Workgroup: openpgp
Internet-Draft:
draft-dkg-openpgp-stateless-cli-06
Published: 10 March 2023
Intended Status: Informational
Expires: 11 September 2023
Authors: D. K. Gillmor
ACLU

Stateless OpenPGP Command Line Interface

Abstract

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, known as sop. It aims for a minimal, well-structured API covering OpenPGP object security.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://dkg.gitlab.io/openpgp-stateless-cli/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/.

Discussion of this document takes place on the OpenPGP Working Group mailing list (mailto:openpgp@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/openpgp/. Subscribe at https://www.ietf.org/mailman/listinfo/openpgp/.

Source for this draft and an issue tracker can be found at https://gitlab.com/dkg/openpgp-stateless-cli/.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Different OpenPGP implementations have many different requirements, which typically break down in two main categories: key/certificate management and object security.

The purpose of this document is to provide a "stateless" interface that primarily handles the object security side of things, and assumes that secret key management and certificate management will be handled some other way.

Isolating object security from key/certificate management should make it easier to provide interoperability testing for the object security side of OpenPGP implementations, as described in Section 1.3.

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, known here by the placeholder sop. It aims for a minimal, well-structured API.

An OpenPGP implementation should not name its executable sop to implement this specification. It just needs to provide a program that conforms to this interface.

A sop implementation should leave no trace on the system, and its behavior should not be affected by anything other than command-line arguments and input.

Obviously, the user will need to manage their secret keys (and their peers' certificates) somehow, but the goal of this interface is to separate out that task from the task of interacting with OpenPGP messages.

While this document identifies a command-line interface, the rough outlines of this interface should also be amenable to relatively straightforward library implementations in different languages.

1.1. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terminology

This document uses the term "key" to refer exclusively to OpenPGP Transferable Secret Keys (see Section 11.2 of [RFC4880]).

It uses the term "certificate" to refer to OpenPGP Transferable Public Key (see Section 11.1 of [RFC4880]).

"Stateless" in "Stateless OpenPGP" means avoiding secret key and certificate state. The user is responsible for managing all OpenPGP certificates and secret keys themselves, and passing them to sop as needed. The user should also not be concerned that any state could affect the underlying operations.

OpenPGP revocations can have "Reason for Revocation" (Section 5.2.3.23 of [RFC4880]), which can be either "soft" or "hard". The set of "soft" reasons is: "Key is superseded" and "Key is retired and no longer used". All other reasons (and revocations that do not state a reason) are "hard" revocations.

1.3. Using sop in a Test Suite

If an OpenPGP implementation provides a sop interface, it can be used to test interoperability (e.g., [OpenPGP-Interoperability-Test-Suite]).

Such an interop test suite can, for example, use custom code (*not* sop) to generate a new OpenPGP object that incorporates new primitives, and feed that object to a stable of sop implementations, to determine whether those implementations can consume the new form.

Or, the test suite can drive each sop implementation with a simple input, and observe which cryptographic primitives each implementation chooses to use as it produces output.

1.4. Semantics vs. Wire Format

The semantics of sop are deliberately simple and very high-level compared to the vast complexity and nuance available within the OpenPGP specification. This reflects the perspective of nearly every piece of tooling that relies on OpenPGP to accomplish its task: most toolchains don't care about the specifics, they just want the high-level object security properties.

Given this framing, this document generally tries to avoid overconstraining the details of the wire format objects emitted, or what kinds of wire format structures should be acceptable or unacceptable. This allows a test suite to evaluate and contrast the wire format choices made by different implementations in as close to their native configuration as possible. It also makes it easier to promote interoperability by ensuring that the native wire formats emitted by one implementation can be consumed by another, without relying on their choices of wire format being constrained by this draft.

Where this draft does identify specific wire format requirements, that might be due to an ambiguity in the existing specifications (which maybe needs fixing elsewhere), or to a bug in this specification that could be improved.

2. Examples

These examples show no error checking, but give a flavor of how sop might be used in practice from a shell.

The key and certificate files described in them (e.g. alice.sec) could be for example those found in [I-D.draft-bre-openpgp-samples-01].

```
sop generate-key "Alice Lovelace <alice@openpgp.example>" > alice.sec
sop extract-cert < alice.sec > alice.pgp

sop generate-key "Bob Babbage <bob@openpgp.example>" > bob.sec
sop extract-cert < bob.sec > bob.pgp

sop sign --as=text alice.sec < statement.txt > statement.txt.asc
sop verify statement.txt.asc alice.pgp < statement.txt

sop encrypt --sign-with=alice.sec bob.pgp < msg.eml > ciphertext.asc
sop decrypt bob.sec < ciphertext.asc > cleartext.eml
```

See Section 6 for more information about errors and error handling.

3. Subcommands

sop uses a subcommand interface, similar to those popularized by systems like git and svn.

If the user supplies a subcommand that sop does not implement, it fails with UNSUPPORTED_SUBCOMMAND. If a sop implementation does not handle a supplied option for a given subcommand, it fails with UNSUPPORTED_OPTION.

All subcommands that produce OpenPGP material on standard output produce ASCII-armored (Section 6 of [I-D.ietf-openpgp-crypto-refresh-07]) objects by default (except for sop dearmor). These subcommands have a --no-armor option, which causes them to produce binary OpenPGP material instead.

All subcommands that accept OpenPGP material on input should be able to accept either ASCII-armored or binary inputs (see Section 9.4) and behave accordingly.

See Section 5 for details about how various forms of OpenPGP material are expected to be structured.

3.1. version: Version Information

sop version [--backend|--extended|--sop-spec]

- *Standard Input: ignored

- *Standard Output: version information

This subcommand emits version information as UTF-8-encoded text.

With no arguments, the version string emitted should contain the name of the sop implementation, followed by a single space, followed by the version number. A sop implementation should use a version number that respects an established standard that is easily comparable and parsable, like [SEMVER].

If --backend is supplied, the implementation should produce a comparable line of implementation and version information about the primary underlying OpenPGP toolkit.

If --extended is supplied, the implementation may emit multiple lines of version information. The first line **MUST** match the information produced by a simple invocation, but the rest of the text has no defined structure.

If --sop-spec is supplied, the implementation should emit a single line of text indicating the latest version of this draft that it targets, for example, draft-dkg-openpgp-stateless-cli-06. If the implementation targets a specific draft but the implementer knows the implementation is incomplete, it should prefix the draft title with a "~" (TILDE, U+007E), for example: ~draft-dkg-openpgp-stateless-cli-06. The implementation **MAY** emit additional text about its relationship to the targeted draft on the lines following the versioned title.

--backend, --extended, and --sop-spec are mutually-exclusive options.

Example:

```
$ sop version
ExampleSop 0.2.1
$ sop version --backend
LibExamplePGP 3.4.2
$ sop version --extended
ExampleSop 0.2.1
Running on MonkeyScript 4.5
LibExamplePGP 3.4.2
LibExampleCrypto 3.1.1
LibXCompression 4.0.2
See https://pgp.example/sop/ for more information
$ sop version --sop-spec
~draft-dkg-openpgp-stateless-cli-06

This implementation does not handle @FD: special designators for output.
$
```

3.2. list-profiles: Describe Available Profiles

```
sop list-profiles SUBCOMMAND
```

- *Standard Input: ignored

- *Standard Output: PROFILELIST (Section 5.12)

This subcommand emits a list of profiles supported by the identified subcommand.

If the indicated SUBCOMMAND does not accept a --profile option, it returns UNSUPPORTED_PROFILE.

Example:

```
$ sop list-profiles generate-key
default: use the implementer's recommendations
rfc4880: use algorithms from RFC 4880
$
```

3.3. generate-key: Generate a Secret Key

```
sop generate-key [--no-armor]
    [--with-key-password=PASSWORD]
    [--profile=PROFILE]
    [--] [USERID...]
```

- *Standard Input: ignored

- *Standard Output: KEYS (Section 5.3)

Generate a single default OpenPGP key with zero or more User IDs.

The generated secret key **SHOULD** be usable for as much of the sop functionality as possible. In particular:

* It should be possible to extract an OpenPGP certificate from the key in KEYS with sop extract-cert.

* The key in KEYS should be able to create signatures (with sop sign) that are verifiable by using sop verify with the extracted certificate.

* The key in KEYS should be able to decrypt messages (with sop decrypt) that are encrypted by using sop encrypt with the extracted certificate.

The detailed internal structure of the certificate is left to the discretion of the sop implementation.

If the --with-key-password option is supplied, the generated key will be password-protected (locked) with the supplied password. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 5). See also the guidance on ensuring that the password is human-readable in Section 9.8.1.

If the --profile argument is supplied and the indicated PROFILE is not supported by the implementation, sop will fail with UNSUPPORTED_PROFILE.

If no --with-key-password option is supplied, the generated key will be unencrypted.

Example:

```
$ sop generate-key 'Alice Lovelace <alice@openpgp.example>' > alice.sec
$ head -n1 < alice.sec
-----BEGIN PGP PRIVATE KEY BLOCK-----
$
```

3.4. extract-cert: Extract a Certificate from a Secret Key

```
sop extract-cert [--no-armor]
```

* Standard Input: KEYS (Section 5.3)

* Standard Output: CERTS (Section 5.2)

The output should contain one OpenPGP certificate in CERTS per OpenPGP Transferable Secret Key found in KEYS. There is no guarantee what order the CERTS will be in.

sop extract-cert **SHOULD** work even if any of the keys in KEYS is password-protected.

Example:

```
$ sop extract-cert < alice.sec > alice.pgp
$ head -n1 < alice.pgp
-----BEGIN PGP PUBLIC KEY BLOCK-----
$
```

3.5. sign: Create Detached Signatures

```
sop sign [--no-armor] [--micalg-out=MICALG]
     [--with-key-password=PASSWORD...]
     [--as={binary|text}] [--] KEYS [KEYS...]
```

*Standard Input: DATA (Section 5.11)

*Standard Output: SIGNATURES (Section 5.6)

Exactly one signature will be made by each key in the supplied KEYS arguments.

--as defaults to binary. If --as=text and the input DATA is not valid UTF-8 (Section 9.7), sop sign fails with EXPECTED_TEXT.

--as=binary **SHOULD** result in OpenPGP signatures of type 0x00 ("Signature of a binary document"). --as=text **SHOULD** result in OpenPGP signatures of type 0x01 ("Signature of a canonical text document"). See Section 5.2.1 of [RFC4880] for more details.

When generating PGP/MIME messages ([RFC3156]), it is useful to know what digest algorithm was used for the generated signature. When --micalg-out is supplied, sop sign emits the digest algorithm used to the specified MICALG file in a way that can be used to populate the micalg parameter for the Content-Type (see Section 5.8). If the specified MICALG file already exists in the filesystem, sop sign will fail with OUTPUT_EXISTS.

When signing with multiple keys, sop sign **SHOULD** use the same digest algorithm for every signature generated in a single run, unless there is some internal constraint on the KEYS objects. If --micalg-out is requested, and multiple incompatibly-constrained KEYS objects are supplied, sop sign **MUST** emit the empty string to the designated MICALG.

If the signing key material in any key in the KEYS objects is password-protected, sop sign **SHOULD** try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the PASSWORD options unlock the key (or if no such option is supplied), sop sign will fail with KEY_IS_PROTECTED. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 5). Note also the guidance for retrying variants of a non-human-readable password in Section 9.8.2.

If any key in the KEYS objects is not capable of producing a signature, sop sign will fail with KEY_CANNOT_SIGN.

sop sign **MUST NOT** produce any extra signatures beyond those from KEYS objects supplied on the command line.

Example:

```
$ sop sign --as=text alice.sec < message.txt > message.txt.asc
$ head -n1 < message.txt.asc
-----BEGIN PGP SIGNATURE-----
$
```

3.6. verify: Verify Detached Signatures

```
sop verify [--not-before=DATE] [--not-after=DATE]
    [--] SIGNATURES CERTS [CERTS...]
```

*Standard Input: DATA (Section 5.11)

*Standard Output: VERIFICATIONS (Section 5.10)

--not-before and --not-after indicate that signatures with dates outside certain range **MUST NOT** be considered valid.

--not-before defaults to the beginning of time. Accepts the special value - to indicate the beginning of time (i.e. no lower boundary).

--not-after defaults to the current system time (now). Accepts the special value - to indicate the end of time (i.e. no upper boundary).

sop verify only returns OK if at least one certificate included in any CERTS object made a valid signature in the time window specified over the DATA supplied.

For details about the valid signatures, the user **MUST** inspect the VERIFICATIONS output.

If no CERTS are supplied, sop verify fails with MISSING_ARG.

If no valid signatures are found, sop verify fails with NO_SIGNATURE.

See Section 11.1 for more details about signature verification.

Example:

(In this example, we see signature verification succeed first, and then fail on a modified version of the message.)

```
$ sop verify message.txt.asc alice.pgp < message.txt
2019-10-29T18:36:45Z EB85BB5FA33A75E15E944E63F231550C4F47E38E EB85BB5FA3
$ echo $?
0
$ tr a-z A-Z < message.txt | sop verify message.txt.asc alice.pgp
$ echo $?
3
$
```

3.7. encrypt: Encrypt a Message

```
sop encrypt [--as={binary|text}]
    [--no-armor]
    [--with-password=PASSWORD...]
    [--sign-with=KEYS...]
    [--with-key-password=PASSWORD...]
    [--profile=PROFILE]
    [--] [CERTS...]
```

- Standard Input: DATA (Section 5.11)

- Standard Output: CIPHERTEXT (Section 5.4)

--as defaults to binary. The setting of --as corresponds to the one octet format field found in the Literal Data packet at the core of the output CIPHERTEXT. If --as is set to binary, the octet is b (0x62). If it is text, the format octet is u (0x75).

--with-password enables symmetric encryption (and can be used multiple times if multiple passwords are desired).

--sign-with creates exactly one signature by for each secret key found in the supplied KEYS object (this can also be used multiple times if signatures from keys found in separaate files are desired). If any key in any supplied KEYS object is not capable of producing a signature, sop sign will fail with KEY_CANNOT_SIGN. If any signing key material in any supplied KEYS object is password-protected, sop encrypt **SHOULD** try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the --with-key-password=PASSWORD options can unlock any locked signing key material (or if no such option is supplied), sop encrypt will fail with KEY_IS_PROTECTED.

All signatures made must be placed inside the encryption produced by sop encrypt.

Note that both --with-password and --with-key-password supply PASSWORD arguments, but they do so in different contexts which are not interchangeable. A PASSWORD supplied for symmetric encryption (--with-password) **MUST NOT** be used to try to unlock a signing key (--with-key-password) and a PASSWORD supplied to unlock a signing key **MUST NOT** be used to symmetrically encrypt the message. Regardless of context, each PASSWORD argument is presented as an indirect data type from which the actual password is acquired (Section 5). If sop encrypt encounters a password which is not a valid UTF-8 string (Section 9.7), or is otherwise not robust in its representation to humans, it fails with PASSWORD_NOT_HUMAN_READABLE. If sop encrypt sees trailing whitespace at the end of a password, it will trim the trailing whitespace before using the password. See Section 9.8 for more discussion about passwords.

If --as is set to binary, then --sign-with will sign as a binary document (OpenPGP signature type 0x00).

If --as is set to text, then --sign-with will sign as a canonical text document (OpenPGP signature type 0x01). In this case, if the input DATA is not valid UTF-8 (Section 9.7), sop encrypt fails with EXPECTED_TEXT.

If --sign-with is supplied for input DATA that is not valid UTF-8, sop encrypt **MAY** sign as a binary document (OpenPGP signature type 0x00).

sop encrypt **MUST NOT** produce any extra signatures beyond those from KEYS objects identified by --sign-with.

The resulting CIPHERTEXT should be decryptable by the secret keys corresponding to every certificate included in all CERTS, as well as each password given with --with-password.

If no CERTS or --with-password options are present, sop encrypt fails with MISSING_ARG.

If at least one of the identified certificates requires encryption to an unsupported asymmetric algorithm, sop encrypt fails with UNSUPPORTED_ASYMMETRIC_ALGO.

If at least one of the identified certificates is not encryption-capable (e.g., revoked, expired, no encryption-capable flags on primary key and valid subkeys), sop encrypt fails with CERT_CANNOT_ENCRYPT.

If the --profile argument is supplied and the indicated PROFILE is not supported by the implementation, sop will fail with UNSUPPORTED_PROFILE. The use of a profile for this subcommand allows an implementation faced with parametric or algorithmic choices to make a decision coarsely guided by the operator. For example, when encrypting with a password, there is no knowledge about the capabilities of the recipient, and an implementation may prefer cryptographically modern algorithms, or it may prefer more broad compatibility. In the event that a known recipient (i.e., one of the CERTS) explicitly indicates a lack of support for one of the features preferred by the indicated profile, the implementation **SHOULD** conform to the recipient's advertised capabilities where possible.

If sop encrypt fails for any reason, it emits no CIPHERTEXT.

Example:

(In this example, bob.bin is a file containing Bob's binary-formatted OpenPGP certificate. Alice is encrypting a message to both herself and Bob.)

```
$ sop encrypt --as=text --sign-with=alice.key alice.asc bob.bin < messag
$ head -n1 encrypted.asc
-----BEGIN PGP MESSAGE-----
$
```

3.8. decrypt: Decrypt a Message

```
sop decrypt [--session-key-out=SESSIONKEY]
    [--with-session-key=SESSIONKEY...]
    [--with-password=PASSWORD...]
    [--with-key-password=PASSWORD...]
    [--verifications-out=VERIFICATIONS
     [--verify-with=CERTS...]
     [--verify-not-before=DATE]
     [--verify-not-after=DATE] ]
    [--] [KEYS...]
```

*Standard Input: CIPHERTEXT (Section 5.4)

*Standard Output: DATA (Section 5.11)

The caller can ask sop for the session key discovered during decryption by supplying the --session-key-out option. If the specified file already exists in the filesystem, sop decrypt will fail with OUTPUT_EXISTS. When decryption is successful, sop decrypt writes the discovered session key to the specified file.

--with-session-key enables decryption of the CIPHERTEXT using the session key directly against the SEIPD packet. This option can be used multiple times if several possible session keys should be tried. SESSIONKEY is an indirect data type from which the actual sessionkey value is acquired (Section 5).

--with-password enables decryption based on any SKESK (Section 5.3 of [I-D.ietf-openpgp-crypto-refresh-07]) packets in the CIPHERTEXT. This option can be used multiple times if the user wants to try more than one password.

--with-key-password lets the user use password-protected (locked) secret key material. If the decryption-capable secret key material in any key in the KEYS objects is password-protected, sop decrypt **SHOULD** try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for decryption. If none of the --with-key-password options unlock the key (or if no such option is supplied), and the message cannot be decrypted with any other KEYS, --with-session-key, or --with-password options, sop decrypt will fail with KEY_IS_PROTECTED.

Note that the two kinds of PASSWORD options are for different domains: --with-password is for unlocking an SKESK, and --with-key-password is for unlocking secret key material in KEYS. sop decrypt **SHOULD NOT** apply the --with-key-password argument to any SKESK, or the --with-password argument to any KEYS.

Each PASSWORD argument is an indirect data type from which the actual password is acquired (Section 5). If sop decrypt tries and fails to use a password supplied by a PASSWORD, and it observes that there is trailing UTF-8 whitespace at the end of the password, it will retry with the trailing whitespace stripped. See Section 9.8.2 for more discussion about consuming password-protected key material.

--verifications-out produces signature verification status to the designated file. If the designated file already exists in the filesystem, sop decrypt will fail with OUTPUT_EXISTS.

The return code of sop decrypt is not affected by the results of signature verification. The caller **MUST** check the returned VERIFICATIONS to confirm signature status. An empty VERIFICATIONS output indicates that no valid signatures were found.

--verify-with identifies a set of certificates whose signatures would be acceptable for signatures over this message.

If the caller is interested in signature verification, both --verifications-out and at least one --verify-with must be supplied. If only one of these options is supplied, sop decrypt fails with INCOMPLETE_VERIFICATION.

--verify-not-before and --verify-not-after provide a date range for acceptable signatures, by analogy with the options for sop verify (see Section 3.6). They should only be supplied when doing signature verification.

See Section 11.1 for more details about signature verification.

If no KEYS or --with-password or --with-session-key options are present, sop decrypt fails with MISSING_ARG.

If unable to decrypt, sop decrypt fails with CANNOT_DECRYPT.

sop decrypt only emits cleartext to Standard Output that was successfully decrypted.

Example:

(In this example, Alice stashes and re-uses the session key of an encrypted message.)

```
$ sop decrypt --session-key-out=session.key alice.sec < ciphertext.asc >
$ ls -l ciphertext.asc cleartext.out
-rw-r--r-- 1 user user   321 Oct 28 01:34 ciphertext.asc
-rw-r--r-- 1 user user   285 Oct 28 01:34 cleartext.out
$ sop decrypt --with-session-key=session.key < ciphertext.asc > cleartex
$ diff cleartext.out cleartext2.out
$
```

3.8.1. Historic Options for sop decrypt

The sop decrypt option --verifications-out used to be named --verify-out. An implementation **SHOULD** accept either form of this option, and **SHOULD** produce a deprecation warning to standard error if the old form is used.

3.9. armor: Convert Binary to ASCII

```
sop armor [--label={auto|sig|key|cert|message}]
```

* Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, CIPHERTEXT, or INLINESIGNED)

* Standard Output: the same material with ASCII-armoring added, if not already present

The user can choose to specify the label used in the header and tail of the armoring.

The default for --label is auto, in which case, sop inspects the input and chooses the label appropriately, based on the OpenPGP packets encountered. If the type of the first OpenPGP packet is:

* 0x05 (Secret-Key), the packet stream should be parsed as a KEYS input (with Armor Header BEGIN PGP PRIVATE KEY BLOCK).

* 0x06 (Public-Key), the packet stream should be parsed as a CERTS input (with Armor Header BEGIN PGP PUBLIC KEY BLOCK).

* 0x01 (Public-key Encrypted Session Key) or 0x03 (Symmetric-key Encrypted Session Key), the packet stream should be parsed as a CIPHERTEXT input (with Armor Header BEGIN PGP MESSAGE).

* 0x04 (One-Pass Signature), the packet stream should be parsed as an INLINESIGNED input (with Armor Header BEGIN PGP MESSAGE).

* 0x02 (Signature), the packet stream may be either a SIGNATURES input or an INLINESIGNED input. If the packet stream contains only Signature packets, it should be parsed as aSIGNATURES input (with Armor Header BEGIN PGP SIGNATURE). If it contains any packet other than a Signature packet, it should be parsed as an INLINESIGNED input (with Armor Header BEGIN PGP MESSAGE).

If the input packet stream does not match the expected sequence of packet types, sop armor fails with BAD_DATA.

Note that --label=message may be used for either INLINESIGNED or CIPHERTEXT inputs.

Since sop armor accepts ASCII-armored input as well as binary input, this operation is idempotent on well-structured data. A caller can use this subcommand blindly to ensure that any well-formed OpenPGP packet stream is 7-bit clean.

FIXME: what to do if the input is a CSF INLINESIGNED message? Three choices:

* Leave it untouched -- this violates the claim about blindly ensuring 7-bit clean, since UTF-8-encoded message text is not necessarily 7-bit clean.

* Convert to ASCII-armored INLINESIGNED -- this requires synthesis of OPS packet (from the CSF Hash header) and Literal Data packet (from the message body).

* Raise a specific error.

Example:

```
$ sop armor < bob.bin > bob.pgp
$ head -n1 bob.pgp
-----BEGIN PGP PUBLIC KEY BLOCK-----
$
```

3.10. dearmor: Convert ASCII to Binary

```
sop dearmor
```

* Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, CIPHERTEXT, or INLINESIGNED)

* Standard Output: the same material with any ASCII-armoring removed

If the input packet stream does not match any of the expected sequence of packet types, sop dearmor fails with BAD_DATA. See also Section 9.4.

Since sop dearmor accepts binary-formatted input as well as ASCII-armored input, this operation is idempotent on well-structured data. A caller can use this subcommand blindly ensure that any well-formed OpenPGP packet stream is in its standard binary representation.

FIXME: what to do if the input is a CSF INLINESIGNED? Three choices:

* Leave it untouched -- output data is not really in binary format.

* Convert to binary-format INLINESIGNED -- this requires synthesis of OPS packet (from CSF Hash header) and Literal Data packet (from the message body).

* Raise a specific error.

Example:

```
$ sop dearmor < message.txt.asc > message.txt.sig
$
```

3.11. inline-detach: Split Signatures from an Inline-Signed Message

```
sop inline-detach [--no-armor] --signatures-out=SIGNATURES
```

* Standard Input: INLINESIGNED

* Standard Output: DATA (the message without any signatures)

In some contexts, the user may expect an inline-signed message of some form or another (INLINESIGNED, see Section 5.5) rather than a message and its detached signature. sop inline-detach takes such an inline-signed message on standard input, and splits it into:

* the potentially signed material on standard output, and

* a detached signature block to the destination identified by --signatures-out

Note that no cryptographic verification of the signatures is done by this subcommand. Once the inline-signed message is separated, verification of the detached signature can be done with sop verify.

If no --signatures-out is supplied, sop inline-detach fails with MISSING_ARG.

Note that there may be more than one Signature packet in an inline-signed message. All signatures found in the inline-signed message will be emitted to the --signatures-out destination.

If the inline-signed message uses the Cleartext Signature Framework, it may be dash-escaped (see Section 7.1 of [RFC4880]). The output of sop detach-inband-signature-and-message will have any dash-escaping removed.

If the input is not an INLINESIGNED message, sop inline-detach fails with BAD_DATA. If the input contains more than one object that could be interpreted as an INLINESIGNED message, sop inline-detach also fails with BAD_DATA. A sop implementation **MAY** accept (and discard) leading and trailing data when the incoming INLINESIGNED message uses the Cleartext Signature Framework.

If the file designated by --signatures-out already exists in the filesystem, sop detach-inband-signature-and-message will fail with OUTPUT_EXISTS.

Note that --no-armor here governs the data written to the --signatures-out destination. Standard output is always the raw message, not an OpenPGP packet.

Example:

```
$ sop inline-detach --signatures-out=Release.pgp < InRelease >Release
$ sop verify Release.pgp archive-keyring.pgp < Release
$
```

3.12. inline-verify: Verify an Inline-Signed Message

```
sop inline-verify [--not-before=DATE] [--not-after=DATE]
    [--verifications-out=VERIFICATIONS]
    [--] CERTS [CERTS...]
```

*Standard Input: INLINESIGNED (Section 5.5)

*Standard Output: DATA (Section 5.11)

This command is similar to sop verify (Section 3.6) except that it takes an INLINESIGNED message (see Section 5.5) and produces the message body (without signatures) on standard output. It is also similar to sop inline-detach (Section 3.11) except that it actually performs signature verification.

--not-before and --not-after indicate that signatures with dates outside certain range **MUST NOT** be considered valid.

--not-before defaults to the beginning of time. Accepts the special value - to indicate the beginning of time (i.e. no lower boundary).

--not-after defaults to the current system time (now). Accepts the special value - to indicate the end of time (i.e. no upper boundary).

sop inline-verify only returns OK if INLINESIGNED contains at least one valid signature made during the time window specified by a certificate included in any CERTS object.

For details about the valid signatures, the user **MUST** inspect the VERIFICATIONS output.

If no CERTS are supplied, sop inline-verify fails with MISSING_ARG.

If no valid signatures are found, sop inline-verify fails with NO_SIGNATURE and emits nothing on standard output.

See Section 11.1 for more details about signature verification.

Example:

(In this example, we see signature verification succeed first, and then fail on a modified version of the message.)

```
$ sop inline-verify -- alice.pgp < message.txt
Hello, world!
$ echo $?
0
$ sed s/Hello/Goodbye/ < message.txt | sop inline-verify -- alice.pgp
$ echo $?
3
$
```

3.13. inline-sign: Create an Inline-Signed Message

```
sop inline-sign [--no-armor]
     [--with-key-password=PASSWORD...]
     [--as={binary|text|clearsigned}]
     [--] KEYS [KEYS...]
```

*Standard Input: DATA (Section 5.11)

*Standard Output: INLINESIGNED (Section 5.5)

Exactly one signature will be made by each key in the supplied KEYS arguments.

The generated output stream will be an inline-signed message, by default producing an OpenPGP "Signed Message" packet stream.

--as defaults to binary. If --as= is set to either text or clearsigned, and the input DATA is not valid UTF-8 (Section 9.7), sop inline-sign fails with EXPECTED_TEXT.

--as=binary **SHOULD** result in OpenPGP signatures of type 0x00 ("Signature of a binary document"). --as=text **SHOULD** result in an OpenPGP signature of type 0x01 ("Signature of a canonical text document"). See Section 5.2.1 of [RFC4880] for more details. --as=clearsigned **SHOULD** behave the same way as --as=text except that it produces an output stream using the Cleartext Signature Framework (see Section 7 of [RFC4880] and Section 9.5).

If both --no-armor and --as=clearsigned are supplied, sop inline-sign fails with INCOMPATIBLE_OPTIONS.

If the signing key material in any key in the KEYS objects is password-protected, sop inline-sign **SHOULD** try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the PASSWORD options unlock the key (or if no such option is supplied), sop inline-sign will fail with KEY_IS_PROTECTED. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 5). Note also the guidance for retrying variants of a non-human-readable password in Section 9.8.2.

If any key in the KEYS objects is not capable of producing a signature, sop inline-sign will fail with KEY_CANNOT_SIGN.

sop inline-sign **MUST NOT** produce any extra signatures beyond those from KEYS objects supplied on the command line.

Example:

```
$ sop inline-sign --as=clearsigned alice.sec < message.txt > message-sig
$ head -n5 < message-signed.txt
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

This is the message.
-----BEGIN PGP SIGNATURE-----
$
```

4. Input String Types

Some material is passed to sop directly as a string on the command line.

4.1. DATE

An ISO-8601 formatted timestamp with time zone, or the special value now to indicate the current system time.

Examples:

*now

*2019-10-29T12:11:04+00:00

*2019-10-24T23:48:29Z

*20191029T121104Z

In some cases where used to specify lower and upper boundaries, a DATE value can be set to - to indicate "no time limit".

A flexible implementation of sop **MAY** accept date inputs in other unambiguous forms.

Note that whenever sop emits a timestamp (e.g. in Section 5.10) it **MUST** produce only a UTC-based ISO-8601 compliant representation with a resolution of one second, using the literal Z suffix to indicate timezone.

4.2. USERID

This is an arbitrary UTF-8 string (Section 9.7). By convention, most User IDs are of the form Display Name <email.address@example.com>, but they do not need to be.

4.3. SUBCOMMAND

This is an ASCII string that matches the name of one of the subcommands listed in Section 3.

4.4. PROFILE

Some sop subcommands can accept a --profile option, which takes as an argument the name of a profile.

A profile name is a UTF-8 string that has no whitespace in it.

Which profiles are available depends on the sop implementation.

Similar to OpenPGP Notation names, profile names are divided into two namespaces: the IETF namespace and the user namespace. A profile name in the user namespace ends with the @ character (0x40) followed by a DNS domain name. A profile name in the IETF namespace does not have an @ character.

A profile name in the user space is owned and controlled by the owner of the domain in the suffix. A sop implementation that implements a user profile but does not own the domain in question **SHOULD** hew as closely as possible to the semantics described by the owner of the domain.

A profile name in the IETF namespace that begins with the string rfc should have semantics that hew as closely as possible to the referenced RFC. Similarly, a profile name in the IETF namespace that begins with the string draft- should have semantics that hew as closely as possible to the referenced Internet Draft.

The reserved profile name default in the IETF namespace simply refers to the implementation's default choices.

Note that this profile mechanism is intended to provide a limited way for an implementation to select among a small set of options that the implementer has vetted and is satisfied with. It is not intended to provide an arbitrary channel for complex configuration, and a sop implementation **MUST NOT** use it in that way.

5. Input/Output Indirect Types

Some material is passed to sop indirectly, typically by referring to a filename containing the data in question. This type of data may also be passed to sop on Standard Input, or delivered by sop to Standard Output.

If any input data is specified explicitly to be read from a file that does not exist, sop will fail with MISSING_INPUT.

If any input data does not meet the requirements described below, sop will fail with BAD_DATA.

5.1. Special Designators for Indirect Types

An indirect argument or parameter that starts with "@" (COMMERCIAL AT, U+0040) is not treated as a filename, but is reserved for special handling, based on the prefix that follows the @. We describe two of those prefixes (@ENV: and @FD:) here. A sop implementation that receives such a special designator but does not know how to handle a given prefix in that context **MUST** fail with UNSUPPORTED_SPECIAL_PREFIX.

If the filename for any indirect material used as input has the special form @ENV:xxx, then contents of environment variable $xxx is used instead of looking in the filesystem. @ENV is for input only: if the prefix @ENV: is used for any output argument, sop fails with UNSUPPORTED_SPECIAL_PREFIX.

If the filename for any indirect material used as either input or output has the special form @FD:nnn where nnn is a decimal integer, then the associated data is read from file descriptor nnn.

See Section 9.9 for more details about safe handling of these special designators.

5.2. CERTS

One or more OpenPGP certificates (Section 11.1 of [I-D.ietf-openpgp-crypto-refresh-07]), aka "Transferable Public Key". May be armored (see Section 9.4).

Although some existing workflows may prefer to use one CERTS object with multiple certificates in it (a "keyring"), supplying exactly one certificate per CERTS input will make error reporting clearer and easier.

5.3. KEYS

One or more OpenPGP Transferable Secret Keys (Section 11.2 of [I-D.ietf-openpgp-crypto-refresh-07]). May be armored (see Section 9.4).

Secret key material is often locked with a password to ensure that it cannot be simply copied and reused. If any secret key material is locked with a password and no --with-key-password option is supplied, sop may fail with error KEY_IS_PROTECTED. However, when a cleartext secret key (that is, one not locked with a password) is available, sop should always be able to use it, whether a --with-key-password option is supplied or not.

Although some existing workflows may prefer to use one KEYS object with multiple keys in it (a "secret keyring"), supplying exactly one key per KEYS input will make error reporting clearer and easier.

5.4. CIPHERTEXT

sop accepts only a restricted subset of the arbitrarily-nested grammar allowed by the OpenPGP Messages definition (Section 11.3 of [I-D.ietf-openpgp-crypto-refresh-07]).

In particular, it accepts and generates only:

An OpenPGP message, consisting of a sequence of PKESKs (Section 5.1 of [I-D.ietf-openpgp-crypto-refresh-07]) and SKESKs (Section 5.3 of [I-D.ietf-openpgp-crypto-refresh-07]), followed by one SEIPD (Section 5.13 of [I-D.ietf-openpgp-crypto-refresh-07]).

The SEIPD can decrypt into one of two things:

* "Maybe Signed Data" (see below), or
* Compressed data packet that contains "Maybe Signed Data"

"Maybe Signed Data" is a sequence of:

* N (zero or more) one-pass signature packets, followed by
* zero or more signature packets, followed by
* one Literal data packet, followed by
* N signature packets (corresponding to the outer one-pass signatures packets)

FIXME: does any tool do compression inside signing? Do we need to handle that?

May be armored (see Section 9.4).

5.5. INLINESIGNED

An inline-signed message may take any one of three different forms:

* A binary sequence of OpenPGP packets that matches a subset of the "Signed Message" element in the grammar in Section 11.3 of [I-D.ietf-openpgp-crypto-refresh-07]

* The same sequence of packets, but ASCII-armored (see Section 9.4)

* A message using the Cleartext Signature Framework described in Section 7 of [I-D.ietf-openpgp-crypto-refresh-07]

The subset of the packet grammar expected in the first two forms consists of either:

* a series of Signature packets followed by a Literal Data packet

* a series of One-Pass Signature (OPS) packets, followed by one Literal Data packet, followed by an equal number of Signature packets corresponding to the OPS packets

When the message is in the third form (Cleartext Signature Framework), it has the following properties:

* The stream **SHOULD** consist solely of UTF-8 text

* Every Signature packet found in the stream **SHOULD** have Signature Type 0x01 (canonical text document).

* It **SHOULD NOT** contain leading text (before the -----BEGIN PGP SIGNED MESSAGE----- cleartext header) or trailing text (after the -----END PGP SIGNATURE----- armor tail).

While some OpenPGP implementations **MAY** produce more complicated inline signed messages, a sop implementation **SHOULD** limit itself to producing these straightforward forms.

5.6. SIGNATURES

One or more OpenPGP Signature packets. May be armored (see Section 9.4).

5.7. SESSIONKEY

This documentation uses the GnuPG defacto ASCII representation:

ALGONUM:HEXKEY

where ALGONUM is the decimal value associated with the OpenPGP Symmetric Key Algorithms (Section 9.3 of [I-D.ietf-openpgp-crypto-refresh-07]) and HEXKEY is the hexadecimal representation of the binary key.

Example AES-256 session key:

```
9:FCA4BEAF687F48059CACC14FB019125CD57392BAB7037C707835925CBF9F7BCD
```

A sop implementation **SHOULD** produce session key data in this format. When consuming such a session key, sop **SHOULD** be willing to accept either upper or lower case hexadecimal digits, and to gracefully ignore any trailing whitespace.

5.8. MICALG

This output-only type indicates the cryptographic digest used when making a signature. It is useful specifically when generating signed PGP/MIME objects, which want a micalg= parameter for the multipart/signed content type as described in Section 5 of [RFC3156].

It will typically be a string like pgp-sha512, but in some situations (multiple signatures using different digests) it will be the empty string. If the user of sop is assembling a PGP/MIME signed object, and the MICALG output is the empty string, the user should omit the micalg= parameter entirely.

5.9. PASSWORD

This input-only is expected to be a UTF-8 string (Section 9.7), but for sop decrypt, any bytestring that the user supplies will be accepted. Note the details in sop encrypt and sop decrypt about trailing whitespace!

See also Section 9.8 for more discussion.

5.10. VERIFICATIONS

This output-only type consists of one line per successful signature verification. Each line has three structured fields delimited by a single space, followed by arbitrary text to the end of the line that forms a message describing the verification.

* ISO-8601 UTC datestamp of the signature, to one second precision, using the Z suffix

* Fingerprint of the signing key (may be a subkey)

* Fingerprint of primary key of signing certificate (if signed by primary key, same as the previous field)

* (optional) a string describing the mode of the signature, either mode:text or mode:binary

* message describing the verification (free form)

Note that while Section 4.1 permits a sop implementation to accept other unambiguous date representations, its date output here **MUST** be a strict ISO-8601 UTC date timestamp. In particular:

* the date and time fields **MUST** be separated by T, not by whitespace, since whitespace is used as a delimiter

* the time **MUST** be emitted in UTC, with the explicit suffix Z

* the time **MUST** be emitted with one-second precision

Example:

```
2019-10-24T23:48:29Z C90E6D36200A1B922A1509E77618196529AE5FF8 C4BC2DDB38
```

5.11. DATA

Cleartext, arbitrary data. This is either a bytestream or UTF-8 text.

It **MUST** only be UTF-8 text in the case of input supplied to sop sign --as=text or sop encrypt --as=text. If sop receives DATA containing non-UTF-8 octets in this case, it will fail (see Section 9.7) with EXPECTED_TEXT.

5.12. PROFILELIST

This output-only type consists of simple UTF-8 textual output, with one line per profile. Each line consists of the profile name optionally followed by a colon (0x31), a space (0x20), and a brief human-readable description of the intended semantics of the profile. Each line may be at most 1000 bytes, and no more than 4 profiles may be listed.

These limits are intended to force sop implementers to make hard decisions and to keep things simple.

The first profile **MAY** be explicitly named default. If it is not named default, then default is an alias for the first profile listed. No profile after the first listed may be named default.

See Section 4.4 for more discussion about the namespace and intended semantics of each profile.

6. Failure Modes

sop return codes have both mnemonics and numeric values.

When sop succeeds, it will return 0 (OK) and emit nothing to Standard Error. When sop fails, it fails with a non-zero return code, and emits one or more warning messages on Standard Error. Known return codes include:

Value	Mnemonic	Meaning
0	OK	Success
3	NO_SIGNATURE	No acceptable signatures found (sop verify)
13	UNSUPPORTED_ASYMMETRIC_ALGO	Asymmetric algorithm unsupported (sop encrypt)
17	CERT_CANNOT_ENCRYPT	Certificate not encryption-capable (e.g., expired, revoked, unacceptable usage flags) (sop encrypt)
19	MISSING_ARG	Missing required argument
23	INCOMPLETE_VERIFICATION	Incomplete verification instructions (sop decrypt)
29	CANNOT_DECRYPT	Unable to decrypt (sop decrypt)
31	PASSWORD_NOT_HUMAN_READABLE	Non-UTF-8 or otherwise unreliable password (sop encrypt, sop generate-key)
37	UNSUPPORTED_OPTION	Unsupported option
41	BAD_DATA	Invalid data type (no secret key where KEYS expected, etc)
53	EXPECTED_TEXT	Non-text input where text expected
59	OUTPUT_EXISTS	Output file already exists
61	MISSING_INPUT	Input file does not exist
67	KEY_IS_PROTECTED	A KEYS input is password-protected (locked), and sop cannot unlock it with any of the --with-key-password options
69	UNSUPPORTED_SUBCOMMAND	Unsupported subcommand
71	UNSUPPORTED_SPECIAL_PREFIX	An indirect parameter is a special designator (it starts with @) but sop does not know how to handle the prefix
73	AMBIGUOUS_INPUT	A indirect input parameter is a special designator (it starts with @), and a filename matching the designator is actually present

Value	Mnemonic	Meaning
79	KEY_CANNOT_SIGN	Key not signature-capable (e.g., expired, revoked, unacceptable usage flags) (sop sign and sop encrypt with --sign-with)
83	INCOMPATIBLE_OPTIONS	Options were supplied that are incompatible with each other
89	UNSUPPORTED_PROFILE	The requested profile is unsupported (sop generate-key, sop encrypt), or the indicated subcommand does not accept profiles (sop list-profiles)

Table 1: Error return codes

If a sop implementation fails in some way not contemplated by this document, it **MAY** return any non-zero error code, not only those listed above.

7. Known Implementations

The following implementations are known at the time of this draft:

Project name	URL	cli name	notes
Sequoia SOP	https://gitlab.com/sequoia-pgp/sequoia-sop	sqop	Implemented in Rust using the sequoia-openpgp crate
gosop	https://github.com/ProtonMail/gosop	gosop	Implemented in golang (Go) using GOpenPGP
PGPainless SOP	https://codeberg.org/PGPainless/pgpainless/src/branch/master/pgpainless-sop	pgpainless-cli	Implemented in Java using PGPainless
sopgpy	https://gitlab.com/sequoia-pgp/openpgp-interoperability-test-suite/-/blob/main/glue/sopgpy	sopgpy	Implemented in Python using PGPy
sop-openpgp.js	https://github.com/openpgpjs/sop-openpgpjs	sop-openpgp	Implemented in JavaScript using OpenPGP.js
gpgme-sop	https://gitlab.com/sequoia-pgp/gpgme-sop	gpgme-sop	A Rust wrapper around the

Project name	URL	cli name	notes
			gpgme C library
RNP-sop	https://gitlab.com/sequoia-pgp/rnp-sop	rnp-sop	A Rust wrapper around the librnp C library
dkg-sop	https://git.savannah.nongnu.org/cgit/dkgpg.git/tree/tools/dkg-sop.cc	dkg-sop	Implemented in C++ using the LibTMCG library

Table 2: Known implementations

8. Alternate Interfaces

This draft primarily defines a command line interface, but future versions may try to outline a comparable idiomatic interface for C or some other widely-used programming language.

Comparable idiomatic interfaces are already active in the wild for different programming languages, in particular:

*Rust: [RUST-SOP]

*Java: [SOP-JAVA]

*Python: [PYTHON-SOP]

These programmatic interfaces are typically coupled with a wrapper that can automatically generate a command-line tool compatible with this draft.

An implementation that uses one of these languages should target the corresponding idiomatic interface for ease of development and interoperability.

9. Guidance for Implementers

sop uses a few assumptions that implementers might want to consider.

9.1. One OpenPGP Message at a Time

sop is intended to be a simple tool that operates on one OpenPGP object at a time. It should be composable, if you want to use it to deal with multiple OpenPGP objects.

FIXME: discuss what this means for streaming. The stdio interface doesn't necessarily imply streamed output.

9.2. Simplified Subset of OpenPGP Message

While the formal grammar for OpenPGP Message is arbitrarily nestable, sop constrains itself to what it sees as a single "layer" (see Section 5.4).

This is a deliberate choice, because it is what most consumers expect. Also, if an arbitrarily-nested structure is parsed with a recursive algorithm, this risks a denial of service vulnerability. sop intends to be implementable with a parser that defensively declines to do recursive descent into an OpenPGP Message.

Note that an implementation of sop decrypt **MAY** choose to handle more complex structures, but if it does, it should document the other structures it handles and why it chooses to do so. We can use such documentation to improve future versions of this spec.

9.3. Validate Signatures Only from Known Signers

There are generally only a few signers who are relevant for a given OpenPGP message. When verifying signatures, sop expects that the caller can identify those relevant signers ahead of time.

9.4. OpenPGP Inputs can be either Binary or ASCII-armored

OpenPGP material on input can be in either ASCII-armored or binary form. This is a deliberate choice because there are typical scenarios where the program can't predict which form will appear. Expecting the caller of sop to detect the form and adjust accordingly seems both redundant and error-prone.

The simple way to detect possible ASCII-armoring is to see whether the high bit of the first octet is set: Section 4.2 of [RFC4880] indicates that bit 7 is always one in the first octet of an OpenPGP packet. In standard ASCII-armor, the first character is "-" (HYPHEN-MINUS, U+002D), so the high bit should be cleared.

When considering an input as ASCII-armored OpenPGP material, sop **MAY** reject an input based on any of the following variations (see Section 6.2 of [RFC4880] for precise definitions):

* An unknown Armor Header Line

* Any text before the Armor Header Line

* Malformed lines in the Armor Headers section

* Any non-whitespace data after the Armor Tail

* Any Radix-64 encoded line with more than 76 characters

* Invalid characters in the Radix-64-encoded data

* An invalid Armor Checksum

* A mismatch between the Armor Header Line and the Armor Tail

For robustness, sop **SHOULD** be willing to ignore whitespace after the Armor Tail.

When considering OpenPGP material as input, regardless of whether it is ASCII-armored or binary, sop **SHOULD** reject any material that doesn't produce a valid stream of OpenPGP packets. For example, sop **SHOULD** raise an error if an OpenPGP packet header is malformed, or if there is trailing garbage after the end of a packet.

For a given type of OpenPGP input material (i.e., SIGNATURES, CERTS, KEYS, INLINESIGNED, or CIPHERTEXT), sop **SHOULD** also reject any input that does not conform to the expected packet stream. See Section 5 for the expected packet stream for different types.

9.5. Complexities of the Cleartext Signature Framework

sop prefers a detached signature as the baseline form of OpenPGP signature, but provides affordances for dealing with inline-signed messages (see INLINESIGNED, Section 5.5) as well.

The most complex form of inline-signed messages is the Cleartext Signature Framework (CSF). Handling the CSF structure requires parsing to delimit the multiple parts of the document, including at least:

* any preamble before the message

* the inline message header (delimiter line, OpenPGP headers)

* the message itself

* the divider between the message and the signature (including any OpenPGP headers there)

* the signature

* the divider that terminates the signature

* any suffix after the signature

Note also that the preamble or the suffix might be arbitrary text, and might themselves contain OpenPGP messages (whether signatures or otherwise).

If the parser that does this split differs in any way from the parser that does the verification, or parts of the message are confused, it would be possible to produce a verification status and an actual signed message that don't correspond to one another.

Blurred boundary problems like this can produce ugly attacks similar to those found in [EFAIL].

A user of sop that receives an inline-signed message (whether the message uses the CSF or not) can detach the signature from the message with sop inline-detach (see Section 3.11).

Alternately, the user can send the message through sop inline-verify to confirm required signatures, and then (if signatures are valid) supply its output to the consumer of the signed message.

9.6. Reliance on Supplied Certs and Keys

A truly stateless implementation may find that it spends more time validating the internal consistency of certificates and keys than it does on the actual object security operations.

For performance reasons, an implementation may choose to ignore validation on certificate and key material supplied to it. The security implications of doing so depend on how the certs and keys are managed outside of sop.

9.7. Text is always UTF-8

Various places in this specification require UTF-8 [RFC3629] when encoding text. sop implementations **SHOULD NOT** consider textual data in any other character encoding.

OpenPGP Implementations **MUST** already handle UTF-8, because various parts of [RFC4880] require it, including:

- User ID

- Notation name

- Reason for revocation

- ASCII-armor Comment: header

Dealing with messages in other charsets leads to weird security failures like [Charset-Switching], especially when the charset indication is not covered by any sort of cryptographic integrity check. Restricting textual data to UTF-8 universally across the OpenPGP ecosystem eliminates any such risk without losing functionality, since UTF-8 can encode all known characters.

9.8. Passwords are Human-Readable

Passwords are generally expected to be human-readable, as they are typically recorded and transmitted as human-visible, human-transferable strings. However, they are used in the OpenPGP protocol as bytestrings, so it is important to ensure that there is a reliable bidirectional mapping between strings and bytes. The maximally robust behavior here is for sop encrypt and sop generate-key (that is, commands that use a password to encrypt) to constrain the choice of passwords to strings that have such a mapping, and for sop decrypt and sop sign (and sop inline-sign, as well assop encrypt when decrypting a signing key; that is, commands that use a password to decrypt) to try multiple plausible versions of any password supplied by PASSWORD.

9.8.1. Generating Material with Human-Readable Passwords

When generating material based on a password, sop encrypt and sop generate-key enforce that the password is actually meaningfully human-transferable. In particular, an implementation generating material based on a new paasword **SHOULD** apply the following considerations to the supplied password:

* require UTF-8

* trim trailing whitespace

Some sop encrypt and sop generate-key implementations may make even more strict requirements on input to ensure that they are transferable between humans in a robust way.

For example, a more strict sop encrypt or sop generate-key **MAY** also:

* forbid leading whitespace

* forbid non-printing characters other than SPACE (U+0020), such as ZERO WIDTH NON-JOINER (U+200C) or TAB (U+0009)

* require the password to be in Unicode Normal Form C ([UNICODE-NORMALIZATION])

Violations of these more-strict policies **SHOULD** result in an error of PASSWORD_NOT_HUMAN_READABLE.

A sop encrypt or sop generate-key implementation typically **SHOULD NOT** attempt enforce a minimum "password strength", but in the event that some implementation does, it **MUST NOT** represent a weak password with PASSWORD_NOT_HUMAN_READABLE.

9.8.2. Consuming Password-protected Material

When sop decrypt receives a PASSWORD input, either from a --with-key-password or --with-password option, it sees its content as a bytestring. sop sign also sees the content of any PASSWORD input supplied to its --with-key-password option as a bytestring. If the bytestring fails to work as a password, but ends in UTF-8 whitespace, it will try again with the trailing whitespace removed. This handles a common pattern of using a file with a final newline, for example. The pattern here is one of robustness in the face of typical errors in human-transferred textual data.

A more robust sop decrypt or sop sign implementation that finds neither of the above two attempts work for a given PASSWORD **MAY** try additional variations if they produce a different bytestring, such as:

* trimming any leading whitespace, if discovered

* trimming any internal non-printable characters other than SPACE (U+0020)

* converting the supplied PASSWORD into Unicode Normal Form C ([UNICODE-NORMALIZATION])

A sop decrypt or sop sign implementation that stages multiple decryption attempts like this **SHOULD** consider the computational resources consumed by each attempt, to avoid presenting an attack surface for resource exhaustion in the face of a non-standard PASSWORD input.

9.9. Be Careful with Special Designators

As documented in Section 5.1, special designators for indirect inputs like @ENV: and @FD: (and indirect outputs using @FD:) warrant some special/cautious handling.

For one thing, it's conceivable that the filesystem could contain a file with these literal names. If sop receives an indirect output parameter that starts with an "@" (COMMERCIAL AT, U+0040) it **MUST NOT** write to the filesystem for that parameter. A sop implementation that receives such a parameter as input **MAY** test for the presence of such a file in the filesystem and fail with AMBIGUOUS_INPUT to warn the user of the ambiguity and possible confusion.

These special designators are likely to be used to pass sensitive data (like secret key material or passwords) so that it doesn't need to touch the filesystem. Given this sensitivity, sop should be careful with such an input, and minimize its leakage to other processes. In particular, sop **SHOULD NOT** leak any environment variable identified by @ENV: or file descriptor identified by @FD: to any subprocess unless the subprocess specifically needs access to that data.

10. Guidance for Consumers

While sop is originally conceived of as an interface for interoperability testing, it's conceivable that an application that uses OpenPGP for object security would want to use it.

FIXME: more guidance for how to use such a tool safely and efficiently goes here.

FIXME: if an encrypted OpenPGP message arrives without metadata, it is difficult to know which signers to consider when decrypting. How do we do this efficiently without invoking sop decrypt twice, once without --verify-* and again with the expected identity material?

10.1. Choosing Between --as=text and --as=binary

A program that invokes sop to generate an OpenPGP signature typically needs to decide whether it is making a text or binary signature.

By default, sop will make a binary signature. The caller of sop sign should choose --as=text only when it knows that:

* the data being signed is in fact textual, and encoded in UTF-8, and

* the signed data might be transmitted to the recipient (the verifier of the signature) over a channel that has the propensity to transform line-endings.

Examples of such channels include FTP ([RFC0959]) and SMTP ([RFC5321]).

10.2. Special Designators and Unusual Filenames

In some cases, a user of sop might want to pass all the files in a given directory as positional parameters (e.g., a list of CERTS files to test a signature against).

If one of the files has a name that starts with --, it might be confused by sop for an option. If one of the files has a name that starts with @, it might be confused by sop as a special designator (Section 5.1).

If the user wants to deliberately refer to such an ambiguously-named file in the filesystem, they should prefix the filename with ./ or use an absolute path.

Any specific @FD: special designator **SHOULD NOT** be supplied more than once to an invocation of sop. If a sop invocation sees multiple copies of a specific @FD:n input (e.g., sop sign @FD:3 @FD:3), it **MAY** fail with MISSING_INPUT even if file descriptor 3 contains a valid KEYS, because the bytestream for the KEYS was consumed by the first argument. Doubling up on the same @FD: for output (e.g., sop decrypt --session-key-out=@FD:3 --verifications-out=@FD:3) also results in an ambiguous data stream.

11. Security Considerations

The OpenPGP object security model is typically used for confidentiality and authenticity purposes.

11.1. Signature Verification

In many contexts, an OpenPGP signature is verified to prove the origin and integrity of an underlying object.

When sop checks a signature (e.g. via sop verify or sop decrypt --verify-with), it **MUST NOT** consider it to be verified unless all of these conditions are met:

* The signature must be made by a signing-capable public key that is present in one of the supplied certificates

* The certificate and signing subkey must have been created before or at the signature time

* The certificate and signing subkey must not have been expired at the signature time

* The certificate and signing subkey must not be revoked with a "hard" revocation

* If the certificate or signing subkey is revoked with a "soft" revocation, then the signature time must predate the revocation

* The signing subkey must be properly bound to the primary key, and cross-signed

* The signature (and any dependent signature, such as the cross-sig or subkey binding signatures) must be made with strong cryptographic algorithms (e.g., not MD5 or a 1024-bit RSA key)

Implementers **MAY** also consider other factors in addition to the origin and authenticity, including application-specific information.

For example, consider the application domain of checking software updates. If software package Foo version 13.3.2 was signed on 2019-10-04, and the user receives a copy of Foo version 12.4.8 that was signed on 2019-10-16, it may be authentic and have a more recent signature date. But it is not an upgrade (12.4.8 < 13.3.2), and therefore it should not be applied automatically.

In such cases, it is critical that the application confirms that the other information verified is *also* protected by the relevant OpenPGP signature.

Signature validity is a complex topic (see for example the discussion at [DISPLAYING-SIGNATURES]), and this documentation cannot list all possible details.

11.2. Compression

The interface as currently specified does not allow for control of compression. Compressing and encrypting data that may contain both attacker-supplied material and sensitive material could leak information about the sensitive material (see the CRIME attack).

Unless an application knows for sure that no attacker-supplied material is present in the input, it should not compress during encryption.

12. Privacy Considerations

Material produced by sop encrypt may be placed on an untrusted machine (e.g., sent through the public SMTP network). That material may contain metadata that leaks associational information (e.g., recipient identifiers in PKESK packets (Section 5.1 of [I-D.ietf-openpgp-crypto-refresh-07])). FIXME: document things like PURBs and --hidden-recipient)

12.1. Object Security vs. Transport Security

OpenPGP offers an object security model, but says little to nothing about how the secured objects get to the relevant parties.

When sending or receiving OpenPGP material, the implementer should consider what privacy leakage is implicit with the transport.

13. References

13.1. Normative References

[I-D.ietf-openpgp-crypto-refresh-07] Wouters, P., Huigens, D., Winter, J., and N. Yutaka, "OpenPGP Message Format", Work in Progress, Internet-Draft, draft-ietf-openpgp-crypto-refresh-07, 23 October 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-crypto-refresh-07>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC3156] Elkins, M., Del Torto, D., Levien, R., and T. Roessler, "MIME Security with OpenPGP", RFC 3156, DOI 10.17487/RFC3156, August 2001, <https://www.rfc-editor.org/rfc/rfc3156>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/rfc/rfc3629>.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <https://www.rfc-editor.org/rfc/rfc4880>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

13.2. Informative References

[Charset-Switching] Gillmor, D. K., "Inline PGP Considered Harmful", 24 February 2014, <https://dkg.fifthhorseman.net/notes/inline-pgp-harmful/>.

[DISPLAYING-SIGNATURES] Brunschwig, P., "On Displaying Signatures", n.d., <https://admin.hostpoint.ch/pipermail/enigmail-users_enigmail.net/2017-November/004683.html>.

[EFAIL] Poddebniak, D. and C. Dresen, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", n.d., <https://efail.de>.

[I-D.draft-bre-openpgp-samples-01] Einarsson, B. R., "juga", and D. K. Gillmor, "OpenPGP Example Keys and Certificates", Work in Progress, Internet-Draft, draft-bre-openpgp-samples-01, 20 December 2019, <https://

datatracker.ietf.org/doc/html/draft-bre-openpgp-samples-01>.

[OpenPGP-Interoperability-Test-Suite] "OpenPGP Interoperability Test Suite", 25 October 2021, <https://tests.sequoia-pgp.org/>.

[PYTHON-SOP] Gillmor, D., "SOP for python", n.d., <https://pypi.org/project/sop/>.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, DOI 10.17487/RFC0959, October 1985, <https://www.rfc-editor.org/rfc/rfc959>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/rfc/rfc5321>.

[RUST-SOP] Winter, J., "A Rust implementation of the Stateless OpenPGP Protocol", n.d., <https://sequoia-pgp.gitlab.io/sop-rs/>.

[SEMVER] Preston-Werner, T., "Semantic Versioning 2.0.0", 18 June 2013, <https://semver.org/>.

[SOP-JAVA] Schaub, P., "Stateless OpenPGP Protocol for Java.", n.d., <https://github.com/pgpainless/sop-java>.

[UNICODE-NORMALIZATION] Whistler, K., "Unicode Normalization Forms", 4 February 2019, <https://unicode.org/reports/tr15/>.

Appendix A. Acknowledgements

This work was inspired by Justus Winter's [OpenPGP-Interoperability-Test-Suite].

The following people contributed helpful feedback and considerations to this draft, but are not responsible for its problems:

* Allan Nordhoey

* Antoine Beaupre

* Edwin Taylor

* Heiko Schaefer

* Jameson Rollins

* Justus Winter

* Paul Schaub

* Vincent Breitmoser

Appendix B. Future Work

* certificate transformation into popular publication forms:

  - WKD

  - DANE OPENPGPKEY

  - Autocrypt

* sop encrypt -- specify compression? (see Section 11.2)

* sop encrypt -- specify padding policy/mechanism?

* sop decrypt -- how can it more safely handle zip bombs?

* sop decrypt -- what should it do when encountering weakly-encrypted (or unencrypted) input?

* sop encrypt -- minimize metadata (e.g. --throw-keyids)?

* specify an error if a DATE arrives as input without a time zone?

* add considerations about what it means for armored CERTS to contain multiple certificates -- multiple armorings? one big blob?

* do we need an interface or option (for performance?) with the semantics that sop doesn't validate certificates internally, it just accepts whatever's given as legit data? (see Section 9.6)

* do we need to be able to convert a message with a text-based signature to a CSF INLINESIGNED message? I'd rather not, given the additional complications.

Appendix C. Document History

C.1. Substantive Changes between -05 and -06:

* version: add --sop-spec argument

* encrypt: add --profile argument

C.2. Substantive Changes between -04 and -05:

* decrypt: change --verify-out to --verifications-out

*encrypt: add missing --with-key-password

*add the concept of "profiles", use with generate-key

*include table of known implementations

*VERIFICATIONS can now indicate the type of the signature (mode:text or mode:binary)

C.3. Substantive Changes between -03 and -04:

*Reinforce that PASSWORD and SESSIONKEY are indirect data types

*encrypt: remove --as=mime option

*Handle password-locked secret key material: add --with-key-password options to generate-key, sign, and decrypt.

*Introduce INLINESIGNED message type (Section 5.5)

*Rename detach-inband-signature-and-message to inline-detach, clarify its possible inputs

*Add inline-verify

*Add inline-sign

C.4. Substantive Changes between -02 and -03:

*Added --micalg-out parameter to sign

*Change from KEY to KEYS (permit multiple secret keys in each blob)

*New error code: KEY_CANNOT_SIGN

*version now has --backend and --extended options

C.5. Substantive Changes between -01 and -02:

*Added mnemonics for return codes

*decrypt should fail when asked to output to a pre-existing file

*Removed superfluous --armor option

*Much more specific about what armor --label=auto should do

*armor and dearmor are now fully idempotent, but work only well-formed OpenPGP streams

* Dropped armor --allow-nested

* Specified what encrypt --as= means

* New error code: KEY_IS_PROTECTED

* Documented expectations around human-readable, human-transferable passwords

* New subcommand: detach-inband-signature-and-message

* More specific guidance about special designators like @FD: and @ENV:, including new error codes UNSUPPORTED_SPECIAL_PREFIX and AMBIGUOUS_INPUT

C.6. Substantive Changes between -00 and -01:

* Changed generate subcommand to generate-key

* Changed convert subcommand to extract-cert

* Added "Input String Types" section as distinct from indirect I/O

* Made implicit arguments potentially explicit (e.g. sop armor --label=auto)

* Added --allow-nested to sop armor to make it idempotent by default

* Added fingerprint of signing (sub)key to VERIFICATIONS output

* Dropped --mode and --session-key arguments for sop encrypt (no plausible use, not needed for interop)

* Added --with-session-key argument to sop decrypt to allow for session-key-based decryption

* Added examples to each subcommand

* More detailed error codes for sop encrypt

* Move from CERT to CERTS (each CERTS argument might contain multiple certificates)

Author's Address

Daniel Kahn Gillmor
American Civil Liberties Union
125 Broad St.
New York, NY, 10004

United States of America

Email: dkg@fifthhorseman.net