robdns 0.2 - manual

# Introduction

ROBDNS is the *fastest* DNS server for handling *authoritative* queries. It is designed for *Internet-scale* applications, such as *root* servers, *TLD*s with 100 million names, *hosting* with 1 million zones, and *blackholing* with thousands of updates per second. It’s is designed to be exposed to the public Internet under constant attack by hackers and DDoS.

It is currently in *prototype* stage of development. It has the basic features for a working DNS server in order to benchmark and test features, but requires extensive testing before it would be adequate for real-world use.

The server is broadly compatible with BIND9, reading the same zonefiles and conf files. Where BIND9 features are supported, configuration parameters have the same names. Configuration is described in this document.

The purpose of this server is to act as an Internet-exposed *slave server* (and *secondary master*), protecting a *hidden master* from attack. It is *authoritative only*. It cannot perform recursive queries, so cannot be used as a *resolver* or *proxy* *slave*. It is designed for a very specific deployment scenario.

Zones can be either *dynamic* slave zones, updated through DNS (UPDATE, NOTIFY, IXR, AXFR), or *static* master zones which read updates from the zonefiles on the drive. Presumably, such zonefiles come from a hidden master through an out-of-band system such as rsync or scripts.

It is a *modern* server, being up-to-date with the latest *RR types*, *EDNS0*, *NSID*, *TSIG*, and so on. It supports *DynDNS* features (NOTIFY, IXFR, UPDATE)*,* including zone creation/deletion for *superslave* mode. It supports DNSSEC record types, but has no ability to sign or validate zones. In such cases, the hidden master may not be a DNS server at all, but scripts which simply extract information from an SQL database generating zonefiles.

It supports many operating systems, such as Windows, Mac OS X, and Linux. These are the primary testing environments, with Linux the preferred system. While it’s not tested for other platforms, it will likely compile and run on almost any system.

## Internet scale DNS

The purpose of *ROBDNS* is to explore Internet-scale DNS performance. DNS is an essential part of Internet infrastructure. However, rather than being written like routers and firewalls, DNS software is written like typical user apps. This project changes that, creating a DNS server written with the same infrastructure techniques as hardware appliances.

The biggest problem is performance under DDoS attacks, where attackers will attempt to flood the server with packets, overloading it with queries. Since such attacks can be sourced from around the Internet to generate 100-gbps of network traffic, they will be far faster than any machine can handle, no matter how that software is written. However, the more the code is optimized, the fewer number of servers will be needed to cope with attacks.

This software is not simply faster than other servers, but 10 to 100 times faster. A single machine running ROBDNS can answer queries at a rate 100 times faster than BIND9, even for difficult problems like the *.com* zone or serving a million zones. Using this software would thus significantly reduce hardware requirements.

The intended deployment is as a widely *anycasted* service through the Internet in a *hosting* environment with a million customers. Each server should be able to handle a full 10-gbps of incoming queries, and a spread of many anycast machines should diffuse DDoS attacks.

## Designed for cybersecurity

Beyond simply DDoS threats, the primary design point for this software is cybersecurity.

# Building and Installing

This is an open-source project, with code hosted on GitHub in the usual manner. It compiles on Windows, Mac OS X, Linux, and most other platforms. It has few dependencies.

To get the source code, you can run the *git* command:

$ git clone https://github.com/robertdavidgraham/robdns

Alternatively, you may want to fetch the zip file containing the latest code.

$ wget https://github.com/robertdavidgraham/robdns/archive/master.zip

$ unzip master.zip

You will probably need to download libpcap/winpcap, and then you can simply build it:

$ cd robdns

$ make

$ make test

Always run the regression test suite, *make test*, after a build in order verify that everything is working correctly.

## Building on Debian/Ubuntu Linux

On a raw system, you’ll need the following:

$ apt-get install build-essentials libpcap-dev

$ git clone https://github.com/robertdavidgraham/robdns

$ cd robdns

$ make test

## Building on Mac OS X

To build the software, you’ll need the development tools from Apple. They are free.

After that, there are two options for building the software. The first is just like Linux:

$ git clone https://github.com/robertdavidgraham/robdns

$ cd robdns

$ make test

The second option is to use the XCode project in the subdirectory *xcode4*.

## Building on Windows

To build the software, there are two options. The first is to use Microsoft’s compiler. You can download *Visual Studio Express*, then load the workspace project in the subdirectory *robdns/vs10*.

The second option is GNU compiler in the *msys* collection, in which case the same technique of simply typing “make” as for Linux will work.

For raw socket performance, you’ll also need to install the WinPCap library from [www.winpcap.org](http://www.winpcap.org).

# Resource Requirements

## Hardware requirements

Desktop-class systems and old hardware work fine for 10-gbps operation. Large server iron is likely overkill.

Virtual machine are often limited in the raw network packet rate, so should not be used for more than 1-gbps operation.

## CPU requirements

No special CPUs are needed. Both 32-bit and 64-bit processors work fine. However, for large zones, or many zones, each in millions range, 64-bit processors will be needed to address the memory. For 1-gpbs network connections, low-power 1-ghz processors will work fine. For 10-gbps, larger multicore processors will be needed.

The software scales well with multiple cores. It’s only been tested up to 6 hyperthreaded cores, but it appears to scale well beyond that.

It’s been tested with ARM and x86 processors. It should compile for any processor type, although there may be some issues with the thread synchronization primitives for other processors.

## Memory requirements

The amount of memory needed is roughly twice the zonefile system. An 8-gigabyte zonefile for the .com domain requires about 15-gigabytes of memory. A million small zones requires about 1-gigabytes of memory.

When doing a full software update, two processes will be running side-by-side with the full zonefiles. Therefore, twice the memory will be needed to support this feature.

## Network adapter

Instead of using the sockets API to the operating-system’s TCP/IP stack, the software contains an optional custom TCP/IP stack that runs many times faster using *zero-copy* drivers. These are special drivers that DMA packets directly to/from user-mode software. Three different drivers can be used: PF\_RING, DPDK, and *netmap*. However, all of them require network adapters based on Intel chips. For 1-gbps operation (instead of 10-gbps), the *netmap* driver also supports RealTek chips.

## Supported operating systems

The software compiles and runs on most all operation systems. It is specifically tested on recent versions of Windows, Mac OX X, and Linux.

The *zero-copy* driver options are only available for Linux and FreeBSD.

## Network infrastructure

The software is designed to withstand DDoS attacks. It’s likely that the network infrastructure resources will fail before the server does. Even as low as 1-gbps, switches and routers start failing under high packet rate. DNS packets are much smaller than typical web traffic, meaning the packet rate is much higher in order to fill 1-gbps. Many network engineers have not tested for these high packet rates, so their infrastructure often fails under the load.

## Dual-homed environment

The system splits communications into *data-plane* and *control-plane*, which may require separate network adapters: one exposed to hackers and DDoS attacks from the Internet, and the second which may take a different, more protected route to the network. Ideally, the data-plane and control-plane connections are widely spaced in the data center, taking different routes to the Internet, so that an attack on the data-plane adapter will not adversely affect control-plane communications.

# Configuration

The server supports roughly the same file-format as BIND9 as well as similar configuration options. As the server can only be configured for *authoritative slave* and *secondary master* operation, only those options are supported.

## Quick testing

Instead of a configuration, the software supports easy testing on the command-line, which tries to infer intent. For example, to start running, simply type:

$ **robdns example.zone**

This will parse the indicated zonefile and start listening for queries on port 53, on both TCP and UDP, on both *0.0.0.0* and *[::]*. You can use the *dig* utility to query this zone to verify that it’s working.

The software contains its own TCP/IP stack. To test it, type something like the following:

$ **robdns example.zone eth0 10.1.2.3**

The software will infer that you want to use the *eth0* network adapter, but that instead of using the IP addresses of the operating-system’s TCP/IP stack, you want to *spoof* an unused address. You might then flood it with queries in order to benchmark the difference in speed between the operating-system’s stack and the custom stack.

## Sample configuration

The following is a sample configuration for a typical dynamically updated slave operation.

options {

directory "/var/cache/named";

listen-on port 53 { 192.168.17.254; }

listen-on port 53 { 2001:0db8:100::4; };

};

key MyKey {

algorithm HMAC-MD5;

secret "7zTrFAk8z5YP2IaHNdy0ig=="

};

zone "example.com" {

type slave;

file "example.zone";

allow-update { key MyKey; };

};

This configuration, which works the same way as for BIND9. It reads the zonefile contents from *example.zone* and starts serving queries on that zone. It allows updates from a master using the specified secret key.

The key was generated by generating random data:

$ **dd if=/dev/urandom bs=1 count=16 | base64**

7zTrFAk8z5YP2IaHNdy0ig==

16+0 records in

16+0 records out

16 bytes (16 B) copied, 6.6499e-05 s, 241 kB/s

One way of updating the records is using the *nsupdate* program from *bind-utils*. The following is an example:

$ **nsupdate –y HMAC-MD5:MyKey:7zTrFAk8z5YP2IaHNdy0ig==**

> **update add www.example.com 600 cname www1.example.com.**

> **send**

Standard zonefile format for *example.zone* file would be used, such as having the following contents:

$TTL 86400 ; 24 hours could have been written as 24h or 1d

$ORIGIN example.com.

@ 1D IN SOA ns1.example.com. hostmaster.example.com. (

2012022401 ; serial

3H ; refresh

15 ; retry

1w ; expire

3h ; minimum

)

IN NS ns1.example.com. ; in the domain

IN NS ns2.smokeyjoe.com. ; external to domain

IN MX 10 mail.another.com. ; external mail provider

; server host definitions

ns1 IN A 192.168.0.1 ;name server definition

www IN A 192.168.0.2 ;web server definition

ftp IN CNAME www.example.com. ;ftp server definition

; non server domain hosts

bill IN A 192.168.0.3

fred IN A 192.168.0.4

This example also works on 10-year old versions of BiND9. Note that there is a difference in emphasis. Most BIND9 examples control access based on IP addresses. In this example, access is controlled solely by a key – anybody who knows the secret key is able to update records.

## Signals

On UNIX, the service will respond as expected to signals. SIGHUP will reload zone information. SIGTERM and SIGINT will terminate the program gracefully.

Note that SIGHUP will only reload zonefile information and change some configuration parameters, but not all parameters. Changing everything requires a full *reload* of the process.

Note that we distinguish between *dynamic* and *static* updates. Dynamic updates are through DNS from the master server using UPDATE, NOTIFY/IXFR, and AXFR. The server overwrites the zonefiles whenever new information arrives from the master server. Static updates are when the zonefiles are edited by hand, through automated tools, or through such processes as *rsync*. Doing both at the same time will likely corrupt the data. Zones marked with “*cache dynamic*” will not be updated during SIGHUP to prevent this.

# Advanced DNS Features

## Notify

DNS *NOTIFY* [RFC1996] is a packet sent to all slaves whenever the master’s copy of the zone has changed.

As a slave, when we receive a NOTIFY from the master, we’ll initiate an IXFR transaction to retrieve the new zone information from the master.

As a secondary master, once we’ve retrieved the latest zone information, we will transmit NOTIFY messages to secondary slaves.

All NOTIFY (and IXFR/AXFR) messages must be signed with TSIG keys.

### Superslave

As defined by the RFCs, only existing zones can be updated with NOTIFY packets. However, we support the PowerDNS *superslave* extension, whereby NOTIFY packets from a trusted master can cause new zones to be created. These new zones inherit the default zone configuration for that master.

## Update

DNS *UPDATE* [RFC2136] is a packet sent to update the contents of a zone. We treat it as essentially a small NOTIFY/IXFR transaction. All the same rules apply, such as the requirement that all packets be signed with TSIG keys.

## Increment Zones Transfer (IXFR)

A zone update is triggered either by a NOTIFY from the master server, or by frequently checking the SOA serial number.

To update a zone, the server will first attempt to establish a TCP connection. If successful, it will then generate an IXFR transaction to update the zone. If IXFR fails, it will backoff and attempt an AXFR. If TCP fails, it will attempt an IXFR request over UDP.

As a secondary master, IXFR transactions are not supported. Therefore, secondary slaves will need to execute full AXFR requests. This is likely to change in the future.

## Split DNS

Split DNS is no supported, nor is it ever likely to be supported.

## TSIG

Transaction Signatures (TSIG) are required for all dynamic DNS update transactions (UPDATE, NOTIFY, IXFR, AXFR). The key (aka. password) is using the *key {}* statement in the configuration files.

A TSIG key requires two parts, a *name* and a *secret*. Virtually anything can be used as a name and a secret. Most guides for BIND9 describe using the *dnssec-keygen* utility to generate these, but that’s overkill, simply wrapping the generation of the random secret with a lot of complexity. TSIG isn’t, itself, a DNSSEC feature, so using DNSSEC tools is overly complex.

You could simply make up your own key, such as in the following configuration:

key MyKey {

algorithm HMAC-MD5;

secret "Password"

};

This will work fine, but will be horrible insecure. Transactions can be sniffed from the Internet, and hackers will be able to quickly crack the password. Remember that hackers can execute *billions* of guesses per second, so simply trying all 8 letter combinations can be done in minutes (known as *brute-force* cracking). Worse yet, hackers can do *dictionary* attacks, where they first guess common dictionary words with minor variations, where a “dictionary” consists now only of words in the English dictionary, but other common words people might base passwords on, like “GoBroncos!”.

To defeat dictionary attacks, keys should be chosen completely at random. To defeat brute-force attacks, keys should be long, at least 128-bits. A billion guesses per second of brute force cracking is equivalent to 30 bits being cracked per second, meaning 298 seconds will be needed. A billion years is 230 years. Thus, even with a billion computers, hackers would need longer than the age of the universe to crack 128-bit randomly generated keys.

On any Unix system, the way to do this would be:

$ **dd if=/dev/random bs=1 count=16 | base64**

7zTrFAk8z5YP2IaHNdy0ig==

16+0 records in

16+0 records out

16 bytes (16 B) copied, 6.6499e-05 s, 241 kB/s

The virtual file */dev/random* produces a constant stream of random binary data. The *dd* command can used to extract just the first 16 bytes. The *base64* command will convert this from binary to a string, such as *7zTrFAk8z5YP2IaHNdy0ig==* in this example.

This example uses */dev/random* in this example, but it is only *mostly* random. On Linux, */dev/urandom* is a much more secure choice.

Both the master server on our slave server need to know the same key. Therefore, you are confronted with how to do this securely. If you send over e-mail, it’s likely that evil doers will be able to intercept this. The best way is over SSH, SCP, rsync, or some other similarly secure channel.

Likewise, you should pay attention to whether other users on a system will be able to view the key within configuration files. Since it’s likely only sysadmins have access to your DNS servers anyway, this guide will assume it’s not a significant issue. However, typical BIND9 configuration files are designed so that the *secret* key can be placed in one file that only root can access, while the rest of the configuration files are world-readable, with only the *name* of the key is visible.

Thus, in our example, the configuration we use is the following:

key MyKey {

algorithm HMAC-MD5;

secret "Password"

};

This text could be imported both into master DNS server (which we will assume is a BIND9 server) and the slavee server configuration files. If the master server is so configured, it’ll transmit NOTIFY message to the slave, signed by this key, whenever changes happen.

There should probably be a unique key for each slave. Thus, when a hacker breaks into one slave DNS server, they will not be able to modify the zone information for other slaves.

In order to test that the key words, it can be used with the *nsupdate* program, as shown below:

$ **nsupdate –y HMAC-MD5:MyKey:7zTrFAk8z5YP2IaHNdy0ig==**

> **update add www.example.com 600 cname www1.example.com.**

> **send**

## DNSSEC

All DNSSEC records are supported, but none of the DNSSEC functionality. It is assumed that master is responsible for signing zones and creating correct zone data. The slaves simply echo that information.

## Zone information store

This software supports only one store of zone information: local files in *zonefile* format.

For *dynamic* zones, the software will update the information in these files as it is retrieved over DNS from the master server. They will only be read when the server restarts/reloads. For *static* zones, the software won’t change their contents, and will reload their contents whenever they change, such as on a scheduled basis, or whenever a HUP is received.

## IPv6 support

IPv6 is fully supported in every way imaginable.

One exception is the deprecated *A6* records from [RFC3363] – only AAAA records are supported.

# Configuration reference

This software uses the same rough configuration format as BIND9. Every effort has been made for backwards compatibility, so some BIND9 configuration files can be used directly. Features this software doesn’t support will either be silently ignored, generate warnings, or cause a hard failure when the configuration file is clearly wanting something important that this software doesn’t support. For example, setting *recursion yes;* will cause a warning message telling the user that this isn’t supported, but will start the server anyway since there is no harm otherwise.

## Basic configuration grammar

Whitespace and comments work as expected.

Newlines are just another form of whitespace, except when ending comments that started with // or #

All comments can start anywhere on a line, even after other text. There are three styles of comments. As is traditional, the character # starts a comment until the end of line. Like C++, the // starts a comment until end-of-line. Like C, the /\* starts a comment that can cross many lines until it ends with \*/. Some examples:

# This is the normal Unix style

// this is the normal C++ style

/\* this is the normal

C style \*/

**token {** # this part is a comment

**token;**

**};**

**token** /\*this is allowed\*/ **token { token; token; };** # so is this

Besides comments and whitespace, the file contains *statements*.

A *statement* is consists of one *token*, optionally followed by additional tokens, optionally followed by additional *statements* within braces, followed by a semicolon. A *token* consists either of something like a DNS name (letters, digits, -, \_, .), or a quoted *string* containing other characters, with double-quotes escaped in the usual manner.

quotes = %x22

token = label / string

label = 1\*(ALPHA / DIGIT / “-“ / “.” / “\_” / “[“ / “]” / “:” / “/”)

string = quotes 0\*(ANY / “\\” / “\””) quotes

statement = 1\*(token) [ **{** 0\*(statement) **}** ] **;**

Some examples:

token;

token "string";

token { token; token { token token; }; };

token "string" token {

token token;

token {

token token;

{ token; token; };

};

token;

};

A common error is to forget the semicolon.

What we refer to as a *top-level* statement is the outer statement containing everything inside. In the above example, there are four top-level statements.

The first step in reading a configuration file is a context-free parse of the top-level statement. The entire statement will be parsed according to this grammar before it’s processed. For performance reasons, each top-level statement is read one at a time, so a configuration file with a million zones doesn’t require being read into a gigabyte’s worth of memory, but instead can be processed one zone at a time.

The second step in processing a top-level is process the contents. The value of the first token determines what sorts of sub-statements can be included. Examples of top-level statements are *options {}*, *zone {}*, *zone-directory {}*, *include*, *key*.

## Global top-level statements

The top-level statements *options* and *zone* are the most important top-level statements, and are described in the own sub-chapters. This sub-chapters describes the remaining ones.

### *include* statement

The include statement looks like the following:

**include “***filename***”;**

This is an include statement of the usual form. Processing of the current file will be halted while the contents of *<filename>* will be loaded.

One reason to use include statements is to separate zone configuration from the rest of configuration information. Zones are frequently updated, whereas the other configuration information is not.

Another reason is for *separation of roles*. The system administrator has more access to the system than lesser administrators. For example, the system administrator may have access to the secret keys, whereas those administering zones do not.

Thus, a configuration file may look like the following:

include "/etc/named/keys.conf"

include "/etc/named/zones.conf"

Thus, the configuration directory may looks something like the following, where *zoneadmins* can write the zone file, read (but not write) the main configuration file, and not read the secret key file:

-rw------- 1 root root 138 Jun 14 00:47 keys.conf

-rw-r--r-- 1 root root 311 Jun 14 00:49 named.conf

-rw-rw-r-- 1 zoneadmins zoneadmins 2382 Jun 14 00:49 zones.conf

### *key* statement

Dynamic updates can only come from trusted sources, with what’s effectively a *username* and *password*. Messages are then signed using this information with *TSIG*. Since human-chosen passwords can easily be cracked, we instead use randomly generated binary keys, and copy/paste the information instead. An example of this configuration is the following:

**key** *username* **{**

**algorithm** *name***;**

**secret** *password***;**

**};**

We describe this as “username” and “password” here, as these are the terms most people are accustomed to. Officially, the username is instead called the “key name”, the “key id”, or even a “key domain name”. Officially, the “password” is known as a “shared secret key”.

The *algorithm* can be one of *HMAC-MD5*, *hmac-sha1*, *hmac-sha256*, or *hmac-sha512*. This parameter is optional. If not set, then HMAC-MD5 will be assumed. Truncated hashes are supported by appending the number of bits with a dash, such as HMAC-MD5-80.

The *secret* is BASE64 encoded. If invalid BASE64 information is included, warning messages will be generated, and the key information will not be loaded, and attempts to use the key will fail. Thus, if you wanted to use “Password1234” as a key, you would first need to BASE64 encode first in order to get UGFzc3dvcmQxMjM0.

This statement must appear at the top level of the configuration file.

For security reasons, you *should* use a separate key for each server you are talking to. That way, if one server in the system gets compromised, the damage will be limited.

## Zone statements

A zone is configured using the top level *zone* statement. Typically, a zone will have its own configuration. If not, it will inherit settings from the top-level *options* statement.

This server also supports a non-standard feature of a *zone-directory*, which contains many (possibly millions) of zonefiles, all sharing the same configuration.

### *type* statement

Two types of zones can be specified, a *master* or a *slave*. The format for this option looks like one of the following two statements:

**type master;**

**type slave;**

For this software, all zones are effectively *slaves*. The ultimate source of all zone information is from a true *master* server, and all changes can be propagated to *secondary slaves*. What this parameter instead configures is the definitive source of zone information.

For a *master* zone, the definitive source is the zonefile on the disk. The server monitors for changes to the zonefile, and will re-read it when it changes, not immediately, but after a time interval. A HUP signal can also force a reload from the zonefile.

For a *slave* zone, the definitive source is the hidden master server via dynamic DNS. The zonefile on disk is optional. If it doesn’t exist, then the server must first fetch the domain via an AXFR from the master. If it does exist on the disk, then the server will load the file, then fetch the SOA record from the master in order to see if it’s up-to-date. The slave will continue to fetch the SOA record on a regular **refresh** in order to monitor for changes, and will also response to dynamic DNS NOTIFY and UPDATE commands from the master. A slave zone must have **masters** configured in this zone, or have a default one set in *options*.

### *file* statement

The *file* statement specifies the location of the zonefile for this zone. This must either be an absolute path starting from root “/”, or a relative path to the current working **directory**.

**file “***filename***”;**

For a *master* zone, this server will never change the file. It is assumed that some other process will edit it, such as a chron job running rsync to transfer updates from the hidden master.

For a *slave* zone, this file is optional (though recommended, a warning will be produced if it’s missing). If it exists, then this server will occasionally modify the file as changes are received over dynamic DNS from the master. The slave will only load this file once at startup, but never read it again.

### *masters* statement

The slave will talk to one or more hidden masters. It can expect to receive UPDATE and NOTIFY packets. It will need to query SOA information on a regular basis, and download the latest changes with IXFR or AXFR. To do this, it needs to know the IP address (or addresses, if more than one), port (which will often be different than 53), and the key needed to talk to the master.

**masters [***name***] [port** *ip\_port***] {**

*ip\_addr* **[port** *ip\_port***] key** *username* **) ;**

**[...]**

**};**

An example of how this might look is the following:

**masters port** *1127* **{**

*10.2.3.15* **key** *master0***;**

*2001:db8:0:1::15* **port** *1128* **key** *master1***;**

**};**

A *zone* or *zone-directory* top-level statement should have a *masters* statement. If they don’t, then they will inherit the settings from the *options* statement.

### *notify*

This controls whether DNS NOTIFY messages are sent to secondary slaves.

**notify yes;**

**notify no;**

**notify master-only;**

**notify explicit;**

If **no** (the default), no message will be sent.

If **explicit**, then only those IP addresses listed in a separate **also-notify** statement will be notified.

If **yes**, then all servers listed in NS records in the zone will be notified, except the master server listed in the SOA record. The SOA master can also be notified with the separate **notify-to-soa** statement.

If **master-only** is specified, only if the zones of **type master** will send such notifications. This is useful when configuring as a default under *options*.

### *also-notify*

Specifies a list of IP address to notify whenever the zone contents change.

**also-notify {**

ipv4 **[port** number**] key [**username**];**

ipv6 **[port** number**] key [**username**];**

**};**

If specified in a *zone* statement, it completely overrides what might have been specified in the *options* statement.

### *notify-to-soa*

By default, notifications (as described in the **notify** statement) are not sent to the SOA master. This changes that behavior. This is in cases where the SOA master specifies a slave when there is a hidden master.

**notify-to-soa yes;**

**notify-to-soa no;**

## *options* Statement

The *options* statement is likely the first one in a configuration file, configuring all the essential options, like which IP addresses and ports the server will use.

**directory “***filename***”**;

**pid-file** “*filename*”;

**listen-on {** *ipv4-address***; }**

**listen-on-v6 {** *ipv6-address***; }**

### *directory*

This specifies the current working directory for the process. Files that don’t start from the root will be relative to this directory. The syntax is:

**directory “***filename***”;**

A typical example of this would look like the following:

**directory “***/var/cache/named***”;**

Unless specified with absolute paths, all zonefiles, statistics, logging, PIDs, and so on will be relative to this directory.

The filename should be an absolute path. Otherwise, the path will be relative to whichever directory the process was started from. Likewise, if this parameter is not set, the default will be “.”, the directory where the process starts.

### *pid-file*

Typical Unix services write their process ID to a file so that other processes can easily find them. An example would be a script that needs to send a HUP signal to the process in order to force a re-load of zonefiles. By default, this service writes to */var/run/named/named.pid*. This can be changed with the *pid-file* option. Its syntax is the following:

**pid-file “***filename***”;**

**pid-file none;**

A typical example of this would look like the following:

**pid-file “***/var/cache/named/robdns.pid***”;**

The contents of this file will consists of a single decimal number, like so

1387

If **none** is specified, then no PID file will be generated.

### *port*

By default, the service listens on port 53 for queries (the *data-plane* port). This port can be changed with this option.

**port** *number***;**

This is useful for testing the service, or for non-standard configurations, such as when using this software as a hidden master (which should never use standard ports).

The **listen-on** option specifying a port should probably be used instead.

### *listen-on, listen-on-v6*

This configures the *data-plane* IP addresses and ports to use for answering NDS queries from the public Internet (see the option **transfer-source** for configuring the *control-plane* address). The syntax is as follows:

**listen-on [port** number**]** **{**

*ip\_address***;**

**!***ip\_address***;**

*ip\_address/cidr***;**

**!***ip\_address/cidr***;**

**any;**

**none;**

**};**

**listen-on-v6 { ... }**

The statement **listen-on** is used for IPv4 addresses, while **listen-on-v6** is used for IPv6. They are otherwise identical.

Multiple statements may be included, and will behave as if combined into one large statement.

**listen-on {**

**192.168.0.167;**

**};**

**listen-on {**

**10.109.36.15;**

**};**

Of special note is when using *wildcard addresses*. For IPv4, a machine may have many IPv4 addresses assigned in a range. Therefore, a CIDR address, such as 192.168.0/24 can be specified. Any IPv4 addresses currently configured on the machine within that range will be used. Some, however, may be excluded. An example might be the following:

**listen-on {**

**192.168.0/24;**

**!192.168.0.25;**

**};**

For IPv6, wildcarding is built-in to the specification, so that a machine may be reachable with billions of IPv6 addresses. It may look like this:

**listen-on-v6 {**

**2001:db8::/96;**

**};**

When using raw sockets, then wildcarding is supported regardless of the machine’s true IP address. In such cases, IPv4 wildcarding works like IPv6, so that if 0/0 is specified as an IPv4 address, the server will respond to queries to any IPv4 address. (This specific configuration is probably a bad idea, because the raw stack will also respond to ARP for all IPv4 addresses).

As expected, **any** will cause the server to use all IP addresses in the system, and **none** will cause it to not use any.

Note that periodically, the IP addresses of the server may change. The server software will potentially open or close new sockets appropriately. This can be set with **interface-interval**.

### *transfer-source, transfer-source-v6, alt-transfer-source, alt-transfer-source-v6*

This specifies the *control-plane* IP addresses to use in order to communicate with hidden masters, such as doing IXFR and AXFR queries. It’s also the IP address we expect hidden masters to use in order to send us UPDATE and NOTIFY packets, rather than the *data-plane* address. The syntax is essentially the same as for the **listen-on** option. If not specified, then the default is to use the *data-plane* IP addresses – unless the data-plane uses raw-sockets, in which case, **any** will become the default.

**transfer-source [port** number**]** **{**

*ip\_address***;**

**!***ip\_address***;**

*ip\_address/cidr***;**

**!***ip\_address/cidr***;**

**any;**

**none;**

**};**

**transfer-source-v6 { ... }**

Remember that the hidden master itself must be configured to use this IP address in its configuration, such as in **also-notify** options.

More than one IP address may be specified. The server will listen on all such addresses for incoming NOTIFY and UPDATE packets. For outgoing connections, it will try the first listed IPv6 addresses first, and if they fail, will backoff and try IPv4 addresses.

### *interface-interval*

The IP addresses of a machine may change over time, as new hardware interfaces as added (such as plugging in USB Ethernet adapters), when DHCP addresses change, or when sysadmins reconfigure IP addresses. Changes are automatically scanned for whenever the configuration file reloads. The system can be configured to scan for such changes automatically as well.

**interface-interface** *minutes***;**

If this option is not specified, the default will be 60, or every hour. The maximum value is 28 days, or 40320 minutes.

If a change is detected, then the server will re-evaluate the configuration set by **listen-on** and **transfer-source** statements in order to close or open sockets as needed.

### *version*

DNS servers can be configured to return their version information upon a lookup for the name “*version.bind*” in the CHAOS class. It’s a little easter egg started by BIND in the 1980s and used by virtually all DNS software since. Should the user want to unique identify this server, then can set the response as follows:

**version “***string***”;**

**version none;**

Note that the license agreement for this server specifies that that “robns” must always be returned in such requests. Thus, any value specified in option will not replace the default “robdns” string, but will instead be appended to it.

You probably do not want to change this option, but instead use the **server-id** or **hostname** options instead.

### *hostname*

DNS servers can be configured to return their hostname with a CHAOS TXT query for *hostname.bind*. By default, this server will return nothing. This option configures something to be sent in the response.

**hostname “***string***”;**

**hostname none;**

**hostname hostname;**

The default is **none**. If the option **hostname** is used, then the name from the *gethostname()* function will be used. Otherwise, if a string is given, that will be given in the response.

### *server-id*

DNS servers can be configured to return a unique string to a CHAOS TXT query for *ID.SERVER*. They will also report then when given an NSID (Name Server Identifier), as specified in [RFC5001]. This is useful when many servers have the same IP address, such as in a load-balanced or anycasted case, so that the individual servers can be disambiguated.

**server-id “***string***”;**

**server-id none;**

**server-id hostname;**

The default is **none**. If the option **hostname** is used, then the name from the *gethostname()* function will be used. Otherwise, if a string is given, that will be given in the response.

### *allow-new-zones*

Traditional DNS servers did not allow automatic zone creation. This server does, and assumes that as the normal operation. Therefore, the default for this parameter (if not specified) is **yes**, and currently the **no** option is not supported and will generate an error.

**allow-new-zones yes;**

Possibly in the future the **no** option will be supported.

### *recursion*

This statement is for producing warning messages. It has two values.

**recursion yes;**

**recursion no;**

If *yes* is specified, then the following message will be generated:

**WARNING: recursion not supported, statement ignored.**

If *no* is specified, then the following message will be generated:

**WARNING: recursion not supported, statement is redundent.**

In either case, this server does not support recursion (aka. resolver, proxying) functionality.

### *auth-nxdomain*

This statement is provided for backwards compatibility, since a lot of examples seem to contain it. The only valid value is **no**, if another is encountered, a warning will be generated.

**auth-nxdomain no;**

## acl Statement

The *acl* statement specifies a named list of IPv4/IPv6 address ranges. Such a list can be used for a lot of purposes, from filtering incoming requests, to listing which IP addresses to notify on zone changes, to configuration which IP addresses the software should use

## zone-directory Statement

One of the intended uses of this software is for large-scale hosting environments which manage a million zones on behalf of customers. In such cases, all the zones will have the same configuration as far as slaves are concerned. Therefore, individual zones do not need separate configuration. This makes dynamic creation and deletion of zones simpler.

The zone-directory statement behaves in most respects like a zone statement, with some specific exceptions. It looks like the following:

**zone-directory “***name***” {**

**filename “***directory***”;** # the directory where files are located

**type** **[***master***|***slave***]**;

**directory-level [***1***|***2***|***3***];** # for subdividing into subdirectories

**directory-refresh [**true**|**false**];**

**};**

If the type is **master**, the default, then on startup, the server will recursively descend that directory and load all zonefiles ending in “.zone”. The zones will be created for each SOA record found. On a refresh or SIGHUP, it will repeat the process, updating zones, creating new zones, or deleting old zones as required.

If the type is **slave**, with dynamic zones, the situation is a little more complicated. As normal, the server will have to first send a query to get the SOA record from the master in order to verify that the zone is up-to-date. Since there may be millions of zones, this may take a long time. Therefore, the **directory-refresh** can be used to inform the master of the timestamp of the latest information. The master can then return a list of all zones which are out-of-date in one response. The server can then update those zones as required.

As NOTIFY packets are received from the server in superslave configuration, the server will create files as necessary.

For a lot of zones, a two-level, or even three-level, directory structure should be used. For static **master** zones, the software will recursively descend any number of levels. For dynamic **slave** zones, it will hash the name and create either one or two levels, each with 100 subdirectories. The hash creates a directory name between 00 and 99. If there are a million zones, this should create at the leaf directories with 100 zones. The **directory-level** statement will specify the number of levels in the hierarchy that will be created.

# Zonefiles

All the DNS information is stored on disk in the form of *zonefiles*.

In the static **master** zones, external processes like *rsync* will be updating these files in order to reflect the latest zone information from the server. Or, they can be edited by hand.

For dynamic **slave** zones, the zonefiles will be read once on startup, and then kept consistent with dynamic updates from the hidden master, so that the next time the service starts up, it won’t have to download all that information again from the master.

This software doesn’t have a custom binary format. However, that’s okay because the parser is extremely fast. In most cases, this software will still load the text zonefiles faster than other servers can load their binary formats – simply because zonefiles are smaller, so the hard drive (including SSDs) transfer rates are the limiting factor.

The zonefile format is effectively described in RFCs. Each resource-record type is specifies both the *wire-format* as the packet is transmitted across the Internet, and a *zonefile-format* for storing in zonefiles.

# Journaling

Dynamic updates to **slave** zones (via UPDATE, IXFR, and AXFR) must be first saved as *journal* files. This is so that if the server crashes during an update, that no data becomes corrupted. It is also important during the *server reload* process, when the new server takes over from the old ones, all updates to zonefiles must stop and the journaling must take over, until the new server is ready to assert control.

This server simply uses the wire format for these files, saving the raw DNS paylods directly to disk.

# Operational security considerations

## Data-plane adapters/addresses

## Control-plane adapters/addresses

## Chroot and setuid

## Dynamic DNS Security

Other systems want to control this IP address.

This server only supports TSIG. TSIG is mandatory in all dynamic DNS packets.

# Comparisons with other DNS servers