Setting up an email server in 2020 with OpenSMTPD and Dovecot

So, you want to set up your own email server? In that case, welcome.

There are many reasons to run a custom email server, ranging from privacy concerns about providers like Google, to just wanting to do it for fun and/or learning. Since you're here, I assume you've already found a reason.

Beware: this is a messy topic, and the available documentation is even messier, so it could take a while before you get it to work properly. I've compiled this guide according to my experiences in an attempt to make this dark art more accessible, but your mileage may vary considerably. I hope you find it useful.

This guide is aimed at people who are comfortable with the Linux/*BSD command line.

When you're done (if you get that far), take a look at the sequels "Setting up an email server in 2020 with OpenSMTPD and Dovecot: extras" and "Revisiting my email server in 2022" for ideas on how to extend your setup.

Last updated on 2022-09-12.

Preparation

Setting up email is relatively complex compared to e.g. a static website, because you need to configure not one, but *two* server programs, *and* you need to shoehorn modern security features into email's Stone-Age design. I'll start by explaining the general structure of a mail server setup.

How email works

The programs involved in the exchange of emails are called agents. Officially, there are 5 different types of agent: MUA, MSA, MTA, MDA and MRA. But fortunately, it's reasonable to treat the MRA and MSA as being part of the MUA and MTA, respectively.

The *Mail User Agent* (MUA) is simply the client on your device at home that you use to send and receive emails, and this guide assumes you already have a favourite program for this, e.g. Thunderbird. Nowadays it's fashionable to use a web interface for emails, but that's also beyond the scope of this guide.

The *Mail Delivery Agent* (MDA) is a program that watches over the server's copy of your mailbox: it manages your inbox, remembers which messages you have or haven't read, keeps a copy of your drafts, etc. When you open your mailbox, your MUA will connect to your server's MDA using the IMAP protocol (or POP3, but that one's obsolete).

The *Mail Transfer Agent* (MTA) is responsible for making messages arrive at the right destination. When you send an email, your MUA will pass it on to your server's MTA, which will in turn pass it on to the recipient's mail server. Likewise, when someone sends *you* an email from another server, the MTA will receive it and hand it over to the MDA so you can read it later. In both cases the MTA speaks the SMTP protocol.

In this guide our MDA will be Dovecot, which is a very popular choice for that role. As for the MTA, there exist several options, the most popular being Postfix and Exim. However, this guide uses the newer, lesser-known OpenSMTPD, which in my experience is *much* easier to set up: Postfix and Exim have complex configurations and are geared towards large-scale email providers, whereas OpenSMTPD is more beginner-friendly.

Security

The base email system is horribly insecure on its own, so we still need to duct-tape on some security features. In this context, "security" has two meanings: spam protecion and privacy protection (encryption).

Spam protection also means two things here: defending yourself against spammers, and preventing that *your* emails get flagged as spam. The former is optional, but the latter is not: big providers such as Google and Microsoft use infamously strict spam filters, and if they decide that

your server is a spammer, there's almost nothing you can do about it. Spam protection techniques will be discussed in more detail over the course of this guide.

Privacy protection is important in the 21st century: you don't want a random router in the Internet to read all your emails, which may contain sensitive information such as private conversations and account password reset links. You should therefore try to make sure that emails are transported over an encrypted channel. To do this, you have two options for encryption: *mandatory* and *opportunistic* encryption.

Mandatory encryption is only practical for client-server communication (not server-server), and is provided by IMAPS and SMTPS, which wrap the IMAP and SMTP protocols in TLS, in the same way that HTTPS does for HTTP.

For server-server communication, the only option is opportunistic encryption in the form of STARTTLS, where communication is only encrypted if both parties agree after a short unencrypted discussion. That last part is vulnerable to MitM attacks, where anyone along the path of the email servers' discussion can alter the exchange to block the use of encryption, which sometimes actually happens in practice.

The only way to make sure that STARTTLS is used in that case is to refuse any exchange unless the servers agree to use encryption. Unfortunately, that's a risky approach that I can't recommend, because not all servers support encryption (unbelievable, right?). For example, I've received airline booking confirmations, full of personal details, and made with billion-dollar companies, sent across the Internet without any protection.

This guide includes intructions to enable encryption, but assumes that you already have a TLS certificate for that. If not, find a guide to get one from Let's Encrypt (it's free!), and remember that you'll need to renew it every few months. Using a self-signed certificate *may* work, but I don't recommend it.

In the rest of this guide I'll assume that you have a public full-chain TLS certificate at `/etc/ssl/certs/example.com.pem`, and a private encryption key at `/etc/ssl/private/example.com.pem`.

Server

Obviously, you'll need a server to run the MTA and MDA on. You can host your own at home, but the more reliable option is to rent one in a data center (VPS). This guide was written with a Linux server in mind, but in theory it should also work on the BSDs (OpenBSD, FreeBSD, NetBSD, etc.) with minimal adaptation.

The server must be online 24/7, you must have root SSH access, it must have a static IP address, and TCP network port 25 must be open. Especially check that last one: you may need to explicitly ask your home ISP or the server provider to enable port 25, because they often close it to prevent spam. You can usually do this from their web interface.

You also must have a domain name, which I'll call `example.com`. This will be necessary for basically everything: DNS records, TLS certificate, MTA network configuration, etc. If you don't have one yet, you can choose between many registrars to rent one from. Personally I use and can recommend Gandi.

Note that it's a **bad** idea to use a domain like `foo.bar.com`, where you control the `foo` part but *not* the `bar` part: in that case, a spammer in control of `qux.bar.com` could negatively affect *your* reputation in the eyes of other email providers.

Lastly, when setting up an email server, you also have the choice between using to *system* users or *virtual* users. With system users, if an email arrives for `john@example.com`, then the MTA and MDA will expect that there exists a `john` Unix user on the server to deliver it to. With virtual users, you have much more flexibility, so that's what we'll use. All email will be managed under a single Unix user/group called `vmail`. Create it as follows:

```
# GNU CoreUtils:
$ groupadd vmail
$ useradd -g vmail vmail
# BusyBox:
$ addgroup vmail
$ adduser -D -G vmail vmail
# *BSD:
$ no clue, but it should be similar
```

DNS records

Now we must set up all the necessary DNS records, which is usually possible from the domain registrar's web interface. It may take a while for your changes to propagate over the Internet, so I recommend doing this section now and the rest tomorrow.

Firstly, you should already have an A and/or AAAA record to associate your domain `example.com` with the server's IP address. For email it is **essential** that you also have reverse DNS set up correctly. If you're renting your server remotely, you can often do this from the provider's configuration tool, otherwise, you should create a PTR-type DNS record, although that's beyond the scope of this guide.

Once you're done, I recommend testing your DNS records using the MX Lookup online tool.

MX

To inform the rest of the Internet that your server is an email server, create an MX (Mail eXchanger) DNS record for your domain. Note the dot at the end of the domain name:

```
example.com. MX 42 example.com.
```

When a message is sent to an email address ending in `@example.com`, the sender will query DNS for any MX records for `example.com`. There it will find a domain name (in this case `example.com` again), for which it will look up the IP address using an A/AAAA record. The domain name in the record must **not** have an associated CNAME record; it must be directly translatable to an IP address.

You may have multiple MX records, containing different domain names, each with a preference number (`42` in the example above). The sender will try MX records with *lower* numbers first, and if that server is unavailable, it will try a higher number. If you have multiple mail servers (which is a good idea), you can thus declare those as follows:

```
example.com. MX 13 mx1.example.com.
example.com. MX 42 mx2.example.com.
```

Here, a server sending an email to your domain `example.com` will try to send it to the IP address of `mx1.example.com` first, and if that fails, it will move on to `mx2.example.com`. If both `mx1` and `mx2` have the same number, then the sender will randomly choose one, which is useful for load balancing, although that's probably overkill for a private server.

SPF

The Sender Policy Framework (SPF), is a feature which helps prevent spammers from impersonating your server in an attempt to get around blacklists. This security feature is **required** nowadays: if you don't use it, you'll probably get flagged as spam.

SPF works by specifying which IP addresses are authorized to send emails from your domain name. You must publish this information in a TXT-type DNS record (**not** SPF-type, which also exists!) with the following contents:

```
example.com. TXT "v=spf1 mx -all"
```

Everything after the version `v=spf1` is a list of *verification mechanisms* for a spam filter to try out in the given order. The `-all` at the end says to reject your email if all of the previous mechanisms fail. See the SPF spec for details.

I recommend only using the `mx` mechanism, which tells the verifier to look at the A/AAAA addresses of the domains in your MX records. This allows you to add, remove, or change your servers without needing to update this record.

DKIM

Then we have DomainKeys Identified Mail (DKIM), which is a more comprehensive form of anti-impersonation, and, like SPF, is practically **mandatory** in the modern era.

It adds a cryptographic signature to all emails from your server, which the receiver's spam filter will verify using the email's contents, and a public key that you need to publish in a DNS record. Again, you should implement *both* SPF and DKIM, despite their overlap.

To set up DKIM, create an RSA keypair, using the `openssl` utility:

```
$ openssl genrsa -out /path/to/dkim/private.key 2048
$ openssl rsa -in /path/to/dkim/private.key -out /path/to/dkim/public.key
```

The minimum size is 1024 bits, but I recommend 2048 bits. Bigger is better, but because DNS is involved you can't stretch it to 4096 bits without causing discomfort to some servers. And I think it goes without saying that you should keep the private key private.

Importantly, the DKIM DNS record **cannot** be attached directly to your domain `example.com`; instead, it should belong to a subdomain of the form `<selector>._domainkey.example.com`, where `<selector>` is an alphanumeric string you can choose (e.g. today's date), just remember your choice for later when configuring the DKIM signer. And if you change your key, keep the old record around for a while so old emails can still be verified.

Your DKIM policy must be published in a TXT record as follows, where `<pubkey>` is the public RSA key `MI...AB` stored in `/path/to/dkim/public.key`, with the newlines removed:

```
<selector>._domainkey.example.com. TXT "v=DKIM1; t=s; h=sha256; p=<pubkey>"
```

Here, `v=DKIM1` is the version and must be the first tag. The flag `t=s` enables strict mode, as recommended by the DKIM spec, meaning that emails sent from subdomains immediately fail verification. The optional tag `h=sha256` blocks the use of the old SHA1 algorithm.

DMARC

Lastly, we have Domain-based Message Authentication, Reporting and Conformance (DMARC), which is *technically* optional, but *highly* recommended, because it will make you look more legitimate in the eyes of Google and Microsoft. It can modify the behaviour of SPF and DKIM, and also provides advice about what a receiver should do if one of your emails fails verification.

To enable it, create yet another TXT record, which, similarly to DKIM, **must** belong to the subdomain `_dmarc.example.com`, and give it the following contents, where `<admin>` is an email address of your choosing, which may or may not belong to your domain:

```
_dmarc.example.com. TXT "v=DMARC1; p=reject; sp=reject; pct=100; aspf=s; adkim=s; fo=1;
```

The version tag `v=DMARC1` must come first, followed by `p=` and `sp=`, which control what to do to unverified messages coming from the main domain and subdomains, respectively. Here, `reject` means that delivery should be refused, `none` asks to let it through anyway, and `quarantine` tells

the filter to take a closer look or to put it in a spam folder. The percentage `pct=100` says how many of your emails to apply the policy to. Next, `aspf=s` and `adkim=s` enable strict mode for SPF and DKIM, which blocks subdomains from passing. Finally, `fo=1` asks for a forensic report if verification fails, and `ruf=` gives an address to send it to. If in doubt, see the DMARC spec.

MDA: Dovecot

Dovecot is a very popular IMAP server, focused on being lightweight, simple, and secure, and has extensive and up-to-date documentation. It's very flexible and scalable, and keeps up well with the lastest security best-practices.

If you installed Dovecot via a package manager, you'll probably have lots of configuration files in the `/etc/dovecot` directory. I want you to delete all of them. Yes, `rm -rf` that crap. Dovecot is simple to configure, and doesn't care where you put its settings, so having all that chaos in `/etc/dovecot` just makes things unnecessarily confusing.

Network

Create a new configuration file `/etc/dovecot/dovecot.conf`, and start by filling in the details of your TLS certificate, making clear that unencrypted connections are unacceptable:

```
ssl = required
ssl_cert = </etc/ssl/certs/example.com.pem
ssl_key  = </etc/ssl/private/example.com.key

ssl_min_protocol = TLSv1.2
ssl_prefer_server_ciphers = yes

disable_plaintext_auth = yes
```

The final `disable_plaintext_auth` option tells Dovecot to reject any passwords that were sent unencrypted. This means it must be hashed or sent over an encrypted connection, or both.

Next, tell Dovecot which protocols to use and where to expect them as follows:

```
protocols = lmtp imap

service lmtp {
        unix_listener lmtp {
                user  = vmail
                group = vmail
        }
}

service imap-login {
        inet_listener imap {
```

```
                port = 143
        }
        inet_listener imaps {
                port = 993
        }
}
```

LMTP is the Local Mail Transport Protocol, which is basically SMTP but for exchanges within a single server or over a trusted network. When an email is received, Dovecot will start a child process under the `vmail` user/group to deliver the message to its recipient.

Since we set `ssl = required` earlier, clients will only get their mail if the STARTTLS handshake was successful during the IMAP exchange, or if they connect via IMAPS to force the use of encryption. You can therefore optionally remove one of the two `inet_listener`s according to your preferences.

Users

Next, we need to inform Dovecot which email addresses it should handle, and what to do with their messages. Create a file `/etc/dovecot/users` for this, which describes a user on each line in a similar format as `/etc/passwd`:

```
user:password:uid:gid::homedir
```

In this guide, we're using the `vmail` user for all accounts, so leave the `uid`, `gid`, and `homedir` fields blank. We'll be storing all emails in `vmail`'s home directory. The `user` field should be the email address excluding the `@example.com` (in fact, you *can* include it, but this guides assumes a small-scale server managing only one domain, so we exclude it). Create the password hash to put in the `password` field as follows:

```
# If your server is fast and has lots of RAM:
$ doveadm pw -s ARGON2ID-CRYPT
# If you're using a potato:
$ doveadm pw -s SHA512-CRYPT
```

After you've entered your password, simply copy-paste the entire hash string outputted by the program into the `password` field.

Now, Dovecot needs a file describing user accounts on two separate occasions:

- To check whether a client logging into the IMAP server is valid and has given the right password. This is handled by the `passdb` block(s) in the configuration.
- To know which email addresses the server is responsible for. This is given by the `userdb` block(s) in the configuration.

These functions aren't necessarily fulfilled by the same `users` file: you can map multiple email addresses to one acccount, or multiple accounts to one email address. For simplicity, though, we'll use the `users` file for both:

```
passdb {
        driver = passwd-file
        args = scheme=ARGON2ID-CRYPT username_format=%n /etc/dovecot/users
        #args = scheme=SHA512-CRYPT username_format=%n /etc/dovecot/users
}

userdb {
        driver = passwd-file
        args = username_format=%n /etc/dovecot/users
        override_fields = uid=vmail gid=vmail home=/home/vmail/%n
}
```

The `driver` option sets the kind of table Dovecot should expect. We tell it to use a file in the `passwd`-like format described above, but other possibilities include e.g. an SQL database. The options available in `args` depend on the chosen `driver`.

In the `passdb` block, the hashing algorithm is given by `scheme`, while `username_format=%n` says that the `users` file only contains `name` out of `name@example.com`.

In the `userdb` block, we force the use of `vmail:vmail` for all accounts, and tell Dovecot to put their data in `/home/vmail/%n`, where `%n` means the address up until the `@`, so e.g. mail to `name@example.com` is stored in `/home/vmail/name`.

Now that Dovecot knows where to store messages, we just need to specify in what format to store them:

```
mail_location = maildir:~/Maildir
```

The two standard mailbox formats to choose from are `maildir` and `mbox`. I highly recommend `maildir`; it's more modern than the ancient `mbox` format. That `~/Maildir` says which subfolder to use. Ensure that `/home/vmail` is owned by `vmail:vmail`, so that Dovecot has write access.

MTA: OpenSMTPD

OpenSMTPD is an MTA by the OpenBSD project, who are known for their focus on security and minimalism. Compared to other MTAs it's a joy to set up, thanks to its intuitive configuration syntax and to-the-point manual. This guide is for OpenSMTPD version 6.4 or newer: older versions used a substantially different syntax.

If you have any problems with OpenSMTPD, take a look at the maintainer's blog, which contains a lot of useful information, and was a big help when writing this guide.

Users

To begin, delete the contents of the `/etc/smtpd/aliases` file, if it exists, which maps recipient addresses to system users. In our case, we're using `vmail` for everyone, and we'll let Dovecot manage the details by simply writing:

```
@ vmail
```

Then create a new file `/etc/smtpd/passwds` and fill it in according to the following format, where `<user>` could be anything you want, not necessarily the same account name as in Dovecot:

```
<user> <hash>
```

Generate the password hash with this command for each user. Again, the password doesn't need to be the same as in Dovecot, but for your own convenience it's probably best if it is:

```
$ smtpctl encrypt '<password>'
```

Then, like with Dovecot, just delete the contents of the main config file `/etc/smtpd/smtpd.conf`, and start by putting in the following:

```
table aliases "/etc/smtpd/aliases"
table passwds "/etc/smtpd/passwds"
```

Network

Write your domain name on a single line in `/etc/smtpd/mailname`. This is the name that OpenSMTPD will use to introduce itself to other servers, and it's important that this matches the reverse DNS domain name of the server's IP:

```
example.com
# Or mx1.example.com or whatever
```

Then continue in `/etc/smtpd/smtpd.conf` by importing your TLS certificate:

```
pki "example.com" cert "/etc/ssl/certs/example.com.pem"
pki "example.com" key "/etc/ssl/private/example.com.key"
```

And tell OpenSMTPD which keys to use for the Sender Rewriting Scheme, which prevents forwarded emails from breaking SPF and looking like spam:

```
srs key "<secret1>"
srs key backup "<secret2>" # optional, read below
```

It's recommended to change the key every year or so, but in that case you need to ensure that emails from the last month can still be verified using the old one, so if you change it, simply move it to the backup slot for a month.

It's very important that the secrets can't be guessed, otherwise *anyone* can send mail through your server, so I recommend generating these keys randomly as follows:

```
$ head -c 30 /dev/urandom | base64
```

Next, define the spam filters as follows. For the last line to work, you'll need to download and build the `filter-rspamd` adapter binary for yourself, created by OpenSMTPD's official maintainer. We'll set up the Rspamd service in the next section, which will be responsible for spam filtering and DKIM signing:

```
filter   "rdns" phase connect match   !rdns disconnect "550 DNS error"
filter "fcrdns" phase connect match !fcrdns disconnect "550 DNS error"
filter "rspamd" proc-exec "/etc/smtpd/filter-rspamd"
```

I cannot overstate the importance of the first two lines: these will block hundreds of spam attempts, and have, at least for me, never blocked anything legitimate so far.

The only thing left to do here is to tell OpenSMTPD which ports to listen on and what to do with the incoming traffic. This comes in two parts: first, we listen on port 25 for incoming messages for your domain coming from other email servers:

```
# Inbound
listen on eth0 port 25 tls pki "example.com" filter { "rdns", "fcrdns", "rspamd" }
action "RECV" lmtp "/var/run/dovecot/lmtp" rcpt-to virtual <aliases>
match from any for domain "example.com" action "RECV"
```

Line 1 says to listen on `port 25` of interface `eth0`, providing *optional* `tls` using the certificate for `example.com`, and passing everything through the three filters definer earlier. Making TLS mandatory is a **bad** idea here, because not all servers can use TLS.

Line 2 defines an action called `RECV`, which relays an email to Dovecot's LMTP socket, with the `rcpt-to` and `virtual` making sure that Dovecot will actually accept the message.

Line 3 then simply says that any incoming mail for your domain should have the action `RECV` applied to it, unless the spam filters rejected it.

Secondly, we listen on port 465 (SMTPS) and/or port 587 (STARTTLS) for messages getting sent from your email client to the rest of the world, so we require user authentication:

```
# Outbound
listen on eth0 port 465 smtps       pki "example.com" auth <passwds> filter "rspamd"
listen on eth0 port 587 tls-require pki "example.com" auth <passwds> filter "rspamd"
```

```
action "SEND" relay srs
match from any auth for any action "SEND"
```

The only difference between lines 1 and 2 is the port and the protocol. Both demand a mandatory TLS connection, and that users authenticate themselves according to the password file created earlier. The `filter` at the end is for DKIM signing of outgoing mail.

Lines 3 and 4 are only triggered after successful `auth`entication, and, unsurprisingly, `relay` the email to its destination. The `srs` at the end enables using the SRS settings from earlier.

OpenSMTPD is the only program in this setup that needs to handle untrusted connections, when other MTAs send you messages. Since, like most server software, it may have vulnerabilities, it is **very** important that you keep it as up-to-date as possible.

Spam filter: Rspamd

Rspamd is a modern spam filtering solution, with many features that you could spend hours tweaking. In this guide, however, we'll keep it short, because we're only really interested in its ability add our server's DKIM signature to outgoing messages.

Besides, according to my limited experience, for small-scale servers spam filtering isn't essential, as long as you tell OpenSMTPD to check reverse DNS as described above. However, there are some much more experienced people who disagree with me.

Basically, for our purposes, don't touch any of Rspamd's default configuration except for creating/editing the file `/etc/rspamd/local.d/dkim_signing.conf` with the following contents, where `<selector>` is the DKIM selector you chose in the DNS record:

```
allow_username_mismatch = true;

domain {
        example.com {
                path = "/path/to/dkim/private.key";
                selector = "<selector>";
        }
}
```

Make sure that the DKIM `private.key` file is readable (and *only* readable) by `rspamd:rspamd`. Allowing username mismatches is necessary, because OpenSMTPD will only tell Rspamd about `username` while the DKIM signer actually expects `username@example.com`.

And… that's it! Of course, don't forget to start all the necessary daemons.

Testing

Everything is set up now, so it's time to test. Fingers crossed!

As mentioned earlier, you can check the validity of your DNS using the MX Lookup tool. You can use the same website to test OpenSMTPD with the SMTP Diagnostics tool.

All decent email clients include an option to set the server for an email account. This guide doesn't include instructions for that, because it will vary a lot from client to client. If asked for your login type, choose plain/normal/unencrypted passwords. Don't worry, the client-server connection is TLS-encrypted, so nobody will be able to steal it.

Next, to verify that you can send and receive messages, and that your SPF/DKIM/DMARC is working, the following websites will be useful:

- ~~Is my email working?~~ UPDATE: no longer available.
- DKIM validator: by sending an email, checks that SPF and DKIM are set up correctly.
- Email deliverability tool: tests basically everything. As of writing, there's a bug that causes DKIM signatures to fail verification even if you did it right; just ignore that.
- Learn and test DMARC: checks SPF, DKIM and DMARC in detail.

If everything is good so far, congratulations!

Now comes the big scary final test: sending an email to one of the "big guys", like Google, Yahoo or Microsoft. Their spam filters are very strict, so if you get through, great! Try to write your test message so that it doesn't look like spam, otherwise there's no point to this exercise.

If something failed, then you have some investigating to do. Either one of the daemons is misconfigured, or there's a problem with your domain name and/or DNS records. Do some research, and you'll get there, don't give up.

But if everything works, congratulations! You're now the proud administrator of a private email server. Have fun with it, and don't forget to update your TLS certificate and DKIM and SRS keys.

PS: and please don't spam; you'll ruin it for everyone else.

 Visitor statistics