Frank Seifferth

Using delta chat keys with notmuch

22nd October 2025

End-to-end encrypted messaging has long become the new normal for most people. Especially on mobile. Unfortunately, many popular messaging apps, such as WhatsApp or Signal, are designed as centralized systems where a single vendor controls not only the most commonly used client application, but where that same vendor also controls the entire server infrastructure. It comes as little surprise, then, that all those messaging apps suffer from excessive degrees of vendor lock-in. Even the interoperability requirement found in the EU's Digital Markets Act appears to have been a non-starter so far, as well-funded legal departments seem to be rather successful at designing processes that are sufficiently cumbersome as to ensure that the possibility of having third parties interoperate with these systems remains entirely theoretical.

A particularly promising proposal on how to change this state of affairs comes from the developers of Delta Chat. As far as the user interface is concerned, their application is pretty similar to those other messaging apps mentioned above. When it comes to message delivery, however, instead of adding yet another

Permalink: https://tilde.club/~seifferth/blog/delta-chat-keys/

walled garden to the existing messaging landscape, Delta Chat simply relies on good old email. This decision does not only mean that with Delta Chat vendor lock-in becomes very much of a non-issue. By relying on what has been rightly described as the largest decentralized messaging system on the planet, the number of people one can reach with Delta Chat rivals — to say the least — the number of people one can reach using even the most popular of the walled garden solutions. Considering the paramount importance network effects have for the appeal of any kind of messaging solution, this ingenious move might very well be our best bet at ensuring that text-based communication does not become the private monopoly of any single company.

As has been noted before (Marlinspike 2016; Delta Chat Team 2025), rolling out new features in federated ecosystems can be somewhat harder than rolling them out in a centralized system. While this seems to be a major hurdle for the more wide-spread adoption of end-to-end encrypted email, however, there have still been a number of significant advances in this area in recent years. With WKD (Koch 2016–2025) and Autocrypt (Autocrypt Team 2020) there are now two highly convenient ways of automating the distribution of OpenPGP keys; and the example of Delta Chat shows quite clearly that end-to-end encrypted email can be just as approachable as any other messaging system. At the same time, Delta Chat's commitment to open standards like Autocrypt, OpenPGP, and email ensures that the messages themselves are not tied to a single messaging app. If I export my private key using an Autocrypt Setup Message, I can easily migrate to a different email client at any time. I can even use Delta Chat as one among multiple different email clients in a multi-device setup.

In this blog post, I go through the details of how I configured my own multi-device setup, which consists of using Delta Chat on my phone and a notmuch-based command line setup on my laptop. I first explain how I imported the private key used by Delta Chat into my gpg keyring in order to make it available to notmuch and bower. I then show how I use '`notmuch search`' to access public keys found in Autocrypt headers that may or may not have been generated by Delta Chat. Finally, I show how I published my public key to posteo's WKD, which does not only allow other people to retrieve it using, e.g., '`gpg --locate-keys`'; but which also causes posteo to add an Autocrypt header to all outgoing

emails that do not yet contain one.

Importing the private key

Disclaimer: Private keys should be considered sensitive information. When you store your private key's encryption passphrase in a file, you may want to take certain precautions, like ensuring that this file is not saved to disk in an unencrypted manner. I myself usually '`cd /tmp`' before performing operations like those described below. Since '`/tmp`' is a ramdisk on my system, any data stored there is permanently lost shortly after the computer is powered off. You may want to take similar precautions. You may also want to ensure that your gpg keyring and notmuch's xapian index are properly secured before you use them to store possibly sensitive information.

When a new account is added to Delta Chat, the app will automatically create a cryptographic identity by generating an ed25519 primary key that can be used for signing messages and a cv25519 subkey that can be used for encryption. The private key material for this identity can then be exported as an Autocrypt Setup Message. In the Android app, this functionality is available via '**Settings** — **Advanced** — **Send Autocrypt Setup Message**'. After confirming that one really wants to create this message, the app will display a passphrase that looks roughly like this:

```
9503 - 1923 - 2307 -
1980 - 7833 - 0983 -
1998 - 7562 - 1111
```

Since these numbers are used to encrypt the private key material, you should make sure to write them down before they disappear. To that end, let's create a simple text file called '`passphrase.txt`' where we store these numbers like this:

```
9503-1923-2307-1980-7833-0983-1998-7562-1111
```

That is, we include both the numbers and the hyphens, but we ignore any whitespace and line breaks. We could also add a final newline character to the end of this file, but we do not have to do so. Next, we check our email inbox where we should find

a message with a subject of 'Autocrypt Setup Message' and an attachment of type '`application/autocrypt-setup`' that looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Autocrypt Setup Message</title>
  </head>
  <body>
    <h1>Autocrypt Setup Message</h1>
    <p>This is the Autocrypt Setup Message used to [...]
    <pre>
-----BEGIN PGP MESSAGE-----
Passphrase-Format: numeric9x4
Passphrase-Begin: 95

[[ ascii-armored pgp message ]]
-----END PGP MESSAGE-----

</pre>
  </body>
</html>
```

If we store this attachment as '`setup-message.html`', we can import our private key into the gpg keyring by running

```
$ cat setup-message.html |
      gpg --batch --decrypt \
          --passphrase-file passphrase.txt |
      gpg --import
```

Finally, we can confirm that the key was added correctly:

```
$ gpg --list-secret-keys
[...]
sec   ed25519 2024-11-25 [SC]
      B280419AA8EE19B8645C8FDECAC590E4BF6815AE
uid           [ultimate] <frankseifferth@posteo.net>
ssb   cv25519 2024-11-25 [E]
```

Configuring notmuch and gpgsm

Recent versions of notmuch come with both a '`--decrypt`' and a '`--verify`' flag that will cause notmuch to use gnupg for message decryption and signature verification. Adding the private key to the gpg keyring should therefore be enough to allow notmuch to handle end-to-end encrypted email. By default, notmuch will not index the content of end-to-end encrypted messages as this would imply adding parts of the decrypted content to the xapian index. If your xapian index is properly secured, however, you may want to make notmuch index the decrypted content by running '`notmuch config set index.decrypt true`' (Amarilli 2018).

One thing to keep in mind when using '`notmuch --decrypt`' is that this flag does not only cause decryption but that it also implies signature verification; and that gnupg does not only provide '`gpg`' for handling OpenPGP messages but that it also provides '`gpgsm`' for handling S/MIME. Using '`notmuch --decrypt`' can therefore also trigger the verification of S/MIME signatures, which, in turn, can cause '`gpgsm`' to perform an online lookup of S/MIME certificate revocation lists (CRLs). These CRL lookups effectively constitute a backchannel (Poddebniak et al. 2018, 559) that can have privacy implications similar to those of Tracking Pixels. Furthermore, those online lookups introduce notable latency that can become particularly painful in offline settings. To solve these issues, I simply disabled CRC checks — as well as any other kind of S/MIME-related online lookup — in the '`gpgsm`' config file:

```
# ~/.gnupg/gpgsm.conf
disable-crl-checks
disable-dirmngr
```

Importing public keys

Key distribution is well known to be a hard problem. One thing that both Delta Chat and a number of other email clients — such as Thunderbird, K-9 Mail, and Mutt — do to help solve this problem is to add a special Autocrypt header (Autocrypt Team

2020) to all outgoing messages. This header looks roughly like this:

```
Message-ID: <694cdfce-0505-4431-81d0-b203a7dd4565@[...]>
Autocrypt: addr=patrice.lumumba@example.net;
    keydata=[[ base64-encoded public key, which may
    continue over multiple lines ]]
Content-Type: multipart/mixed; boundary="[...]"
```

Since the Autocrypt header simply contains a base64-encoded OpenPGP public key, importing this key into the gpg keyring is pretty straightforward:

```
$ cat <<EOF | base64 -d | gpg --import
[[ base64-encoded public key, which may
continue over multiple lines ]]
EOF
gpg: key 139563682A020D0A: public key "<patrice.lum[...]
gpg: Total number processed: 1
gpg:               imported: 1
```

To make the process a little more convenient, we can also use '`notmuch search`' to automatically import the public key of, say, 'patrice.lumumba@example.net':

```
$ notmuch search \
          --output=files \
          --sort=newest-first \
          --limit=100 \
          from:patrice.lumumba@example.net |
      xargs awk \
          'BEGIN { output=0 };
           /^Autocrypt:/ { print; output=1; next };
           output==1 && /^[ \t]/ { print };
           output==1 && /^[^ \t]/ { exit };' |
      tr -d '\n \t' |
      sed 's,^.*keydata=,,' |
      base64 -d |
      gpg --import
gpg: key 139563682A020D0A: "<patrice.lumumba@exampl[...]
gpg: Total number processed: 1
gpg:              unchanged: 1
```

Note that this command will simply extract the OpenPGP key found in the most recent Autocrypt header sent by 'patrice.lumumba@example.net' and pipe it into '`gpg`'. It does not verify that this OpenPGP key specifies the same address in the User ID Packet (Wouters et al. 2024, 70). A malicious actor could theoretically abuse this missing verification to distribute public keys that are associated with other peoples' email addresses. Since the Autocrypt standard also specifies an optional 'Key Gossip' extension (Autocrypt Team 2020, 10–11) that is explicitly designed for distributing public keys that are associated with other peoples' addresses, this shortcoming does not seem particularly perilous. Still, if '`gpg`' reports importing a key associated with a different address than the one you expect, you might want to manually inspect the relevant emails' source code in order to see what is going on.

Adding the autocrypt key to my email provider's wkd

Another way of distributing OpenPGP keys is the Web Key Directory standard (Koch 2016–2025). While Autocrypt relies on in-band key transmission, with Web Key Directory it becomes the email provider's responsibility to distribute keys via a '`.well-known`' path on their website. This means that unlike Autocrypt, which relies entirely on client-side support of the standard, WKD is only available if the email provider actually offers this functionality. Luckily, my own email provider, posteo, does offer a WKD; even if their implementation seems to be a little out of sync with the current version of the standard's draft.

According to the WKD standard, the Web Key Directory for posteo.net should be located either at https://openpgpkey.posteo.net/.well-known/openpgpkey/posteo.net/ or — if this domain does not exist — at https://posteo.net/.well-known/openpgpkey/. So as a first step, let's check if openpgpkey.posteo.net can be resolved via DNS:

```
$ dig +noall +answer openpgpkey.posteo.net
```

Since this domain does not seem to exist, we can directly

query the second location to retrieve some basic information about the WKD:

```
$ wkd=https://posteo.net/.well-known/openpgpkey
$ curl $wkd/policy
mailbox-only
auth-submit
$ curl $wkd/submission-address
keys@posteo.de
```

What this tells us is that posteo expects a minimized OpenPGP key that only contains an email address but no display name (‘`mailbox-only`’); and that adding this key to the WKD is as simple as sending an email to ‘keys@posteo.de’. Since posteo already performs authentication for all emails it receives from its users, no further authentication steps are necessary (‘`auth-sumbit`’).

According to the current version of the WKD draft, posteo should have published the public key for ‘keys@posteo.de’ in their own WKD. However, querying the directory with ‘`gpg`’ shows that this is not the case:

```
$ gpg --locate-external-keys keys@posteo.de
gpg: error retrieving 'keys@posteo.de' via WKD: No data
gpg: error reading key: No data
```

This is not really a security concern in this case, since the final recipient of the message is the email provider anyhow and the SMTP connection is already wrapped in TLS. However, it does point to the fact that even if the WKD standard specifies one thing, what ultimately counts is what the email provider’s implementation will accept. After some trial and error, I found out that ‘keys@posteo.de’ expects to receive ‘multipart/mixed’ messages, so eventually I submitted my key like this:

```
$ msmtp -t <<EOF
From: frankseifferth@posteo.net
To: keys@posteo.de
Subject: Key publishing request
MIME-Version: 1.0
Content-Type: multipart/mixed;
 boundary="=-=01-e8k41e11ob31eefa36wo=-="
```

```
--=-=01-e8k41e11ob31eefa36wo=-=
Content-Type: application/pgp-keys

-----BEGIN PGP PUBLIC KEY BLOCK-----

[[ base64-encoded public key, which may
continue over multiple lines ]]
-----END PGP PUBLIC KEY BLOCK-----

--=-=01-e8k41e11ob31eefa36wo=-=--
EOF
```

To get the base64-encoded public key we need to include in this message, we can either copy it from the Autocrypt header of any email that was sent with Delta Chat, or we can export it from our gpg keyring by running '`gpg --export --armor`'. Finally, to confirm that the key has indeed been added to posteo's WKD, we can simply run

```
$ gpg --locate-external-keys frankseifferth@posteo.net
gpg: key CAC590E4BF6815AE: "<frankseifferth@posteo.[...]
gpg: Total number processed: 1
gpg:              unchanged: 1
pub   ed25519 2024-11-25 [SC]
      B280419AA8EE19B8645C8FDECAC590E4BF6815AE
uid           [ultimate] <frankseifferth@posteo.net>
sub   cv25519 2024-11-25 [E]
```

Or, if we want to take a more low-level approach, we can retrieve the key with '`curl`':

```
$ gpg-wks-client \
      --print-wkd-hash frankseifferth@posteo.net |
      cut -d' ' -f1
8xgkkgizbsbnodijjqgu8aohoxdnccqc
$ curl -s $wkd/hu/8xgkkgizbsbnodijjqgu8aohoxdnccqc |
      base64
[[ base64-encoded public key, which may
continue over multiple lines ]]
```

A note on the key format

If you take a close look at the OpenPGP key found in the Autocrypt header and at the one exported from the gpg keyring, you may notice some minor differences. In my case, the base64-encoded key found in the Autocrypt header generated by Delta Chat (when reflowed to 55 columns) looks like this:

```
xjMEZOSeKxYJKwYBBAHaRw8BAQdAlazYBqp64p/JrNvzDii/pTrJh9b
aPQ5dNDGJeIKt1UPNGzxmcmFua3NlaWZmZXJ0aEBwb3NOZW8ubmVOPs
KLBBAWCAAzAhkBBQJnRJ4rAhsDBAsJCAcGFQgJCgsCAxYCARYhBLKAQ
Zqo7hm4ZFyP3srFkOS/aBWuAAoJEMrFkOS/aBWu6XsBAJdX3IUozE9T
txU9Jl7+1cLvlF1stp4G9FYsLOzZEIw3AP4pWOWJB8bpw2OTIub5R9Q
rvfOtrDk7Rz24QA9wf9VWBM44BGdEnisSCisGAQQBl1UBBQEBBODe/l
3qsbPGWQiQjuZ65HED9S5lN8OVhvgjJxLsTLIWJgMBCAfCeAQYFggAI
AUCZOSeKwIbDBYhBLKAQZqo7hm4ZFyP3srFkOS/aBWuAAoJEMrFkOS/
aBWuOXkA/10VNpBkalIwleT44utmpU+iBxjqrlGyVhfqjfDxNC/xAP9
VvnbAR6x946fOkPLMco3hs7PSRNu5JbaMBIWaOxRbBw==
```

While the key generated by '`gpg --export --armor`' (again reflowed to 55 columns) looks like this:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mDMEZOSeKxYJKwYBBAHaRw8BAQdAlazYBqp64p/JrNvzDii/pTrJh9b
aPQ5dNDGJeIKt1UOOGzxmcmFua3NlaWZmZXJ0aEBwb3NOZW8ubmVOPo
iLBBAWCAAzAhkBBQJnRJ4rAhsDBAsJCAcGFQgJCgsCAxYCARYhBLKAQ
Zqo7hm4ZFyP3srFkOS/aBWuAAoJEMrFkOS/aBWu6XsBAJdX3IUozE9T
txU9Jl7+1cLvlF1stp4G9FYsLOzZEIw3AP4pWOWJB8bpw2OTIub5R9Q
rvfOtrDk7Rz24QA9wf9VWBLg4BGdEnisSCisGAQQBl1UBBQEBBODe/l
3qsbPGWQiQjuZ65HED9S5lN8OVhvgjJxLsTLIWJgMBCAeIeAQYFggAI
AUCZOSeKwIbDBYhBLKAQZqo7hm4ZFyP3srFkOS/aBWuAAoJEMrFkOS/
aBWuOXkA/10VNpBkalIwleT44utmpU+iBxjqrlGyVhfqjfDxNC/xAP9
VvnbAR6x946fOkPLMco3hs7PSRNu5JbaMBIWaOxRbBw==
=L5ZF
-----END PGP PUBLIC KEY BLOCK-----
```

One difference between these two versions is that the second one contains a '`PGP PUBLIC KEY BLOCK`' header and footer; as well as a base64-encoded CRC checksum ('`L5ZF`'). Interestingly, this checksum is both strongly discouraged and entirely irrelevant according to the latest version of the OpenPGP standard

(Wouters et al. 2024, 76–77; Coles et al. 2024, sec. 13.4.1), yet it still seems to be very common in OpenPGP keys found in the wild. Additionally, there are a number of minor differences in the base64-encoded OpenPGP data itself. This is most noticeable at the beginning of the string, which starts with '`xj`' in the version generated by Delta Chat, while it starts with '`mD`' in the version produced by '`gpg`'. If we decode these two characters like this:

```
$ base64 -d <<<xj== | od -tx1
0000000 c6
0000001
$ base64 -d <<<mD== | od -tx1
0000000 98
0000001
```

We see that the version produced by Delta Chat starts with hexadecimal '`c6`', which is the same as binary '`11000110`'; while the version created by '`gpg`' starts with hexadecimal '`98`', which in binary is '`10011000`'. This first byte is an OpenPGP Packet Header, which can come in one of two formats (Wouters et al. 2024, 22):

```
                          +---------------+
Encoded Packet Type ID: |7 6 5 4 3 2 1 0|
                          +---------------+
OpenPGP format:
  Bit 7 -- always one
  Bit 6 -- always one
  Bits 5 to 0 -- Packet Type ID
Legacy format:
  Bit 7 -- always one
  Bit 6 -- always zero
  Bits 5 to 2 -- Packet Type ID
  Bits 1 to 0 -- length-type
```

Essentially, Delta Chat uses the new format for this Packet Header, which starts with the bits '`11`' and then uses the next six bits for the Packet Type ID; while '`gpg`' (more specifically gpg version 2.2.40) uses the legacy format, which starts with the bits '`10`' and then uses only four bits for the Packet Type ID. The Packet Type ID itself is the same in both cases — even if the alignment of this ID within the byte is slightly different. Interestingly, the

legacy format produced by '`gpg`' was already considered 'old' in the penultimate version of the OpenPGP RFC (Callas et al. 2007, 14); and the latest version of the standard clarifies that it has been obsoleted in 1998 and that it 'SHOULD NOT be used to generate new data' (Wouters et al. 2024, 22).

Except for this difference in the format of Packet Headers, however, the two versions are identical, as can be seen by running '`sq packet dump`' like this:

```
$ diff <(base64 -d key.v1 | sq packet dump) \
       <(base64 -d key.v2 | sq packet dump)
1c1
< Public-Key Packet, new CTB, 51 bytes
---
> Public-Key Packet, old CTB, 51 bytes
9c9
< User ID Packet, new CTB, 27 bytes
---
> User ID Packet, old CTB, 27 bytes
12c12
< Signature Packet, new CTB, 139 bytes
---
> Signature Packet, old CTB, 139 bytes
30c30
< Public-Subkey Packet, new CTB, 56 bytes
---
> Public-Subkey Packet, old CTB, 56 bytes
38c38
< Signature Packet, new CTB, 120 bytes
---
> Signature Packet, old CTB, 120 bytes
```

What this output also tells us is that the OpenPGP key found in my emails' Autocrypt header consists of a primary Public-Key, a User ID, and a single Public-Subkey. If we follow the advice in Coles et al. (2024, chap. 26) and look at the output of '`sq packet dump`' — or even '`sq packet dump -x`' — in more detail, we will discover that the key has the following structure:

```
$ base64 -d key.v1 | sq packet dump
Public-Key Packet, new CTB, 51 bytes
```

```
    [...]
User ID Packet, new CTB, 27 bytes
    Value: <frankseifferth@posteo.net>

Signature Packet, new CTB, 139 bytes
    [...]
      Key flags: CS
    [...]
Public-Subkey Packet, new CTB, 56 bytes
    [...]
Signature Packet, new CTB, 120 bytes
    [...]
      Key flags: EtEr
    [...]
```

That is, the User ID Packet specifies a single email address and no display name. (Remember that including a display name in the User ID Packet was also disallowed in posteo's WKD policy via the '`mailbox-only`' directive.) The primary Public-Key Packet contains an ed25519 key that, according to the Key Flags found in the first signature, may be used for certifying other keys ('`C`') and for signing messages ('`S`'). And finally, the Public-Subkey Packet contains an cv25519 key that, according to the Key Flags found in the second signature, may be used both for encryption in transport ('`Et`') and for encryption at rest ('`Er`').

Code and future work

In order to simplify the use of Autocrypt keys on my laptop, I bundled the code provided above into a script that is available at https://github.com/seifferth/notmuch-autocrypt/. This script makes it easy to extract OpenPGP keys from the notmuch database by invoking it like this:

```
$ notmuch autocrypt --locate-keys \
    patrice.lumumba@example.net \
    frankseifferth@posteo.net
gpg: key 139563682A020D0A: "<patrice.lumumba@exampl[...]
gpg: Total number processed: 1
gpg:              unchanged: 1
```

```
gpg: key CAC590E4BF6815AE: "<frankseifferth@posteo.[...]
gpg: Total number processed: 1
gpg:              unchanged: 1
```

The script also provides an '`--import-secret-key`' command that can be used to import the private key from an Autocrypt Setup Message. While these two functions already cover my own use case, however, this is hardly a feature-complete implementation of Autocrypt for notmuch. Most importantly, the script is still lacking support for generating, rather than only importing, Autocrypt keys and Autocrypt Setup Messages. There is also no support for the optional 'Key Gossip' extension yet and there is no support for adding Autocrypt headers to outgoing email. Since sending emails is out of scope for what notmuch is supposed to do, this last feature would probably need to be implemented in notmuch frontends.* The other features mentioned above, though, would clearly be useful additions to the script. So if anyone should be interested in seeing a more feature-complete version of '`notmuch autocrypt`' — and maybe even willing to help test these new features — I would be happy to receive an email regarding this matter. Possibly even an end-to-end encrypted one.

References

Amarilli, Antoine. 2018. 'Indexing Encrypted Email with Notmuch'. Blog Post. https://a3nm.net/blog/notmuch-encrypted.html.

Autocrypt Team. 2020. *Autocrypt Level 1 Specification: Release 1.1.0*. https://docs.autocrypt.org/autocrypt-spec-1.1.0.pdf.

Callas, Jon, Lutz Donnerhacke, Hal Finney, David Shaw and Rodney Thayer. 2007. 'OpenPGP Message Format'. RFC 4880. https://www.rfc-editor.org/rfc/pdfrfc/rfc4880.txt.pdf.

Coles, Tammi, Sabrina Kurtz, Wiktor Kwapisiewicz, David Runge, Paul Schaub and Heiko Schäfer. 2024. *OpenPGP for Application Developers*. https://openpgp.dev/book/.

*I myself am using an excellent notmuch frontend called bower (https://github.com/wangp/bower/) that is also lacking support for adding Autocrypt headers. However, the fact that my email provider will helpfully add Autocrypt headers to all emails that lack them kind of solves this issue, at least for my own use case.

Delta Chat Team. 2025. 'Delta Chat V2: A Major Security Upgrade, Beautified Contact Profiles, New Email Action and Direct App Access in Chats'. Blog Post. https://delta.chat/en/2025-08-04-encryption-v2.

Koch, Werner. 2016–2025. 'OpenPGP Web Key Directory'. Internet-Draft. https://datatracker.ietf.org/doc/draft-koch-openpgp-webkey-service/.

Marlinspike, Moxie. 2016. 'Reflections: The Ecosystem is Moving'. Blog Post. https://signal.org/blog/the-ecosystem-is-moving/.

Poddebniak, Damian, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky and Jörg Schwenk. 2018. 'Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels'. In *Proceedings of the 27th USENIX Security Symposium,* 549–566. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-poddebniak.pdf.

Wouters, Paul, Daniel Huigens, Justus Winter and Yutaka Niibe. 2024. 'OpenPGP'. RFC 9580. https://www.rfc-editor.org/rfc/rfc9580.pdf.