Post-quantum OpenPGP with KMail on Fedora 45 KDE

Stavros Kousidis

Published 2026-09-27
Tested on Fedora 45 KDE Plasma · KMail 26.08.1 · GPGME 2.0.1 with experimental RFC 9980 patch · Sequoia Chameleon GnuPG 0.13.1

This guide covers the KMail-specific part of the Fedora 45 RFC 9980 proof of concept. Complete the shared setup first, including the patched GPGME installation, Chameleon command selection, and disposable test certificate import and authorization.

KMail uses its OpenPGP integration through GPGME:

```
KMail
  -> QGpgME
  -> GPGME++
  -> patched GPGME
  -> Sequoia Chameleon
  -> Sequoia OpenPGP / Keystore
```

The steps below configure KMail, create PGP/MIME messages, and independently inspect their output.

Install KMail

Install KMail using Fedora's normal package:

```
sudo dnf install -y kmail
```

Do not explicitly install `qgpgme-qt6` or `gpgmepp`. They are normal KMail dependencies and are installed automatically by Fedora.

Verify the relevant packages:

```
rpm -q \
  kmail \
  kmail-libs \
  gpgme \
  gpgmepp \
  qgpgme-qt6 \
  libkleo \
  akonadi-server
```

Verify that your package versions match or supersede the tested baseline:

```
kmail-26.08.1-1.fc45.x86_64
kmail-libs-26.08.1-1.fc45.x86_64
gpgme-2.0.1-6.rfc9980.1.fc45.x86_64
gpgmepp-2.0.0-2.fc45.x86_64
qgpgme-qt6-2.0.0-4.fc45.x86_64
libkleo-26.08.1-1.fc45.x86_64
akonadi-server-26.08.1-2.fc45.x86_64
```

Start KMail and verify the runtime stack

Use the same terminal in which you set the compatibility `PATH` in the shared setup. Close any existing KMail instance before continuing; do not start KMail from the desktop launcher for this test.

Confirm the selected OpenPGP command and GPGME package:

```
command -v gpg
gpgconf --list-components | grep -E '^(gpg:|gpgsm:)'
rpm -q gpgme
```

The OpenPGP component should point to `/usr/libexec/rfc9980-openpgp/bin/gpg`, while `gpgsm` remains `/usr/bin/gpgsm`. The GPGME package should be the experimental RFC 9980 build from the shared setup.

Start KMail once from this terminal and keep this instance open for the remaining steps:

```
kmail >/tmp/rfc9980-kmail.log 2>&1 &
KMAIL_PID=$!
sleep 8
kill -0 "$KMAIL_PID"
```

If the last command fails, inspect `/tmp/rfc9980-kmail.log` and check whether an earlier KMail instance was still running. Do not assume that `$KMAIL_PID` identifies the active application after asynchronous startup.

Inspect the crypto libraries loaded by the process:

```
grep -E \
  'libgpgme\.so|libgpgmepp\.so|libqgpgme' \
  "/proc/$KMAIL_PID/maps" \
  | awk '{print $6}' \
  | sort -u
```

The tested runtime loaded:

```
/usr/lib64/libgpgme.so.45.0.1
/usr/lib64/libgpgmepp.so.7.0.0
/usr/lib64/libqgpgmeqt6.so.15v2.7.0
```

The library paths establish the GPGME integration, while the package and `gpgconf` checks establish the patched GPGME build and Chameleon command selection.

Configure a KMail test identity

On the first KMail start, skip the account setup assistant. This test does not require a receiving account. Configure the test identity and outgoing transport manually under **Settings → Configure KMail**.

Under **Identities**, create a dedicated test identity:

```
Name:  RFC9980 PoC
Email: rfc9980-poc@example.invalid
```

Modify the identity and open its **Cryptography** tab. KMail should automatically match the identity to the certified post-quantum OpenPGP certificate generated during the shared setup. Verify that the fingerprint and algorithm are correct.

Use KMail's normal `Local Folders` resource for Drafts, Sent and Outbox. Do not create a receiving account, separate Maildir resource, or isolated Akonadi instance for this test.

Under **Sending**, create a deliberately non-functional local SMTP transport so KMail can queue the test messages. Configure:

```
Name: RFC9980 Test
Server: localhost
Port: 25
Encryption: None
Authentication: LOGIN
```

KMail 26.08.1 does not offer a `None` authentication method in this dialog. No credentials are needed for this test because the transport is deliberately non-functional. Set **Send message** to proceed anyway.

The purpose of this transport is only to let KMail construct the message and place it in Outbox. A working SMTP server is not required.

Test unsigned mail

Compose a message to:

```
rfc9980-poc@example.invalid
```

Disable both signing and encryption, then queue it.

The resulting Outbox message should have an ordinary content type such as:

```
Content-Type: text/plain; charset="utf-8"
```

This verifies the KMail, Akonadi and Outbox configuration independently of OpenPGP.

Test signing

Compose another message to the same recipient. Enable OpenPGP signing and disable encryption, then queue it.

The Outbox message should use PGP/MIME:

```
Content-Type: multipart/signed;
 protocol="application/pgp-signature"
```

Test signing and encryption

Compose a third message to the same recipient. Enable both OpenPGP signing and encryption, then queue it.

The outer message should use PGP/MIME encryption:

```
Content-Type: multipart/encrypted;
 protocol="application/pgp-encrypted"
```

Locate the Outbox messages

The tested installation stored Local Folders in an Akonadi Maildir resource, but its instance number is not portable. First discover candidate Outbox directories:

```
find "$HOME/.local/share" -type d -name outbox \
  -path '*/akonadi_maildir_resource_*/*' -print
```

Compare the candidates with the **Local Folders → Outbox** shown in KMail. Identify the matching resource:

```
OUTBOX="$HOME/.local/share/akonadi_maildir_resource_0/outbox"
```

The `resource_0` value above is **only an example**. Confirm the selected directory contains Maildir subdirectories:

```
test -d "$OUTBOX/cur" && test -d "$OUTBOX/new"
find "$OUTBOX" -type f \( -path '*/new/*' -o -path '*/cur/*' \) \
  -printf '%T@ %p\n' | sort -nr | head
```

Match the test messages by their decoded subjects and MIME types:

```
find "$OUTBOX" -type f \( -path '*/new/*' -o -path '*/cur/*' \) -print0 |
while IFS= read -r -d '' MESSAGE; do
  printf '\n=== %s ===\n' "$MESSAGE"
  python3 - "$MESSAGE" <<'PY'
from email import policy
from email.parser import BytesParser
from pathlib import Path
import sys

path = Path(sys.argv[1])
message = BytesParser(policy=policy.default).parsebytes(path.read_bytes())

print("Subject:", message.get("Subject", ""))
print("Content-Type:", message.get_content_type())
PY
done
```

For the three messages created above, identify the entries with subjects `Unsigned`, `Signed only`, and `Encrypted + signed`. Their MIME types should be `text/plain`, `multipart/signed`, and `multipart/encrypted` respectively.

Independent message verification

Use the actual KMail Outbox message paths found above. Set both paths and one verification directory:

```
SIGNED="SIGNED_MESSAGE"
ENCRYPTED="ENCRYPTED_MESSAGE"
WORK="$HOME/rfc9980-mail-test/kmail-verify"
export SIGNED ENCRYPTED WORK

mkdir -p "$WORK"
```

Replace both placeholders with full paths to the corresponding raw MIME files.

Verify the signed message

Extract the detached PGP/MIME signature and the signed MIME part. PGP/MIME signatures are computed over MIME content using canonical CRLF line endings:

```
python3 - <<'PY'
from email import policy
from email.parser import BytesParser
from pathlib import Path
import os

signed = Path(os.environ["SIGNED"])
work = Path(os.environ["WORK"])

m = BytesParser(policy=policy.SMTP).parsebytes(signed.read_bytes())
parts = list(m.iter_parts())
assert len(parts) >= 2, "Signed message does not have expected MIME parts"

work.joinpath("signed-content.txt").write_bytes(
    parts[0].as_bytes(policy=policy.SMTP)
)
work.joinpath("signed-signature.asc").write_bytes(
    parts[1].get_payload(decode=True)
)
PY
```

Inspect the signature packet:

```
gpg --list-packets "$WORK/signed-signature.asc"
```

Confirm that the signature contains:

```
:signature packet: algo 30
```

Verify it:

```
gpg --status-fd=1 \
  --verify \
  "$WORK/signed-signature.asc" \
  "$WORK/signed-content.txt"
```

A successful verification reports:

```
gpg: using MLDSA65_Ed25519 key ...
[GNUPG:] GOODSIG ...
[GNUPG:] VALIDSIG ... 30 ...
```

and returns status `0`.

Verify the encrypted message

Extract the encrypted OpenPGP payload:

```
python3 - <<'PY'
from email import policy
```

```
from email.parser import BytesParser
from pathlib import Path
import os

encrypted = Path(os.environ["ENCRYPTED"])
work = Path(os.environ["WORK"])

m = BytesParser(policy=policy.SMTP).parsebytes(encrypted.read_bytes())
parts = list(m.iter_parts())
assert len(parts) >= 2, "Encrypted message does not have expected MIME parts"

work.joinpath("encrypted.pgp").write_bytes(
    parts[1].get_payload(decode=True)
)
PY
```

Inspect the packet:

```
gpg --list-packets "$WORK/encrypted.pgp"
```

Confirm that the message contains:

```
:pubkey enc packet: version 6, algo 35
```

Decrypt it:

```
gpg --status-fd=1 \
  --output "$WORK/decrypted.eml" \
  --decrypt "$WORK/encrypted.pgp"
```

A successful decryption reports:

```
[GNUPG:] ENC_TO ... 35 0
gpg: encrypted with MLKEM768_X25519 key ...
[GNUPG:] DECRYPTION_OKAY
```

The decrypted MIME entity is itself signed:

```
grep -m1 '^Content-Type:' "$WORK/decrypted.eml"
```

The expected result is:

```
Content-Type: multipart/signed;
```

Extract the inner signature:

```
python3 - <<'PY'
from email import policy
from email.parser import BytesParser
from pathlib import Path
import os

work = Path(os.environ["WORK"])
m = BytesParser(policy=policy.SMTP).parsebytes(
    work.joinpath("decrypted.eml").read_bytes()
)
parts = list(m.iter_parts())
assert len(parts) >= 2, "Decrypted message is not multipart/signed"

work.joinpath("inner-content.txt").write_bytes(
    parts[0].as_bytes(policy=policy.SMTP)
)
work.joinpath("inner-signature.asc").write_bytes(
    parts[1].get_payload(decode=True)
)
PY
```

Inspect the inner signature:

```
gpg --list-packets "$WORK/inner-signature.asc"
```

Confirm that the result contains:

```
:signature packet: algo 30
```

Verify it:

```
gpg --status-fd=1 \
  --verify \
  "$WORK/inner-signature.asc" \
  "$WORK/inner-content.txt"
```

A successful verification reports:

```
gpg: using MLDSA65_Ed25519 key ...
[GNUPG:] GOODSIG ...
[GNUPG:] VALIDSIG ... 30 ...
```

and returns status `0`.

Result

The procedure exercises this complete path:

```
KMail
  -> QGpgME
  -> GPGME++
  -> patched GPGME
  -> Sequoia Chameleon
  -> Sequoia Keystore
```

For signing:

```
OpenPGP algorithm: 30
Construction: ML-DSA-65+Ed25519
Signature packet version: 6
KMail PGP/MIME creation: successful
Independent verification: successful
```

For encryption:

```
OpenPGP algorithm: 35
Construction: ML-KEM-768+X25519
Public-key encrypted session key packet version: 6
KMail PGP/MIME creation: successful
Independent decryption: successful
Inner signature verification: successful
```

This establishes that KMail can use the Sequoia-based RFC 9980 implementation through GPGME and Chameleon to create and process post-quantum OpenPGP mail.

Scope and next guides

This procedure demonstrates KMail integration on Fedora 45 KDE using the experimental GPGME build and Sequoia Chameleon. See the shared setup for the common Chameleon environment and the Claws Mail and Evolution guides for other client tests.