSSEC support
- Fully RFC compliant
- High performance
  - Even with validation
- Used as
  - Stub resolver
  - Full caching name server
  - Resolver library
- Elegant design of validator, resolver, cache modules
  - Provide the ability to pick and choose modules
- Robust
- In C, open source: The BSD license
- Highly portable, targets include modern Unix systems, such as *BSD, Solaris, linux, and maybe also the windows platform
- Smallest as possible component that does the job
- Stub-zones can be configured (local data or AS112 zones)

24.1.4 4. Non-Goals

- An authoritative name server
- Too many Features

24.1.5 5. Choices

- rfc2181 discourages duplicates RRs in RRsets. unbound does not create duplicates, but when presented with duplicates on the wire from the authoritative servers, does not perform duplicate removal. It does do some rrsig duplicate removal, in the msgparser, for dnssec qtype rrsig and any, because of special rrsig processing in the msgparser.
- The harden-glue feature, when yes all out of zone glue is deleted, when no out of zone glue is used for further resolving, is more complicated than that, see below.

  Main points:

  - rfc2182 trust handling is used
  - data is let through only in very specific cases
  - spoofability remains possible

Not all glue is let through (despite the name of the option). Only glue which is present in a delegation, of type A and AAAA, where the name is present in the NS record in the authority section is let through. The glue that is let through is stored in the cache (marked as 'from the additional section'). And will then be used for sending queries to. It will not be present in the reply to the client (if RD is off). A direct query for that name will attempt to get a msg into the message cache. Since A and AAAA queries are not synthesized by the unbound cache, this query will be (eventually) sent to the authoritative server and its answer will be put in the cache, marked as 'from the answer section' and thus remove the 'from the additional section' data, and this record is returned to the client.

The message has a TTL smaller or equal to the TTL of the answer RR. If the cache memory is low; the answer RR may be dropped, and a glue RR may be inserted, within the message TTL time, and thus return the spoofed glue to a client. When the message expires, it is refetched and the cached RR is updated with the correct content. The server can be spoofed by getting it to visit a especially prepared domain. This domain then inserts an address for another authoritative server into the cache, when visiting that other domain, this address may then be used to send queries to. And fake answers may be returned. If the other domain is signed by DNSSEC, the fakes will be detected.

In summary, the harden glue feature presents a security risk if disabled. Disabling the feature leads to possible better performance as more glue is present for the recursive service to use. The feature is implemented so as to minimise the security risk, while trying to keep this performance gain.

- The method by which dnssec-lameness is detected is not secure. DNSSEC lame is when a server has the zone in question, but lacks dnssec data, such as signatures. The method to detect dnssec lameness looks at nonvalidated data from the parent of a zone. This can be used, by spoofing the parent, to create a false sense of dnssec-lameness in the child, or a false sense or dnssec-non-lameness in the child. The first results in the server marked lame, and not used for 900 seconds, and the second will result in a validator failure (SERVFAIL again), when the query is validated later on.

  Concluding, a spoof of the parent delegation can be used for many cases of denial of service. I.e. a completely different NS set could be returned, or the information withheld. All of these alterations can be caught by the validator if the parent is signed, and result in 900 seconds bogus. The dnssec-lameness detection is used to detect operator failures, before the validator will properly verify the messages.

  Also for zones for which no chain of trust exists, but a DS is given by the parent, dnssec-lameness detection enables. This delivers dnssec to our clients when possible (for client validators).

  The following issue needs to be resolved:

  > A server that serves both a parent and child zone, where parent is signed, but child is not. The server must not be marked lame for the parent zone, because the child answer is not signed.

  Instead of a false positive, we want false negatives; failure to detect dnssec-lameness is less of a problem than marking honest servers lame. dnssec-lameness is a config error and deserves the trouble. So, only messages that identify the zone are used to mark the zone lame. The zone is identified by SOA or NS RRsets in the answer/auth. That includes almost all negative responses and also A, AAAA qtypes. That would be most responses from servers. For referrals, delegations that add a single label can be checked to be from their zone, this covers most delegation-centric zones.

  So possibly, for complicated setups, with multiple (parent-child) zones on a server, dnssec-lameness detection does not work - no dnssec-lameness is detected. Instead the zone that is dnssec-lame becomes bogus.

- authority features

  This is a recursive server, and authority features are out of scope. However, some authority features are expected in a recursor. Things like localhost, reverse lookup for 127.0.0.1, or blocking AS112 traffic. Also redirection of domain names with fixed data is needed by service providers. Limited support is added specifically to address this.

  Adding full authority support, requires much more code, and more complex maintenance.

The limited support allows adding some static data (for localhost and so), and to respond with a fixed rcode (NXDOMAIN) for domains (such as AS112).

You can put authority data on a separate server, and set the server in unbound.conf as stub for those zones, this allows clients to access data from the server without making unbound authoritative for the zones.

- The access control denies queries before any other processing.

  This denies queries that are not authoritative, or version.bind, or any. And thus prevents cache-snooping (denied hosts cannot make non-recursive queries and get answers from the cache).

- If a client makes a query without RD bit, in the case of a returned message from cache which is:

```
answer section: empty
auth section: NS record present, no SOA record, no DS record,
              maybe NSEC or NSEC3 records present.
additional: A records or other relevant records.
```

  A SOA record would indicate that this was a NODATA answer. A DS records would indicate a referral. Absence of NS record would indicate a NODATA answer as well.

  Then the receiver does not know whether this was a referral with attempt at no-DS proof) or a nodata answer with attempt at no-data proof. It could be determined by attempting to prove either condition; and looking if only one is valid, but both proofs could be valid, or neither could be valid, which creates doubt. This case is validated by unbound as a 'referral' which ascertains that RRSIGs are OK (and not omitted), but does not check NSEC/NSEC3.

- Case preservation

  Unbound preserves the casing received from authority servers as best as possible. It compresses without case, so case can get lost there. The casing from the query name is used in preference to the casing of the authority server. This is the same as BIND. RFC4343 allows either behaviour.

- Denial of service protection

  If many queries are made, and they are made to names for which the authority servers do not respond, then the requestlist for unbound fills up fast. This results in denial of service for new queries. To combat this the first 50% of the requestlist can run to completion. The last 50% of the requestlist get (200 msec) at least and are replaced by newer queries when older (LIFO). When a new query comes in, and a place in the first 50% is available, this is preferred. Otherwise, it can replace older queries out of the last 50%. Thus, even long queries get a 50% chance to be resolved. And many 'short' one or two round-trip resolves can be done in the last 50% of the list. The timeout can be configured.

- EDNS fallback

  Is done according to the EDNS RFC (and update draft-00). Unbound assumes EDNS 0 support for the first query. Then it can detect support (if the servers replies) or non-support (on a NOTIMPL or FORMERR). Some middleboxes drop EDNS 0 queries, mainly when forwarding, not when routing packets. To detect this, when timeouts keep happening, as the timeout approached 5-10 seconds, and EDNS status has not been detected yet, a single probe query is sent. This probe has a sub-second timeout, and if the server responds (quickly) without EDNS, this is cached for 15 min. This works very well when detecting an address that you use much - like a forwarder address - which is where the middleboxes need to be detected. Otherwise, it results in a 5 second wait time before EDNS timeout is detected, which is slow but it works at least. It minimizes the chances of a dropped query making a (DNSSEC) EDNS server falsely EDNS-nonsupporting, and thus DNSSEC-bogus, works well with middleboxes, and can detect the occasional authority that drops EDNS. For some boxes it is necessary to probe for every failing query, a reassurance that the DNS server does EDNS does not mean that path can take large DNS answers.

- 0x20 backoff

The draft describes to back off to the next server, and go through all servers several times. Unbound goes on get the full list of nameserver addresses, and then makes 3 * number of addresses queries. They are sent to a random server, but no one address more than 4 times. It succeeds if one has 0x20 intact, or else all are equal. Otherwise, servfail is returned to the client.

- NXDOMAIN and SOA serial numbers

  Unbound keeps TTL values for message formats, and thus rcodes, such as NXDOMAIN. Also it keeps the latest rrsets in the rrset cache. So it will faithfully negative cache for the exact TTL as originally specified for an NXDOMAIN message, but send a newer SOA record if this has been found in the mean time. In point, this could lead to a negative cached NXDOMAIN reply with a SOA RR where the serial number indicates a zone version where this domain is not any longer NXDOMAIN. These situations become consistent once the original TTL expires. If the domain is DNSSEC signed, by the way, then NSEC records are updated more carefully. If one of the NSEC records in an NXDOMAIN is updated from another query, the NXDOMAIN is dropped from the cache, and queried for again, so that its proof can be checked again.

- SOA records in negative cached answers for DS queries

  The current unbound code uses a negative cache for queries for type DS. This speeds up building chains of trust, and uses NSEC and NSEC3 (optout) information to speed up lookups. When used internally, the bare NSEC(3) information is sufficient, probably picked up from a referral. When answering to clients, a SOA record is needed for the correct message format, a SOA record is picked from the cache (and may not actually match the serial number of the SOA for which the NSEC and NSEC3 records were obtained) if available otherwise network queries are performed to get the data.

- Parent and child with different nameserver information

  A misconfiguration that sometimes happens is where the parent and child have different NS, glue information. The child is authoritative, and unbound will not trust information from the parent nameservers as the final answer. To help lookups, unbound will however use the parent-side version of the glue as a last resort lookup. This resolves lookups for those misconfigured domains where the servers reported by the parent are the only ones working, and servers reported by the child do not.

24.2 Unbound Resolver Prototype

The following is information regarding the Java based prototype for Unbound. The Java prototype was superseded by a C version, available here. *Older prototype versions can be downloaded* here.

Unbound is a project to develop independent open-source DNS software and documentation. It is jointly funded by VeriSign, Inc., and the University of Southern California/Information Sciences Institute.

This is our prototype full-service, iterative-mode, validating DNS resolver. It is written entirely in Java and is licensed under the BSD open-source license.

Even though it is a prototype, it is fairly full-featured:

- Supports DNSSEC validation
- Supports zone forwarding
- Supports “stub” zones
- Is resistant to cache poisoning
- Plus, other even harder to explain features!

24.2.1 Download the Original Java Prototype

Fetch the latest package and released version here:

Binary:
`unbound-prototype-0.10.1.tar.gz | (sig)`

Source:
`unbound-prototype-0.10.1-src.tar.gz | (sig)`

Note: As of unbound-prototype-0.10, support for NSEC3 validation is part of the trunk. Even more cutting edge versions (including branches) of the Java Prototype may be fetched via the subversion repository.

The above is the link to the whole prototype area, which you probably don't want to check out directly. Instead, either check out the trunk or a single branch:

```
svn co http://svn.verisignlabs.com/unbound/proto/trunk unbound-prototype
```

Tip: Don't have subversion? Don't even know what we are talking about? Head on over to the subversion home.

24.2.2 Prototype?

Java, while a fine, fine language, isn't what we envision as the final implementation language of the non-prototype resolver. That would be C. For the prototype, however, Java was chosen because of the excellent DNSjava library and the familiarity with the same on the part of one of the main developers.

The intent is to use a prototype to explore and validate a particular design for an iterative-resolver and separable DNSSEC validator.

24.2.3 Design

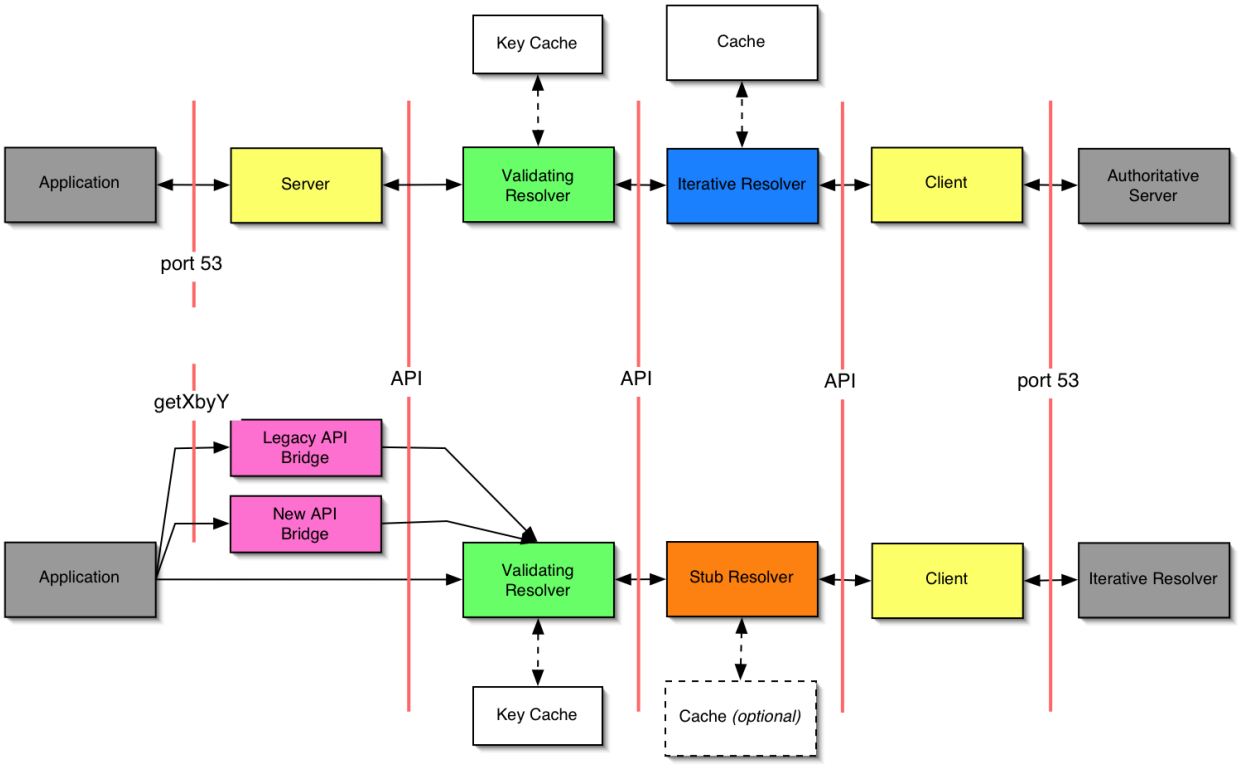
The unbound-resolver was designed to be modular. The idea was to create a "chain" of modules that could be brought together to form either a full-service resolver, or (for instance) a DNSSEC validating stub resolver. It also is designed to be able to use different implementations of a few key components, like the cache.

Here is a diagram that shows the basic components.

In the image, "API" is referring to an internal API which is not standardized *per-se*, but is internally consistent.

This version of the prototype has been written as an event-driven, asynchronous server. Each packet that is received is turned into an event (along with a few internally generated events, like timeouts and generated queries).

```
$Id: unbound-arch.graffle 54 2005-03-11 23:09:52Z davidb $
```



24.3 DNS Cache Poisoning Vulnerability (2008)

Authored April, 2008
Updated and released Jul 9, 2008

24.3.1 Executive Summary

Dan Kaminsky of IOActive has reported a DNS cache poisoning vulnerability to developers of DNS caching software. The details of this vulnerability will be explained by Kaminsky at the upcoming Black Hat conference in August.

A cache poisoning attack allows unauthorized third parties to inject data into a DNS cache, the injected data may cause rerouting of traffic.

There is no definite solution to the form of cache poisoning described to us by Kaminsky. Only DNSSEC will provide the measures to detect malicious data and prevent cache poisoning.

However in absence of DNSSEC being sufficiently deployed to benefit, methods exist to increase resilience against cache poisoning attacks, and Unbound has these implemented by design.

- Unbound was designed to use the maximum amount of randomness for query ports and does not need to be upgraded to improve resilience.
- Unbound has also been designed with a component, called the scrubber, that sanitizes query results and does not add data to the cache it does not trust as being authoritative.

24.3.2 Background

Absent a reference to Dan Kaminsky's work we refer to other papers where the general principle of port and query ID randomization to mitigate cache poisoning attempts are described.

Recently an article appeared in the Register where a statement that Amit Klein made during the RSA Security conference was quoted as:

> *I'm not too comfortable with the quality of the solution from the security and predictability standpoint*

In 2007 and 2008 Klein has published a number of articles on port and query ID predictability on the website of his company. Work started within the IETF by Hubert and van Mook in 2006 already indicated that making optimal use of the full port range improves cache forgery resilience, and recommends the use of good sources of randomness for the choice of query IDs and ports.

A new strategy to improve the resilience against forgery is the work by Vixie and Dagon, that uses the property that authoritative servers copy the query name, while preserving case, to introduce more entropy matching queries against answers.

24.3.3 Unbound

Unbound is being designed to be a modern, secure and up-to-date name server, that in addition to DNSSEC (the only complete solution to DNS forgery) implements all known mechanisms to increase the resilience to forgery of cache entries.

Unbound version 1.0 has implemented query id and port randomization in the following way.

For each outgoing query both the query-ID and port number are set by routines based on the ARC4 algorithm as implemented in OpenBSD. Arc4random is a pseudo random generator based on ARC4 that addresses the problems, as Klein mentions in one of his papers:

> *Fortunately, OpenBSD also uses [[. . .]] the ARC4 algorithm. As such, spoofing responses for the resolver necessitates knowing the UDP source port, predicting the ARC4 algorithm's next output, or exploiting an implementation bug, none of which is currently known to apply.*

To allow for the maximum number of ports to pick from randomly, Unbound uses an advanced strategy. Ports are chosen chosen randomly from the pool of non-allocated ports. Ports can be excluded from this pools by means of a configuration parameter.

There is a small penalty for recursive name servers that will have to do a lot of recursion. This is because there is only a limited set of slots in the socket cache and one may need to wait before one is available. For this penalty there is a compensation by the use of a socket cache which helps the kernel run its select algorithm over open file descriptors.

The mechanisms utilizing case preservation as described by Vixie and Dagon is under development and can be turned on while configuring Unbound 1.0. It should be noted though that the likely hood on false positives, which causes Unbound to ignore an answer thinking it is under attack, is fairly high. There are strategies to deal with these false positives that delay the response time somewhat and are currently under investigation. We plan an implementation of those in once we are satisfied with the behavior.

We believe that the scrubber safeguards against the currently publicly known cache poisoning tactics.

24.3.4 Links

- http://www.theregister.co.uk/2008/04/15/dns_cache_poisoning/
- http://www.trusteer.com/research/ contains various articles focused on predictability of port and query IDs
- http://tools.ietf.org/html/draft-ietf-dnsext-forgery-resilience
- http://tools.ietf.org/id/draft-vixie-dnsext-dns0x20 work in progress
- http://unbound.net/documentation/doxygen/random_8c.html

24.4 Unbound 1.0.2 Patch Announcement

24.4.1 Summary

Unbound version 1.0 was released with port randomization features. The same features that have been made available in the various patches by other vendors the CERT alert last month.

Now Dan Kaminsky disclosed more details during the the august 2008 Blackhat Conference in Las Vegas we release additional counter measures. These counter measures were previously withheld in order to minimize the risk of disclosing details about several variations of the attack through reverse engineering.

With the current set of counter measures added, Unbound offers state of the art protection against the attacks described by Kaminsky. However, state of the art counter measures will not provide full protection, not in Unbound nor in other software. Although DNSSEC is hardly deployed, it is currently the only mechanism known to deal with spoofing and other kinds of attacks on the DNS.

More details in the ways that Unbound protects against spoofing are below.

24.4.2 What is Cache Poisoning

Poisoning a DNS resolver refers to the act of inserting fake, often malicious data into the resolvers cache. This can cause website visitors to be redirected from the site (e.g. their banking site) they thought to visit to a different web site, for example a phishing site.

The basic approach of poisoning DNS queries is to send fake replies that pretend to come from the authority servers to the caching resolver. Every DNS query carries a random query Identifier (16 bit number). Only replies that contain the same number are accepted. In order for the resolver to accept the fake replies, the Identifier in the incoming packet needs to match that of the outstanding question. That is, the attacker has to guess a number of 16 bits in length.

One can calculate the time how long it takes to guess the 16 bits Identifier. a detailed calculation can be found online (draft-ietf-dnsext-forgery-resilience). The document contains formulas with all the variables involved. The example given in the draft has a 4 Mb/s attack rate to get a 50% chance to inject the fake data.

It comes down to this: it takes a certain time, on average, to guess the right random sequence. The example in the reference argues that the Kaminsky exploit takes about 10 seconds to guess the 16 bit value in the identifier. This is confirmed by implementations of the exploit as well as various calculations on public mailinglists.

In order to extend the time by which a packet can be succesfully replaced we need more than the 16 bits of random number that the query Identifier provides us. That can be done by putting random numbers in other parts of the query, and checking if the server puts the same number in a reply, without changing the protocol. It is hard to add these extra random numbers without breaking interoperability, because the reply is only defined to contain a copy of the 16 bit ID value. Once an extra random number is copied into the reply, a fake reply must guess that number. Every extra bit that needs to be guessed, increases the time by a factor of 2. The goal is to add enough extra bits that the chance of poisoning becomes very low (on average).

Unbound implements a number of methods to add random bits. The most important means to add randomness is to vary the port numbers from which the question is asked, another means is to use a hack that randomizes unused bits in the query name. Unbound implements even more methods. In addition, Unbound is careful in what to accept as information that can be cached. These techniques are explained in more detail below.

Note however that the increase in the amount of bits does improve your chances to safely cross the road but a bad packet may still hit you.

Real protection, where you are not subject to the whims of chance, is achieved by using DNSSEC. DNSSEC uses digital signatures to protect the data. With DNSSEC there is no chance of poisoning, independent of the number of random bits used.

24.4.3 Unbound Security

Unbound implements the DNSSEC standard as specified in **RFC 4034** and **RFC 4035**. This means that it can act as a validator and can thus check the digital signatures attached in replies. Of course, the domain name owner must have inserted these digital signatures in the first place.

In the absence of DNSSEC, unbound attempts to provide very good security. Without digital signatures, randomisation and filtering are currently the only options. Below, a technical categorisation is made of the methods employed by unbound to protect unsigned data.

24.4.4 Filtering

Unbound contains a component we call a 'scrubber'. This component takes care of certain checks, disallowing (removing) possibly malicious content.

- Only in-bailiwick data is accepted
- RFC 2181 trust is employed. This means that data from the additional section receives an additional section trust. And data from the answer section receives answer section trust. Data with additional section trust is not used to answer queries from clients. Thus putting a record in the additional section cannot make this record appear to clients.
- The records in the authority and additional section are filtered for relevance to the query in question. If the data is irrelevant, it is removed.
- The answer section is filtered for relevance. Only answers to the query that unbound wants to ask are allowed.
- CNAME chains are cut off, only the first CNAME is kept as answer. The remaining CNAMEs or answer records are not kept, but looked up instead.
- For DNAME records, the CNAME is synthesized by unbound itself, it does not trust the server to do so.
- DNAME records are not taken from the cache to perform the redirection, even if they seem to match. Only for validated DNAME records (where the digital signature was correct) is redirection performed from cache, this requires the use of DNSSEC.

24.4.5 Randomisation

By adding more random data, a spoofed reply has to guess more data to get through, lowering the chances of a successful poison attempt.

- Strong random number generator. Unbound uses a cryptographic strength random number generator. The arc4random() generator from OpenBSD is used. This means that predicting the random numbers generated by unbound is equivalent to cracking an encryption cipher.
- The random number generator is seeded with entropy. Real entropy from the system /dev/random is used to seed the random number generator. Thus, the starting values of the random number generator cannot easily be predicted.
- Query ID bits. Unbound uses all 16 bits in the ID.
- Port randomisation. Unbound uses 16 bits for the port randomisation. To be precise, about 60000 random ports, avoiding ports below 1024 and avoiding IANA allocated UDP ports to avoid system instability of the server. The port randomisation uses the same random number generator as the ID. Unbound takes care that a randomly drawn port is used for one query. Thus every query gets a freshly random port number.
- Destination address randomisation. Unbound performs RTT banding, a method to select the destination server that provides additional randomness. This provides between 1 and 4 bits of randomness. Perhaps 2 on average. Arguments that choosing the fastest destination reduces the attack time window are no longer relevant given the recent full disclosure at the Blackhat conference. Additional time windows are easily achieved.
- Source address randomisation. If configured with multiple public IP addresses, unbound can perform a random choice of interface. This needs operator configuration, but by adding 4 outgoing-interface statements in the config file, an additional 2 bits of randomness are achieved.
- Transport protocol randomisation. If IPv6 is available (yes, yes, not very common), then unbound will obtain another random bit by choosing the IPv4 or IPv6 transport protocol randomly.
- Query aggregation. This prevents identical outstanding queries to the same server. It prevents birthday-paradox attacks.
- Query name strict matching. This prevents an answer from matching a query for which it is not meant. If an answer can match multiple queries, you get the birthday paradox attack again (from the previous item).
- Capitalisation randomisation. Also called dns-0x20. This is an experimental resilience method that uses upper and lower case letters in the question name to obtain randomness. On average about 7 or 8 bits. This method currently has to be turned on by the operator manually, as it may result in maybe 0.4% of domains getting no answers due to no support on the authoritative server side.

24.4.6 Additional security measures

These measures are mostly to prevent remote execution exploits.

- Heap function pointer protection
- chroot() by default
- user privileges are dropped by default
- access control list for clients that are allowed recursion
- No detection of attacks underway. Unbound assumes it is always under attack
- can config the version.bind or hostname.bind answer to return, or block the queries

24.4.7 Randomness Calculation

So the default setup has a randomness of:

```
16 bits ID
16 bits port
2 bits destination address (estimated average).
```

For a total of 34 bits of randomness. Other implementations provide 16 bits (or less) unpatched, 26 bits for patches utilizing only 1024 ports and 32 bits for patches using the fully available port range (around 60k). Unbound has been utilizing the full port range of about 60.000 ports since the release of version 1.0.

With a careful setup, enabling capitalisation and source address randomisation Unbound provides:

```
16 bits ID
16 bits port
2 bits destination address (estimated average)
2 bits source address (estimated average)
8 bits capitalisation (estimated average).
```

in total 44 bits of randomness.

Sample config file items to enable this amount of randomness:

```
server:
    # configures 4 static public IP addresses.
    # you can also enter IPv6 if you have it.
    # this is an example, you must enter your addresses.
    outgoing-interface: 192.0.2.1
    outgoing-interface: 192.0.2.2
    outgoing-interface: 192.0.2.3
    outgoing-interface: 192.0.2.4
    # enable dns-0x20.
    use-caps-for-id: yes
```

24.4.8 Time to infection

We take 10 seconds to infect an unpatched server with 50% chance as a baseline. The table below shows the time until a poison attempt is successful. The numbers are subject to being guesstimates. Better numbers may become available, either from the Blackhat presentation, or other sources. The bottom line is that adding randomness is a short term fix.

Bits	50% chance	5% chance	Aka
16	10 seconds	1 second	unpatched server, random ID
26	2.8 hours	17 minutes	patched, using only 1024 ports
34	28 days	2.8 days	unbound using defaults
44	28444 days	2844.4 days	unbound with capitalisation and source addresses configured *

** : These are not enabled by default. The capitalisation has not been standardised, and could result in a small number of cases in slow or no answer. The source addresses need the operator to configure multiple addresses for the computer.*

In the table above, the Bits column shows the number of random bits that are echoed in replies. The 50% chance column shows the length of time needed before an attack has a 50% chance of success (guessing the random numbers). The 5% chance column shows how long it takes before an attack has a 5% chance of inserting fake data.

Note: 60000 sockets not 65536 sockets used randomly for unbound is assumed in the table entries for unbound. Unbound avoids some port numbers for compatibility.

Also note that the table above assumes a fairly low bandwidth usage. If a large network capacity is available, say a botnet, and it can use 1000x more resources, then perhaps also the attack can be conducted 1000x faster.

In the meeting of the IETF dnsext working group successful poisoning attacks against an unpatched server in as little as 1/10 of a second were demonstrated easily (demo results), showing that much smarter things can be done than the dumb attack assumed for the numbers here. Calculations by members of the working group showed a near perfect chance for 6-8 seconds. This could move the figures to be less optimistic.

Keep in mind that the thousands of days shown for unbound with capitalisation and source addresses configured should not be taken as strong security. It is likely that some measures can be outsmarted. Or that these numbers are overly optimistic (see text above). And the 44 bits is an average. If an attacker can work out how to attack domains or queries with less protection, the the benefits may be partially lost. Thus, the large time listed for 44 bits should be taken as an indication that it is pretty good, but not invulnerable.

As stated earlier, the real solution is to use DNSSEC. DNSSEC makes this time table a non problem, because in all these cases DNSSEC can detect the forgery. Especially users in Brazil, Bulgaria, Puerto Rico and Sweden or people using these zones regularly, should consider turning on DNSSEC because the TLD zone is DNSSEC secured. Do consider using the DNSSEC capabilities in Unbound.

24.5 Unbound Operation Explained in Book

Authored December 2008.

We received a complimentary book by Jan-Piet Mens today, titled Alternative DNS Servers. It covers a whole host of DNS servers, including NSD and Unbound.

The book describes how to set up DNS servers and how to operate them. I found the section on Unbound to be fine (also NSD is fine). I cannot comment on the other products.

One section stood out as it has a performance comparison of the servers. The book has more details, below is one line of results. Here 10 queryperf machines query a DNS cache, and the average queryperf performance is noted. So the cache is doing 10x the number noted. The figures show similar results to what we find for performance comparisons in the NLnet Labs testlab. The results below have been found independently, and compare a greater number of products.

Server	Queries/sec (10 clients)
MaraDNS	3 068
BIND	3 003
dnscache	2 928
PowerDNS Recursor	2 074
Unbound	8 276

The book reviews unbound version 1.0, and the config and operation is the same as 1.1 which was recently released. Unbound 1.1 has DLV support and improved statistics, which may be of interest.

24.6 Unbound Timeout and Server Selection Information

24.6.1 Introduction

Unbound sends requests upstream to the authority servers on the internet and these requests can timeout. These timeouts have to be handled. Either the request has to be sent to another server, or resent. And the responsiveness of the destination server has to be kept track of.

The handling of timeouts is complicated by conflicting requirements. If a server is down and not responding, continuation of sending packets is a waste of resources. These resources are typically the time spent waiting, the socket and port number, and request list entry that are used during that time. But if the server was down briefly, and has come up, then it is important to detect this quickly. Especially in cases where the timeout involves a high traffic destination (say, the local organisation's own domain), then it is especially pressing, and the domain must not be blocked for hours.

Unbound has several different regimes that can be distinguished. These are described separately for explanatory purposes, in the code the mechanisms all operate together.

24.6.2 Normal Operations

In the normal case, requests and replies are flowing like they should. Unbound has to set a timeout because UDP is an unreliable transport mechanism and a packet may get lost once in a while. To do this, it keeps a roundtrip time estimate and performs exponential backoff.

The timeout is measured in milliseconds and is kept per IP-address (so, not by host name but by host address). This is stored in the infra-cache. The infra-cache can be configured in the max-number of elements it stores, and the TTL (time to live) of the elements inside the cache. By default elements exist for 15 minutes in the infra-cache.

The fastest server (randomly picked within a so-called RTT band of 400 msec) is selected when a query has to be sent out. The roundtrip-timeout (`rtt`) is used for selection purposes. This is the value of the timer that would be set if the packet is sent out. When this timer expires, the packet is considered timed-out. If nothing is known about an IP-address a timeout of 376 msec is assumed. This assumed timeout should be successful for most traffic. The 376 is chosen to fall within the 400 msec rtt band and it is also a reasonable value (many pings fall in it) while still allowing several resends within about a single second.

When packets return successfully from the remote server, the ping-time is used to update the estimate of the roundtrip timers. A smoothed average roundtrip time is kept, that can keep track of a slow change towards a new average. Also a smoothed variation measure is kept, that keeps track of the jumps in the times observed. And when a timeout happens the exponential backoff is kept track of. Exponential backoff means that the roundtrip timeout is doubled for every next packet. These values are stored in the infra-cache and return to their defaults when the TTL expires on the element for that IP address.

If a timeout occurs, the packet is considered lost and the cache is updated by doubling the timeout to apply for the next packet. Server selection is performed again, and will likely pick another server to send to. If the server was very fast, then it may be picked again since the doubled value is still very small. But if the server gets slower, it will no longer be preferred and traffic is sent to another server for that domain.

If a server is selected again, the same query can be sent again to the same server, but now with a larger timeout. Unbound no longer listens or wants to receive a reply to the timed-out queries at that point. This is because listening to multiple outstanding versions of the same query sent to a server creates a (small) birthday paradox. And this is avoided for cache-poison resistance reasons.

If many requests are sent to a destination server at the same time, then a short interruption could cause many of them to timeout at about the same moment. This would, with exponential backoff, result in an almost infinite backoff to be applied. Therefore some race-condition protection is applied. The timeout in the infra-cache is increased to double the original value that the query was sent out with. Thus if the doubling has already been applied by another failed packet, it is not applied again. The doubling is only done if the timeout stored is between the original value and its double.

So that if another query has already succeeded and lowered the value in the cache then this is left as-is, since traffic is flowing again.

In normal operations, many threads can have many packets outstanding to an IP address, all at the same time. The infra-cache data is shared between threads.

24.6.3 Probing

When a domain starts to become unresponsive, it is probed. In this regime only one request is allowed to probe to a particular IP. This conserves resources, as other requests are turned away, and do to not use up port-numbers, sockets and requestlist elements. Also it lowers the traffic towards the destination (that is apparently having trouble), which may help it get back up.

An IP address is in the probing regime if it fits the following criteria.

- The timeout (with exponential backoff applied) exceeds 12 seconds
- Two (or more) consecutive exponential backoffs have just been done on it

These conditions can not be configured. They mean that the query has just had two timeouts, and it is already very slow (12 seconds timeout). If it normally has a timeout that is high, say 10 seconds, then the timeout has to reach 40 seconds before this restricted regime applies. If it is normally very fast, then normal operations continue for about 24 seconds (because of exponential backoff, the total time for the timeouts in sequence). For queries that normally take about 100 msec or so, about 6 timeouts have to happen before it hits a 12 second timeout.

In this regime, when a probe request is sent to the destination IP address, the `exclusion time` until another probe can be allowed is stored. This is the current time plus the timeout for this packet plus one (see below about the plus one). Other queries are not sent to this IP address until that time.

The exclusion time is stored in the infra-cache. This means that it is shared by the threads. So normally, one request can probe at a time. In some cases, the code can allow a small window of opportunity and multiple probes, one per thread, happen at the same time. This only happens when traffic it very large towards that domain and is otherwise harmless.

When the probe is done and is it successful - so an answer came back - then the roundtrip estimates are updated with this new observation. And the IP address is put back into normal operations. Many queries are allowed to the destination server.

When the probe is done but it was a timeout, the exponential backoff is increased. And the probe query tries to select a new server for that domain to send to. But because of the plus-one on the exclusion timer, is now excludes itself from sending to that server again. It may probe another IP-address for the same DNS domain at that time, but not the same one right away.

This self-exclusion generates some useful effects.

- If there is very little traffic towards an affected domain, then a single request will slowly probe the different servers (if there are multiple servers, otherwise, with one server only, it will give up quickly).
- If there is a moderate amount of traffic towards an affected domain, then several requests will probe, each picks up a different IP address and probes one time. But because they all arrive randomly the exclusions mean every request performs usually one probe only as the other servers are (being) probed by other requests when it finishes probing an IP address. And there is a little wait before a new query comes in to probe a new server, in that time an already probing query is allowed to probe this IP address again. When another request comes in, probing the servers continues. Thus there are some queries probing one (or some more) different IP addresses, but not all IP addresses are probed at the same time.
- If there is high traffic towards an affected domain, then requests are always available as soon as the exclusion ends. Thus all the servers for that domain are probed at the same time, each server receives one query at a time. The requestlist contains an element for every server to probe.

If more requests arrive at the server than can be used for probing, these are turned away.

When a request is turned away because the servers are probed and this request did not attain probe status, then it gets the DNS error code SERVFAIL. These requests do enter the requestlist, but do not use a socket or a port number, as they get an error reply when it finds out that no servers are available to send packets to.

Another effect is that once a query is excluded from all currently known servers for a domain, the fallback mechanism to handle misconfigured domains is activated. This searches for additional servers that may respond for this domain name.

In the probe regime, IP addresses that are becoming unresponsive are probed by single requests and other requests are turned away. At some point the exponential backoff becomes too large and it seems useless to send further traffic to that server.

24.6.4 Blocking

In the blocking regime, the timeout reached 120 seconds and further requests towards the server seem useless. All requests are turned away and receive SERVFAIL (unless another working server exists for that domain).

Requests do enter the requestlist, briefly, but when it turns out all servers are unresponsive, it is turned away with the error SERVFAIL.

This condition is cached in the infra-cache element for that IP address. The elements in the infra-cache live for infra-ttl seconds (15 minutes by default). When this TTL (time to live) expires, then the domain is probed again.

Performing the full probe sequence would take about 240 seconds (sequence of exponential backoffs until it is 120 seconds). With a 15 minute time to live, this is a bit excessive, especially if normal operations resumes and many resources are expended on this likely-unresponsive server. Therefore only a single probe packet is sent if the infra-ttl has expired. If that probe fails, then the server is blocked for another infra-ttl.

The result is that a server is probed with one packet every 15 minutes. If it succeeds, all traffic is allowed again (normal operations), and if it fails, the next probe is sent after blocking the server for 15 minutes. So if a server comes back up, this is observed within infra-ttl seconds. If a server does not respond, it is probed every 15 minutes, but only if there are queries to send to it.

The way the code works means that if an `expired` infra-cache element exists, and it says the address was blocked, then a single probe is performed. Such expired entries can exist until the cache runs out of memory and flushes elements out to make space for new elements, the infra-cache uses the LRU cache-algorithm for that. Servers for a domain for which very little queries are received, do not get probes sent to them, and when finally a query arrives for it, a single probe is done so as to not squander resources.

24.6.5 Control

The timeout behaviour can be controlled and configured.

The configuration consists of the size of the infra-cache (please allow sufficient elements to store information about IP addresses). And the infra-ttl time can be configured. By setting the infra-ttl lower, unbound will probe servers that are not responsive more aggressively.

The `unbound-control` tool can be used to interact with the running server. It can provide information and flush cache entries. The `flush_infra` command can be used to flush all of the cache or particular elements. The `lookup` command shows status for the servers associated with a particular domain. The `dump_infra` command dumps the entire contents of the infra-cache, a snapshot of the ping-times of the servers on the internet that unbound has contacted.

The output of a `lookup` command can look like this:

```
$ unbound-control lookup nlnetlabs.nl
The following name servers are used for lookup of nlnetlabs.nl.
;rrset 9911 3 1 7 3
nlnetlabs.nl.       9911    IN      NS      omval.tednet.nl.
nlnetlabs.nl.       9911    IN      NS      open.nlnetlabs.nl.
nlnetlabs.nl.       9911    IN      NS      ns3.domain-registry.nl.
nlnetlabs.nl.       9911    IN      RRSIG   NS 8 2 10200 20101129015003
    20101101015003 42393 nlnetlabs.nl. H28rD+MVEYWYm5aceRHg
    rf4gkLplnPhJjeYG5tKc quzyAUtQv2/IfQWDbKWz wdGGwhwFIF91Fio9ogAm
    2UrukBtE5Z7LAp1D0ZUZ uqnbWCsXXYcpayHDO3t T3oCd73JPChm5nPlw+NU
    VmqGWpSP8/4MoDsgPYdR 88MK2NdqZ0F8= ;{id = 42393}
;rrset 177 1 0 8 0
ns3.domain-registry.nl.     177     IN      A       193.176.144.6
;rrset 177 1 0 8 0
ns3.domain-registry.nl.     177     IN      AAAA    2a00:d78:0:102:193:176:144:6
;rrset 5399 1 1 8 3
open.nlnetlabs.nl.  5399    IN      A       213.154.224.1
open.nlnetlabs.nl.  5399    IN      RRSIG   A 8 3 10200 20101129015007
    20101101015007 42393 nlnetlabs.nl. noDw4tW3WSEphAj8eXtg
    aiqt4qNBD3KFvFjv+rss iW/QYkKjxDl7j2xPGLWY pTk1XdWa21k0xYTpgshA
    3vh9JB69FCfwHnuxIC/o Ksy6g43TIOmOYuENaOIs OZ8MwvrHuGpLxjUo5QPq
    rQO/yuVz5pgFFsSScJwZ ZiYQSjwfTBU= ;{id = 42393}
;rrset 5399 2 1 8 3
open.nlnetlabs.nl.  5399    IN      AAAA    2001:7b8:206:1::53
open.nlnetlabs.nl.  5399    IN      AAAA    2001:7b8:206:1::1
open.nlnetlabs.nl.  5399    IN      RRSIG   AAAA 8 3 10200 20101129015007
    20101101015007 42393 nlnetlabs.nl. ZXSeWEgkY4xhEwlDdTsj
    FM12r31L/MMQYaDFeGki YTUeWJRFzGa4w3+A+FHp mibdVKuscGTuPWtsP2zE
    29u6ClcW0NDM+KfbEV+D zUYH88f7P1qs1sZSKGJL owxzREKDVF1t5iThVLIZ
    l49aD/mL97eNJ60Ybwov nsoFVuEt5Ao= ;{id = 42393}
;rrset 18042 1 0 8 3
omval.tednet.nl.    18042   IN      A       213.154.224.17
;rrset 18042 2 0 8 3
omval.tednet.nl.    18042   IN      AAAA    2001:7b8:206:1::17
omval.tednet.nl.    18042   IN      AAAA    2001:7b8:206:1:200:39ff:fe59:b187
Delegation with 3 names, of which 0 can be examined to query further addresses.
It provides 8 IP addresses.
2001:7b8:206:1:200:39ff:fe59:b187   not in infra cache.
2001:7b8:206:1::17  not in infra cache.
213.154.224.17      not in infra cache.
2001:7b8:206:1::1   rto 284 msec, ttl 860, ping 0 var 71 rtt 284, EDNS 0 probed.
2001:7b8:206:1::53  rto 164 msec, ttl 420, ping 0 var 41 rtt 164, EDNS 0 probed.
213.154.224.1       rto 72 msec, ttl 130, ping 0 var 18 rtt 72, EDNS 0 probed.
2a00:d78:0:102:193:176:144:6        not in infra cache.
193.176.144.6       rto 230 msec, ttl 105, ping 2 var 57 rtt 230, EDNS 0 probed.
```

Some servers are listed as not in the infra-cache. For the ones in the infra-cache, the rto (roundtrip timeout with exponential backoff applied) is printed, and the ttl of the infra-cache element. Also the ping-time (the smoothed roundtrip time) is printed (in msec) and the variability (in msec), the roundtrip timeout without exponential backoff (rtt) is also printed (in msec). The infra-cache also contains EDNS status and lameness information which is also shown. In the above example, the ping time is very low as most servers are on the same subnet.

```
192.0.2.1 ttl 316 ping 0 var 94 rtt 376 rto 120000 ednsknown 0 edns 0 delay 0
```

The dump_infra command produces similar output. Here is an example (only a single line from the very long output) that shows a blocked entry. The 120 second rto means it is blocked. The rtt of 376 (still at the assumed default), leads us to assume it never replied. 192.0.2/24 is a netblock for documentation purposes and not deployed on the internet, hence no replies.

24.6.6 Summary

Unbound implements timeout management with exponential backoff and keeps track of average and variance of the ping times. If a server starts to become unresponsive, a probing scheme is applied in which a few queries are selected to probe the IP address. If that fails, the server is blocked for 15 minutes (infra-ttl) and re-probed with one query after that.

Queries that failed to attain probe status, or if the server is blocked due to timeouts, get a reply with the SERVFAIL error. Also, if the available IP addresses for a domain have been probed for 5 times by a query it is also replied with SERVFAIL. New queries must come in to continue the probing.

The status of an IP address can be looked up and flushed. The infra-cache is not flushed on a reload, so the list of blocked sites and ping times is not wiped. If you wish to remove it the flush_infra control command can be used.

24.7 DNSSEC Algorithms with Unbound

Unbound validates DNSSEC signatures and in the case that there are multiple signature algorithms in use, it checks that a valid chain of trust exists for each algorithm separately. Thus the algorithms that are in use must all be subverted before validation can be misdirected.

24.7.1 Algorithms in the Chain of Trust

The algorithms that are checked are signalled via the DS RRset. This means that zones do not receive these checks until they publish multiple algorithms into their DS set. Thus the set of algorithms present in the DS RRset must have DNSKEYs and signatures on every data element.

The RFCs already mandate that for algorithms signalled to be in use for a domain you must have DNSKEYs and signatures on every data element, because a validator is allowed to continue the chain of trust if it supports one algorithm but not the others. These validators that support one of the algorithms must find that the algorithm signalled to be present has keys and signatures, and if these are missing, will conclude that signatures have been 'stripped' away. The extra checks that unbound performs thus must succeed if the domain is properly signed and all signatures are present.

There is some leeway when signing a domain, and this leeway is useful when changes are phased in. The DNSKEY may contain more algorithms, perhaps as part of a rollover. The data may be signed with other algorithms as well. It is possible to have DS records for which no key exists, as long as another DS record for that algorithm has a key. It is possible to have DNSKEY records that do not sign any or only part of the data (as long as signatures are available via other DNSKEYs).

Change in algorithms is possible by introducing keys in the DNSKEY set, and signing with them, and once complete, introducing the DS record. The reverse, first with the takeown of the old algorithm DS records, for removal of a signing algorithm. Older versions of unbound did not allow introduction of a new algorithm key in the DNSKEY set if the signatures on the data were not already present, but newer (since 1.4.8) versions allow this (and rely on the algorithms signalled in the DS RRset).

24.7.2 Protection

The check for multiple algorithms protects against not-known-today algorithmic weaknesses in one algorithm by using the other algorithm. This assumes the (mathematical) properties of the algorithms are dissimilar and that any deficiencies are not discovered simultaneously.

So, for example, RSASHA1 and RSASHA1_NSEC3 is a poor choice in this regard, as the algorithms are identical (the algorithm identifier is used to signal NSEC3 support here, which was useful during the introduction of NSEC3). Also the use of multiple keys only protects like the largest one.

24.8 Trust Anchors

Trust anchors can provide multiple algorithms, if a trust anchor contains multiple algorithms, a valid chain of trust is checked for them. Similar, if a RFC5011 automated key state contains VALID (or MISSING) keys with multiple algorithms, these algorithms are checked. For RFC5011, key revocation is checked and performed before the other checks in the RFC5011 state table when processing a DNSKEY probe, to make algorithm rollover possible (specifically the removal of the last key for the old algorithm).

24.9 Trust anchor retrieval less then 30 days before the KSK rollover

There is an issue for new installations less then 30 days before the rollover with Unbound versions prior to 1.6.5 (1.6.4 or older). The KSK2017 will be added in the ADDPEND state for 30 days (RFC 5011) and will not be in the VALID state during the key rollover. All is fine for trust anchor files created more then 30 days before the KSK rollover or after the KSK rollover, in any Unbound version.

24.9.1 Solution for installations less then 30 days prior to KSK rollover

You can either update to Unbound 1.6.5 (or later) or download the trust anchor file from this website.

Update to Unbound 1.6.5 or later

Delete the root.key file with `rm root.key`, then run `unbound-anchor` (1.6.5 or later) to create the root.key file again. You can verify that worked by checking that both keys have the string VALID in the newly created root.key file.

Download the trust anchor file from the Unbound website

If updating to Unbound 1.6.5 or later is not possible, you can download a trust anchor file containing the two VALID keys.

CHAPTER

TWENTYFIVE

DOCS TO-DO LIST

Since the first release in 2007, the documentation of Unbound has been maintained with a heavy focus on *manual pages*. As the resolver has become more versatile and feature-rich over the years, the NLnet Labs team decided to add this documentation, providing installation guides for different platforms, practical use cases, and background information.

The to-do list below provides an overview if the the topics we still have to cover. If you feel something is missing, please open an issue on GitHub to let us know.

Note: If you would like to write one or more of these pages, we're happy to compensate you for your time. Contact us at docs@nlnetlabs.nl or find us on Twitter.

25.1 Use Cases

- Resolver setup for enterprise networks
- Resolver setup for ISPs
- Maximum privacy resolver

25.2 Topics

- Resiliency (e.g. Rate Limiting, ACLs)
- EDNS Client Subnet

25.3 Filtering and Manipulating Data

- Local Zones and Local Data
- Expansion to all RPZ triggers and actions

25.4 Privacy

- Auth Zone
- Encryption
- QNAME Minimisation

25.5 Internals

- Architecture
- Code structure
- Server selection
- DNSSEC Trust Anchor Management (unbound-anchor and **RFC 5011**)
- Python modules

CHAPTER

TWENTYSIX

INDEX

- genindex

INDEX

Symbols

A

C

D

H

N

R

T

U

V