An Advanced Introduction to GnuPG

Neal H. Walfield

August 1, 2017

Contents

Part I

Main Matter

Chapter 1

OpenPGP

GnuPG is an implementation of OpenPGP, an encryption standard published by the Internet Engineering Task Force (IETF). The IETF's main activity is the development and promotion of standards related to the Internet. Since its formation in 1986, the IETF has standardized many ubiquitous Internet protocols including the HyperText Transfer Protocol (HTTP), and the Transport Layer Security (TLS) protocol. Each standard is managed by a working group, and anyone can participate by joining the appropriate mailing list. The working group responsible for OpenPGP is fittingly called *The OpenPGP Working Group*.

OpenPGP consists of three main parts. First, OpenPGP specifies a collection of cryptographic algorithms for encrypting and decrypting data, generating and verifying digital signatures, and deriving keys from passwords (so-called *key derivication functions* or KDFs). These are built on top of more basic cryptographic building blocks like SHA-1 (a hash algorithm), AES (a symmetric cipher), and RSA (an asymmetric cipher, which is also known as a public-key algorithm). For the most part, the specification does not define these algorithms; it simply says which algorithms should be used where and how to use them. Second, OpenPGP defines a packet-based message format. This format is used not only for exchanging encrypted messages, but also for transferring keys and key meta-data. Finally, OpenPGP includes functionality to help manage keys. This functionality includes the ability to revoke a key, and to sign keys.

The first version of the OpenPGP protocol was published in 1996 as RFC 1991. (Although, at that point it was still known as the PGP prototcol.) Since then, the protocol has undergone two major revisions. The most re-

cent version was published in 2007 as RFC 4880. In 2015, the OpenPGP community again reformed the OpenPGP working group to update the specification [1].

The major goals for the next version are: the deprecation of some old cryptographic algorithms like SHA-1, the introduction of some new cryptographic algorithms based on elliptic curves, the addition of modern message integrity protection in the form of something like Authenticated Encryption with Associated Data (AEAD), and an updated fingerprint format.

From an application programmer or user's perspective, the working group is not considering any major changes to the existing functionality; they are primarily tightening the standard's security and cleaning up a few issues. This is true even of OpenPGP's use of SHA-1, which, although SHA-1 has many flaws, is still considered safe in the way that OpenPGP uses it. That is, the changes are mostly to proactively—not reactively—address weaknesses. In the words of the cryptographer Peter Gutmann, "OpenPGP is still too good enough, there's lots of things there that you can nitpick but nothing really fatal, or even close to fatal" [2].

1.1 Data at Rest

OpenPGP is used to protect both data at rest as well as data in motion. Whereas data at rest refers to data that is stored, e.g., on a hard drive, data in motion refers to data that is transferred, e.g., via HTTP. Thus, an encryption scheme that only protects data in motion, such as TLS, removes the encryption on receipt; the data is only protected on the wire. Another way to think about the difference between data at rest and data in motion is that encryption that protects data at rest protects it in time and space whereas encryption that protects data in motion only protects it in space. Yet another way to think about the difference is that data at rest is to the `tar` or `zip` tools as data in motion is to HTTP or XMPP.

The decision to protect not only data in motion, but also data at rest using the same scheme significantly constrains the solution space. In particular, because data at rest may be accessed asynchronously with respect to the encryption, there is no possibility to negociate parameters on the fly.

Consider an encrypted backup. When you encrypt the data, you can only use the strongest encryption that is available at the time of the encryption. When you access the data 10 years later, your implementation needs

to support that now old encryption algorithm; there is no way to go back in time and say to your former self, "could you use this implementation instead?"

An additional consequence is that upgrading the cryptography becomes very difficult. It is not possible to completely deprecate old algorithms, because old messages (like our backup) still need to be decrypted. Similarly, since people continue to use old software, we often cannot use the latest and greatest encryption scheme, because they might not be able to decrypt the data!

Another result of this decision to protect data at rest is that enabling forward secrecy is not possible. Forward secrecy is an oft-lauded encryption property, which prevents old encrypted messages from being decrypted if the private key material is somehow compromised. Forward secrecy works by mutating the key material in time. This scheme is fine if you never need to decrypt old messages (as is typically the case for data transferred via HTTPS, say), but doesn't work at all for data at rest: if you want to decrypt some data a week later, nevermind 10 years later, then you won't be able to if you've destroyed the private key material needed to decrypt it!

Perfect secrecy becomes even more complicated when a user has multiple devices, and all devices should be able to decrypt all messages. OpenPGP doesn't require that those devices somehow synchronize their state after the private key is copied. But, some type of synchronization is necessary for forward secrecy.

This raises the question: why have a single algorithm for both data in motion and data at rest? The reason is that OpenPGP messages are often not stored on a trusted host or even processed on a trusted host before being stored. Consider email. Email is normally stored on a mail server. Even after the mail is read, it remains on the mail server so that it can be read later—potentially years later—on a different device. Thus, even assuming that we could harden the security of the transport layer, it is not clear that when the data is on a mail server, it is any less vulnerable than when it is on the wire. In fact, data breaches at huge companies entrusted with highly personal information from millions or even billions of users, such as Yahoo! and Adult Friend Finder, are evidence that this is not the case.

1.2 Unbuffered Message Processing

OpenPGP is designed to allow unbuffered message processing. This is partially achieved by mandating that message packets be sorted topologically. That is, if a packet has a dependency, that dependency precedes it in the message.

This property is important for several reasons. First, it allows an OpenPGP implementation to run on memory constrainted systems while being confident that the implementation can in practice process arbitrarily large messages. Second, it ensures that streaming tools can be used, e.g., something like `... | gpg -e -r key | ssh ...`. Finally, this property helps avoid some denial of service attacks, which might otherwise be possible by crafting a malicious message.

In practice, there are some limitations to the degree to which buffering can be avoided. Consider a pipeline in which a message is verified, and the output of the message is somehow processed. Because the OpenPGP implementation requires the whole message to verify it, to process this message in a streaming fashion, the OpenPGP implementation has to output the data before it has been verified. Now, if the consumer can't process the output in a way that can be reverted in the case of a validation failure, the consumer must first buffer the data. But, even if it is possible for the consumer to recover from a validation failure, it's probably error prone if only because code on an error path is rarely tested. Thus, although the OpenPGP implementation could avoid buffering data in this situation, it has merely shifted the burden.

Now, there are some more advanced cryptographic constructs, such as hash chaining, that make it possible to verify the data bit-by-bit. These techniques would help ensure that the consumer only processes verified data, which is an improvement over the status quo. But, they don't completely solve the problem, because they can't protect against message truncation.

1.3 OpenPGP Messages

An OpenPGP message is basically a sequences of packets. OpenPGP defines 17 different packet types that are used to not only encrypt and sign messages, but also to transfer keys and key signatures or certifications, which are used in the web of trust. The format is extensible, and this has

already been used to add new features.

An example of a packet type is the symmetrically encrypted data (SED) packet. A SED packet contains data that has been encrypted using a symmetric algorithm, such as AES. The contents of the packet are zero or more OpenPGP packets. That is, OpenPGP messages are nested; a SED packet is a container. Typically, a SED contains either a signature packet or a compressed data packet, which in turns holds a literal data packet, but the specification doesn't impose any limitations.

This flexibility in message composition is referred to as *agility*. It has both advantages and disadvantages.

A useful advantage that this flexibility offers is that the format can be used in unforeseen situations. For instance, the web key directory (WKD) uses the non-standard sign+encrypt+sign pattern to facilitate spam detection prior to decryption.

Two important disadvantages of this flexibility are that parsing OpenPGP messages is more complicated, and assigning meaning to unusual structures can be difficult. As an example of the latter, consider a message with two literal data packets, the first of which is signed. Assuming the signature is valid, should an implementation report that the message is valid? Probably not. The second part could have been forged. Alternatively a mail program could show both parts and indicate that only the first part is authentic. But, this requires educating the user to understand these nuances. Unfortunately educating users is known to be extremely difficult.

1.4 Encryption

Most lay people and even many technical people assume that encryption includes both an integrity check and authentication. In reality, encryption by itself provides neither. This assumption perhaps arises due to conditioning from web browsers that not only conflate the two concepts, but treat a connection secured with a self-signed certificate (which provides encryption, but not authentication), worse than those that use neither encryption nor authentication. Additionally, in recent years, the term end-to-end encryption has entered the mainstream. Although authentication is as important as encryption in such systems, only encryption is mentioned. Be that as it may, in OpenPGP, encryption and signing are separate, independent operations.

1.4.1 Hybrid Encryption

OpenPGP is a hybrid cryptosystem. A hybrid cryptosystem first encrypts data using a symmetric encryption algorithm like AES with a random so-called *session key*, and then encrypts the session key using the recipient's public key. The result is stored in a so-call *public-key encrypted session key* (PK-ESK) packet.

There are two important reasons for doing this as well as several additional advantages.

First, public key encryption is thousands of times slower than symmetric encryption. Since a session key is just a single block of data (which is N bits for an N bit RSA key), but the data to encrypt could be megabytes or even gigabytes large, this saves a lot of processing power.

Second, it is not unusual to encrypt a message to multiple recipients. The most obvious example of this is in the context of email where an encrypted email is sent to multiple people. But even in other contexts, having multiple recipients is not unusual. Specifically, when encrypting data to another party, most programs will also encrypt the data to the person doing the encryption so that the data remains readable and auditable.

An advantage of this approach is that it is possible to do message-based key escrow. Thus, a company wouldn't need to have access to each employee's private key, but whenever the employee decrypted an email, the session key could automatically be reencrypted with a special escrow key.

Similarly, if law enforcement forces you to reveal the encryption key for some messages, it is sufficient to provide the session keys for decrypting the subpoenaed messages. If you had instead provided your private key, law enforcement could read any message that had been encrypted to you. (In GnuPG, you can extract the session key using the `--show-session-key` option.)

Finally, using hybrid encryption, it is possible to encrypt to both public keys and passwords. To encrypt a message using a password, OpenPGP specifies a key derivation function (S2K), which is used to generate a symmetric key. (This is saved in a so-called *symmetric-key encrypted session key* (SK-ESK) packet.) OpenPGP allows the symmetric key to be used directly as the session key, but it can just as well be used to encrypt a session key. In practice, this is primarily interesting to ensure that the sender is able to later decrypt the contents of the message by also encrypting the session key to her public key.

1.4.2 Algorithm

Encryption in OpenPGP is a more or less standard hybrid encryption scheme:

1. A random *session key* is generated.
2. For each recipient, the OpenPGP implementation encrypts the session key using the recipient's public key, and emits a *public-key encrypted session key* (PK-ESK) packet.
3. If the data should be encrypted using a password, the same thing is done, but instead of emitted a PK-ESK packet, a *session-key encrypted session key* (SK-ESK) packet is emitted.
4. Encrypt the actual data using the session key.

OpenPGP supports multiple symmetric encryption algorithms. To determine which one to use, the OpenPGP implementation selects one from the intersection of the recipients' preferred algorithms. This information isn't negotiated in real time with the recipients (even when this might in theory be possible), but is stored alongside the recipient's public key (specifically, in a user ID's self-signature). Typically, this is just a list of the algorithms that the OpenPGP implementation that generated the key supports at the time the key was created, but it can be updated to reflect changes in the implementation, and may be customized by expert users. Since all implementations are required to at least support TripleDES, and it appears implicitly at the end of the list, the intersection is never empty.

1.4.3 An Encrypted Message

To better understand how messages are laid out, the following example shows the innards of an encrypted message. This output was created using GnuPG's `--list-packets` option. `hot dump`, which is part of hOpenPGP, and `pgpdump` can do something similar.

```
$ echo 'Let us sojourn in Mantua!' | \
> gpg --encrypt -r juliet.capulet@gnupg.net | \
> gpg --list-packets
gpg: encrypted with 2048-bit RSA key, ID C1A010A1D38C4BB8, created 2017-07-07
```

```
      "Juliet Capulet <juliet.capulet@gnupg.net>"
gpg: encrypted with 2048-bit RSA key, ID 5B905AF0423ABB52, created 201
      "Romeo Montague <romeo.montague@gnupg.net>"
# off=0 ctb=85 tag=1 hlen=3 plen=268
:pubkey enc packet: version 3, algo 1, keyid C1A010A1D38C4BB8
data: [2046 bits]
# off=271 ctb=85 tag=1 hlen=3 plen=268
:pubkey enc packet: version 3, algo 1, keyid 5B905AF0423ABB52
data: [2046 bits]
# off=542 ctb=d2 tag=18 hlen=2 plen=85 new-ctb
:encrypted data packet:
length: 85
mdc_method: 2
# off=563 ctb=a3 tag=8 hlen=1 plen=0 indeterminate
:compressed packet: algo=2
# off=565 ctb=cb tag=11 hlen=2 plen=32 new-ctb
:literal data packet:
mode b (62), created 1499445579, name="",
raw data: 26 bytes
```

The example shows a message that Romeo encrypted to Juliet. (Due to limitations of the OpenPGP format—OpenPGP only supports timestamps between 1970 and 2106—Romeo forward dated the creation time of his key.) The first thing that we notice is that even though Romeo only specified a single recipient (using the `-r` option), the message is encrypted to two keys: his and Juliet's. This is because Romeo has the `encrypt-to` option set in his `gpg.conf` file so that he can always read messages that he encrypts to someone else.

Packet Metadata

After listing the recipients, `gpg` outputs each packet. Each packet starts with a line preceded by a `#`. This line shows some meta-data and the packet's header. Specifically, `off` indicates the offset of the packet within the stream (this may not be accurate if there are compressed packets); `ctb` (Content Tag Byte) includes the type of the packet, and some information about the length of the packet (if this is a new format packet, then `new-ctb` will appear towards the end of the line); `tag` is the type of the packet as ex-

tracted from the `ctb`; and, `hlen` and `plen` are the header and body lengths, respectively.

Sometimes the length of a packet is not known apriori. In this case, `plen` will be 0 and `indeterminate` or `partial` will appear towards the end of the line. This can occur when the data is streamed. `indeterminate` means that all data until the end of the message belongs to this packet; `partial` means the packet uses a chunked encoding method to encode the data. The mechanism is similar to HTTP's chunked transfer encoding method. These encoding schemes are essential for supporting unbuffered operations. See Section 4.2.2.4 of RFC 4880 for more details.

The PK-ESK Packets

The first two packets in the message are PK-ESK packets. Each of these holds the session key encrypted to a recipient. A PK-ESK packet also includes the 64-bit key ID of key that the session key was encrypt to.

If the key ID wasn't included, then a recipient wouldn't know whether a given PK-ESK packet is encrypted with her or someone else's key and she would just have to try to decrypt them one by one. The obvious consequence is that CPU cycles could be wasted. But, the more important reason for avoiding a decryption attempt is that the user might have to unlock multiple private keys. This can seriously impact an application's usability.

Avoiding this UX annoyance by including the key ID in the PK-ESK has a cost: it leaks meta-data. In practice, however, this information is exposed in other places, e.g., at the SMTP level. Nevertheless, OpenPGP provides a mechanism to hide this meta-data by setting the key ID to `0`, which means the key ID is speculative. Such key IDs are also referred to as wild card key IDs.

A speculative key ID can be set in GnuPG by either specifying `--throw-keyids` to clear the key ID field for all recipients, or `--hidden-recipient` in place of `--recipient` to clear the key ID field for a particular recipient.

The Encrypted Data Packet

Immediately following the PK-ESK packets is an encrypted data packet. This ordering is mandatory: it ensures that buffering is not required, because the key needed to decrypt the packet is stored prior to the data that it decrypts. As already mentioned, an encrypted data packet is a container,

which contains 0 or more OpenPGP packets. This is not obvious from the output of the `--list-packets` command, because it doesn't show the message's tree structure. In this case, as is usually the case, the encrypted data packet contains a single packet.

In OpenPGP, there are actually two types of encrypted data packets: Symmetrically Encrypted Data (SED) packets and Symmetrically Encrypted Integrity Protected data (SEIP) packets. Although the former are technically allowed by the standard, they are deprecated in practice due to security concerns. For instance, it is possible to conduct an oracle attack [3], and message extension and deletion attacks are also possible. Consequently, when GnuPG encounters such a packet, it emits a warning. GnuPG itself will not emit an encrypted packet without integrity protection.

We can see that the encrypted data packet includes integrity protection based on the packet's tag (18 instead of 9), and the presence of the `mdc_method` field in the above output.

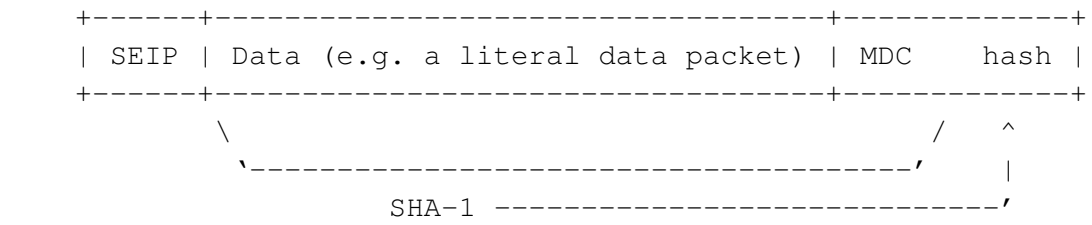
1. Modification Detection Codes

   MDC stands for Modification Detection Code. Like a message authentication code (MAC), an MDC can verify a message's integrity. But, unlike a MAC, an MDC doesn't say anything about its authenticity. A common criticism leveled at the MDC system is that using an HMAC would have been better since it is better understood. Ignoring that the MDC system has proven to be sufficient for its intended purpose, using an HMAC wasn't really an option when the problem was discussed: HMACs and MDCs were developed concurrently. (For more historical notes, see [4].)

   Prior to the introduction of the MDC system in RFC 4880, it was only possible to reliably detect integrity violations using signatures. Signatures, however, have the disadvantage that they expose the signer's identity, which is sometimes undesirable.

   MDC works by computing the SHA-1 over the clear text and the head of the MDC packet. (The rest of the MDC packet is the computed hash.) That is, the hash effectively violates the packet framing. But, this is exactly the behavior that is required to fully ensure the data's integrity: by also including the head of the MDC packet in the hash, extension and removal attacks are mitigated. The following example

illustrates how it works:



The `mdc_method` parameter above seems to suggest that there are multiple MDC methods. This is not the case, and was explicitly avoided to prevent downgrade and cross-grade attacks; the value of 2 is simply SHA-1's OpenPGP algorithm identifier. But even though SHA-1 has since been broken, the relevant security properties for the MDC system remain intact. Nevertheless, the working group is considering replacing the MDC system with one based on Authenticated Encryption with Associated Data (AEAD), which has other useful properties.

As a final note, the MDC packet is not shown in the output of `--list-packets`. This is a technical limitation of GnuPG, which has to do with the way the MDC packet is processed. But, given that `--list-packets` is only a debugging interface and not intended for programmatic use, this limitation is unlikely to be fixed.

Compressed Packet

The compressed packet is nested within the encrypted packet. RFC 4880 specifies three different compression algorithms—ZIP, ZLIB, and BZip2—but notes that they are optional. But even though compression is not required, the RFC recommends it as an operationally useful (even if not rigorous) form of integrity protection. Unfortunately, it has been shown that compressing data prior to encryption can enable a chosen plaintext attack as demonstrated by the CRIME on TLS, and BREACH on HTTP attacks.

Literal Data

Nested within the compression packet is a literal data packet. A literal data packet contains not only the cleartext, but also a bit of metadata. In particular, a literal packet includes a formatting field, which indicates whether

the contents are binary data or text, and, in the latter case, whether the text is believed to be UTF-8 formatted. The packet also contains a filename, which is helpful when transferring a file, but is mostly ignored by GnuPG in practice. And, it contains a timestamp. GnuPG sets the timestamp to the current time when the packet is created (not the file's `mtime`).

It is worth pointing out that when GnuPG is told to decrypt data (`gpg --decrypt`), it doesn't look for an encrypted message to decrypt, but processes the message and tries to decrypt any encypted data that it encounters. This subtle difference in behavior can be important, because if GnuPG is told to decrypt a message with just a literal packet, it will simply output the contents of the literal packet without warning the user that the data was not actually encrypted. If a program uses the ability to decrypt a message as an authentication check (e.g., in AutoCrypt's Setup Message), this behavior could lead to subtle attacks [5].

1.5 Signing

A signature provides cryptographic proof of both the signed data's integrity and its authenticity—assuming the key used to sign the data is trusted. That is, like a checksum, a signature can be used to make sure that the data was not modified in transit. But unlike a checksum, a signature can also provide proof of the data's origin (or at least, who signed off on the message).

Note: the exact semantics of a signature are not defined by the standard. This is done on purpose, and is viewed by the RFC editors as a feature, because, in the end, a signature's meaning is determined by the actual human users of the system—some will be more casual, and some will be more rigorous no matter what some standard says.

1.5.1 Multiple Signers

In OpenPGP, it is possible for a single message to include multiple signatures created by different keys. This mechanism is useful when disparate parties want to sign a document. For instance, multiple developers might sign released software. Rather than providing each signature separately, it is more useful to combine them into a single file.

In GnuPG, this can be done by specifying each of the keys on the command line. For instance:

```
$ echo 'Good-bye cruel world!' | gpg -s -u romeo -u juliet
```

A crippling disadvantage of this approach is that all keys must be available at the time that the signature is generated, which is rarely practical.

Although OpenPGP's packetized message format makes combining signatures relatively easy, GnuPG does not provide support for this. Nevertheless, in practice, writing an ad-hoc script is straightforward (some hints are here: [6]). And, in the special case that the signatures in question are *detached* signatures, combining them is actually trivial: they just need to be concatenated together as shown below:

```
$ echo 'Romeo and Juliet forever!' > note.txt
$ gpg --detach-sign -u romeo --output - note.txt > note.txt.romeo.sig
$ gpg --detach-sign -u juliet --output - note.txt > note.txt.juliet.sig
$ cat note.txt.romeo.sig note.txt.juliet.sig > note.txt.sig
$ gpg --verify note.txt.sig note.txt
gpg: Signature made Tue 11 Jul 2017 11:52:48 AM CEST
gpg:                using RSA key D6636A9EB82A91E94DDEE5066B284A5BE2297415
gpg:                issuer "romeo.montague@gnupg.net"
gpg: Good signature from "Romeo Montague <romeo.montague@gnupg.net>" [full]
gpg: Signature made Tue 11 Jul 2017 11:52:59 AM CEST
gpg:                using RSA key E5156E507DCB8D63AC89E5334954FDC67A46B4C5
gpg:                issuer "juliet.capulet@gnupg.net"
gpg: Good signature from "Juliet Capulet <juliet.capulet@gnupg.net>" [full]
```

In the above examples, the signatures are not nested. That is, they are both only over the data, and one could remove either signature from the OpenPGP message without impacting the validity of the other signature.

Sometimes, it can be useful to nest signatures. For instance, a notary might want to not only notarize some document, but also the client's signature over that document. OpenPGP also provides native support for this type of signature. In fact, both types can be present in the same message. GnuPG does not currently support nested signatures.

1.5.2 Algorithm

As in the encryption case, signing is a two-step process. First, the data to be signed is hashed, and then the resulting hash is signed using public-key cryptography. This two-step process is primarily motivated by performance considerations.

The exact algorithm that is used is slightly different depending on whether the signature should be inline or detached. We start by describing how an inline signature is created.

1. Emit a so-called *One-Pass Signature* (OPS) packet. An OPS packet contains meta-data (what hash algorithm to use, etc.) as well as framing information (specifically, whether the signature is nested or not).
2. Hash and emit the data to sign.
3. Emit a signature packet, which includes the computed hash and the signature.

As its name and the implementation suggest, the OPS packet makes it possible to both create a signature, and verify it without buffering any data. Since detached signatures are separate from the main OpenPGP message, and OPS packets are effectively redundant, to generate a detached signature, we just skip the first step. A limitation of detached signatures is that they are over the entire OpenPGP message. Thus, nesting them is not possible.

1.5.3 Example

Using our above example with inline signatures, the resulting message has the following packets:

```
$ echo 'Good-bye cruel world!' \
> | gpg -s -u romeo -u juliet | gpg --list-packets
# off=0 ctb=a3 tag=8 hlen=1 plen=0 indeterminate
:compressed packet: algo=1
# off=2 ctb=90 tag=4 hlen=2 plen=13
:onepass_sig packet: keyid 4954FDC67A46B4C5
version 3, sigclass 0x00, digest 8, pubkey 1, last=0
# off=17 ctb=90 tag=4 hlen=2 plen=13
:onepass_sig packet: keyid 6B284A5BE2297415
version 3, sigclass 0x00, digest 8, pubkey 1, last=1
# off=32 ctb=cb tag=11 hlen=2 plen=28 new-ctb
:literal data packet:
mode b (62), created 1499772743, name="",
raw data: 22 bytes
```

```
# off=62 ctb=89 tag=2 hlen=3 plen=333
:signature packet: algo 1, keyid 6B284A5BE2297415
version 4, created 1499772743, md5len 0, sigclass 0x00
digest algo 8, begin of digest 88 56
hashed subpkt 33 len 21 (issuer fpr v4 D6636A9EB82A91E94DDEE5066B284A5BE229741
hashed subpkt 2 len 4 (sig created 2017-07-11)
hashed subpkt 28 len 24 (signer's user ID)
subpkt 16 len 8 (issuer key ID 6B284A5BE2297415)
data: [2048 bits]
# off=398 ctb=89 tag=2 hlen=3 plen=333
:signature packet: algo 1, keyid 4954FDC67A46B4C5
version 4, created 1499772743, md5len 0, sigclass 0x00
digest algo 8, begin of digest c5 e3
hashed subpkt 33 len 21 (issuer fpr v4 E5156E507DCB8D63AC89E5334954FDC67A46B4C
hashed subpkt 2 len 4 (sig created 2017-07-11)
hashed subpkt 28 len 24 (signer's user ID)
subpkt 16 len 8 (issuer key ID 4954FDC67A46B4C5)
data: [2047 bits]
```

Compressed Packet

Again, we see that the message starts with a compression container. Since the length of the data is not known apriori, the length is marked as `indeterminate`, which means that the packet includes all of the data until the end of the message.

One-Pass Signature Packets

The next two packets are OPS packets.

These packets include the hash algorithm that was used to generate the signature. This information needs to be available beforehand so that the signature can be verified in a streaming fashion. The hash algorithm, which is also known as the message digest algorithm, is indicated by the `digest` field in the output.

Another piece of information that is necessary to verify the data in a streaming manner is how to interpret the data to sign. This is determined by the signature's class (`sigclass`). Normally, OPS packets are only used with documents (as opposed to keys or user IDs, which are so small that

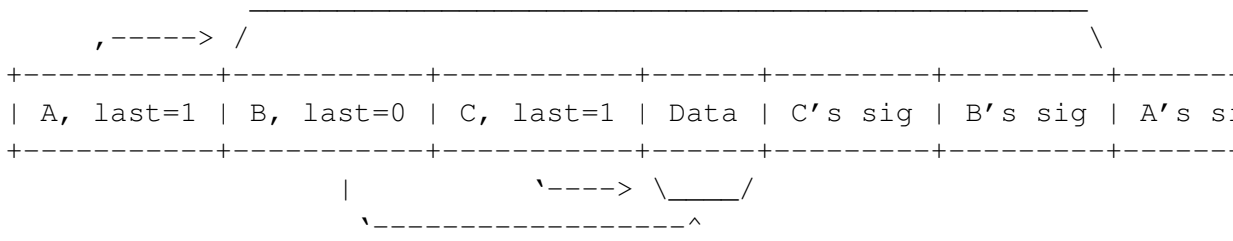
buffering isn't an issue). OpenPGP defines two types of documents: binary data and text data whose respective classes are `0` and `1`. For binary documents, the data is hashed as is; for text documents, the OpenPGP implementation first converts line endings to `<CR><LF>` before hashing.

The OPS packets also include the signer's key ID and the public key algorithm used to generate the signature. This information is strictly speaking redundant as it is also stored in the matching signature packet, but it can help the implementation identify several common cases in which it can't verify the signature prior to actually computing the hash. Specifically, the implementation can't verify a signature if the signer's public key is unavailable, or the public key algorithm used to compute the signature is not supported (even if the hash algorithm is supported). In such cases, the implementation can fail early, or just skip the hashing, which saves some CPU cycles.

Finally, OPS packets include framing information. In GnuPG, this is referred to as the *last signature* flag. In the above output, it is referred to `last`. If `last` is 1, then the signature is over all of the following data up to the OPS's corresponding signature packet; if `last` is 0, then the signature is not nested and is only over the data following the next OPS packet with `last` equal to `1`.

Given this definition of `last`, we see that the first signature in the above example is not nested (`last` is `0`), but the second is. Thus, both signatures are over the data; the outer signature is *not* over the inner signature, just the data.

To better understand how signatures nest, consider the following example, which shows an OpenPGP message with three signatures. The first three packets are OPS packets, the middle packet is a literal data packet, and the last three packets are the OPS' corresponding signature packets.



Working our way in, we see that `last` is set for A's signature. Thus, A's signature is over everything immediately following the OPS packet up

to the matching signature packet. That is, it is over not only the data, but also over B and C's signatures. In contrast, in B's OPS packet, `last` is clear. Thus, B's signature is over everything following the next OPS packet with `last` set to `1`, i.e., everything follow C's OPS packet, up to, but not including, the signature packet matching C's OPS packet. That is, like C's signature, B's signature is only over the literal data packet, not the data packet *and* C's signature.

Literal Data

The literal data packet contains the document to be signed. Of course, if the signatures are nested, then the signature may include other data as well.

Signature Packet

The last two packets are the signature packets that match the OPS packets at the start of the message. Like braces in a programming language, the first OPS packet matches the last signature packet, and the second OPS packet matches the second to last signature packet.

Except for the nesting information, the signature packet includes everything present in the OPS packet as well as some additional meta-data, and the actual signature.

The additional meta-data usually includes a timestamp (the OpenPGP Signature Creation Time subpacket), and the user ID that was used to make the signature (the OpenPGP Issuer subpacket). There are several other pieces of metadata that can be added, but they are not usually set in this context.

The issuer is usually used by a mail user agent to make sure the alleged sender matches the signer. For instance, Romeo might have verified his father's key, but his father might try to trick him by sending him an email that appears to be from Juliet. Because he knows that Romeo always checks a signature's validity, he could just sign the message with his own key. If the mail user agent only shows whether a signature is valid, then Romeo might be tricked. Making sure the from header matches the issuer catches this attack.

1.6 Keys

As mentioned above, OpenPGP messages are not only used to transport documents, but are also used to transport keys and key signatures.

In OpenPGP, a so-call *key* is a lot more than just a public and private key pair. Modern OpenPGP keys normally include at least two key pairs as well as a fair amount of meta-data.

1.6.1 Multiple Public and Private Key Pairs

OpenPGP supports multiple key pairs for several reasons.

First, although it is possible to use the same key pair for encryption and signing, if you do, then the act of decrypting a message is equivalent to signing it (and vice versa), which could be abused by an adversary. In practice, this particular attack is prevented by the use of distinguishing padding schemes. But, using separate keys avoids this problem and prevents any issues that may be discovered in the future.

Second, having multiple keys makes it possible to largely separate identity from key lifetime. In particular, OpenPGP has the concept of primary keys and subkeys. The primary key is used to identify the OpenPGP key. That is, a key's fingerprint is derived from this key, and is independent of any subkeys. This makes it possible for a user to revoke individual subkeys without changing her identity. For instance, each year you could generate a new encryption and a new signing subkey, and revoke the old ones, and there would be no need to create new business cards or even inform your contacts that you have new keys, because, assuming their software is configured to regularly refresh your key, their OpenPGP implementation will automatically find the new subkeys since your primary key did not change. In fact, this type of key rotation approximates forward secrecy [7].

To support an arbitrary number of keys, primary keys and subkeys are marked with so-called *capabilities*. There are (perhaps surprisingly) four capabilities:

1. Encryption
2. Signing
3. Certification
4. Authorization

An encryption capable key can be used for encryption, and a signing capable key can be used for signing documents. But, if a key does not have the encryption capability, then it should not be used for encryption. The certification capability indicates that a key can be used for signing *keys* (as opposed to documents). Thus, since a subkey requires a signature to be valid, only a certification-capable key can be used to create a new subkey. Finally, the authorization capability is used for access control. This is primarily useful for using an OpenPGP key with `ssh`.

It is entirely possible for a key to have multiple capabilities. As mentioned above, it is not advisable to use a key for both signing and encryption, but since mathematically certification is just signing, it is reasonable to mark a key as both signing and certification capable.

Whether this is reasonable depends on how the user wants to manage keys. For instance, if a signing-capable key is compromised, it is possible to recover without generating an entirely new OpenPGP key. But, if a certification-capable key is compromised, then the attacker effectively owns the identity, and the only way to recover is to completely revoke the OpenPGP key and create a new one. This only works if users physically separate the certification key from the signing key, e.g., by only storing the certification key on an offline computer. Since most users don't do this, GnuPG defaults to making the primary key both certification capable and signing capable.

An OpenPGP key can have multiple valid (i.e., not expired and not revoked) subkeys with the same capability. In this case, the RFC does not specify which subkey should be used; it is up to the implementation.

If there are multiple encryption-capable keys, GnuPG uses the newest valid subkey. But this is not the *de facto* standard. For instance, OpenKeychain encrypts a message to all valid encryption-capable keys.

The OpenKeychain behavior has the advantage that one can store different keys on different devices. Then if a particular device is compromised, only the subkeys on that device need to be rotated. But, operationally, the advantages for encryption-capable subkeys are not that large, since an encryption-capable key protects *past* traffic. That is, if an encryption key is compromised, all messages encrypted to it are compromised. Thus, a message is compromised if any encryption key is compromised. So, in this case, one might as well just use a single encryption key.

This line of logic does not apply to signing-capable keys. If a signing-capable subkey is compromised, the attacker can forge messages. But, if

the user has one signing-capable key per device and revokes just the single signing-capable subkey that was compromised, then the attacker will be thwarted and only signatures created using that key will fail to verify after it has been revoked.

1.6.2 Self Signatures

As mentioned previously, an OpenPGP fingerprint is derived only from the primary key, not the subkeys. This makes sense, since new subkeys can be added at any time. Thus, some mechanism is needed to associate subkeys with the corresponding primary key. Further, a mechanism is needed to associate meta-data with an OpenPGP key. Both of these problems are solved using the same mechanism: self-signatures.

A self-signature is like a normal signature, but instead of being over a document, the signature is over structured text, and it is stored alongside the OpenPGP key. A self-signature can only be created (or rather, is only honored if it was created) by a certification-capable key. Since the signature can't be forged, it effectively creates an unforgable binding between the OpenPGP key and the data. Thus, to determine if a subkey really belongs to a given OpenPGP key, it is sufficient to check whether there is a valid self-signature.

Because OpenPGP packets can be combined in whatever way a user wants, an attacker who controls a user's network connection may not be able to modify individual packets without detection, but can drop packets. Thus, if an attacker has compromised a user's key, the user notices, and revokes her key, she is still not safe if the attacker also controls the network path, and filters out the revocation certificate thereby preventing other users from learning that the key was compromised.

1.6.3 Example

The following example shows Romeo's key. This key was created by GnuPG using the default parameters. Thus, it has a primary key, which is signing- and certification-capable, and a single subkey, which is encryption capable.

```
$ gpg --export romeo | gpg --list-packets
# off=0 ctb=99 tag=6 hlen=3 plen=269
:public key packet:
```

```
version 4, algo 1, created 1499443140, expires 0
pkey[0]: [2048 bits]
pkey[1]: [17 bits]
keyid: 6B284A5BE2297415
# off=272 ctb=b4 tag=13 hlen=2 plen=41
:user ID packet: "Romeo Montague <romeo.montague@gnupg.net>"
# off=315 ctb=89 tag=2 hlen=3 plen=340
:signature packet: algo 1, keyid 6B284A5BE2297415
version 4, created 1499443140, md5len 0, sigclass 0x13
digest algo 8, begin of digest 71 f6
hashed subpkt 33 len 21 (issuer fpr v4 D6636A9EB82A91E94DDEE5066B284A5BE229741
hashed subpkt 2 len 4 (sig created 2017-07-07)
hashed subpkt 27 len 1 (key flags: 03)
hashed subpkt 9 len 4 (key expires after 2y0d0h0m)
hashed subpkt 11 len 4 (pref-sym-algos: 9 8 7 2)
hashed subpkt 21 len 5 (pref-hash-algos: 8 9 10 11 2)
hashed subpkt 22 len 3 (pref-zip-algos: 2 3 1)
hashed subpkt 30 len 1 (features: 01)
hashed subpkt 23 len 1 (keyserver preferences: 80)
subpkt 16 len 8 (issuer key ID 6B284A5BE2297415)
data: [2048 bits]
# off=658 ctb=b9 tag=14 hlen=3 plen=269
:public sub key packet:
version 4, algo 1, created 1499443140, expires 0
pkey[0]: [2048 bits]
pkey[1]: [17 bits]
keyid: 5B905AF0423ABB52
# off=930 ctb=89 tag=2 hlen=3 plen=310
:signature packet: algo 1, keyid 6B284A5BE2297415
version 4, created 1499443140, md5len 0, sigclass 0x18
digest algo 8, begin of digest 19 f8
hashed subpkt 33 len 21 (issuer fpr v4 D6636A9EB82A91E94DDEE5066B284A5BE229741
hashed subpkt 2 len 4 (sig created 2017-07-07)
hashed subpkt 27 len 1 (key flags: 0C)
subpkt 16 len 8 (issuer key ID 6B284A5BE2297415)
data: [2043 bits]
```

Public Key Packet

The public key packet normally comes first. It just contains a minimum amount of information: the public key algorithm (`algo`), the public key parameters (`pkey`), the creation time (`created`), and the expiry time (`expires`). Although the `--list-packets` output shows the key ID, this is not included in the packet; it is shown as a matter of convenience. Including it in the packet would be redundant, because it is derived from the creation time and the public key parameters.

In the above listing, there is no self-signature for the public-key packet. The parameters are, however, protected by the self-signature over each user ID packet, which is over not only the user ID packet, but also the primary key. It is possible to make signatures just over the primary key. But, this is typically only used in the case of key revocation.

Not using a self-signature for the key means that meta-data like user preferences needs to be stored someplace else. By convention, they are stored in a user ID's self-signature. Consequently, if you have multiple user IDs, you could have multiple sets of conflicting preferences. This is actually by design: the relevant preferences are determined by how the key is addressed, which allows different sets of preferences for different environments. So, if you have two user IDs, one for work, and one for home, when someone uses your key to encrypt to your work email address, the preferences are taken from the work user ID. If the caller just specifies the key ID, then the preferences are taken from the so-called *primary user ID*. (The primary user ID is the user ID with the primary user ID flag set in its self-signature. If there are no user IDs that have this flag set or multiple user IDs, then RFC 4880 recommends using the user ID with the newest self-signature.) Thus, because it is reasonable to have different preferences for different user IDs, if the intended user ID is known, it—and not the key ID—should be used to address the key.

By convention, self-signatures immediately follow the packet that they certify. As such, any direct key signatures would immediately follow the public key prior to any user ID or subkey packets. In practice, this is not always the case due to implementation bugs or malicious intent. Thus, on import, GnuPG will attempt to fix any out-of-order packets. This can involve some overhead, but this additional overhead is only incurred if the packets are actually out of order.

When some meta-data is changed, a new self-signature is created. Since

data that is publish can't easily be deleted, OpenPGP treats the key as an append-only log. The result is that a user ID packet, for instance, might have multiple self signatures.

In general, if there are multiple self-signed packets for a given packet, only the newest one is used. One important exception is for revocation certificates and any designated revoker settings: it is necessary to respect these even if a later self signature would somehow override them, because this capability could be used by an attacker to invalidate a revocation, which would effectively make revocations of compromised keys impossible.

User ID Packet

User IDs are stored between the public key and any subkeys. In this example, the key only contains a single user ID.

A user ID packet just contains a single value: a free-form string. By convention (per the RFC), this string is an RFC 2822-style mailbox, i.e., a UTF-8 encoded string of the form `Name <email@example.com> (Comment)`.

Normally, a user ID doesn't require a comment, and, like Romeo's key, most keys don't have one. Nevertheless, even though comments can (rarely!) be useful for advanced users, it is recommended that most tools not offer users the option to set it, because most people don't understand what they are for.

There are two main uses for comments: to distinguish security levels and roles. Thus, if a user wants to have two OpenPGP keys associated with a given email address, one for low-security communication, which is stored directly on the device thereby allowing immediate decryption, and one for high security communication, which is, say, stored on an air-gapped computer and therefore may introduce a long delay if the user is not near the air-gapped computer, comments along the lines of "day-to-day key" and "high security key," respectively, might be appropriate. Similarly, if a developer has a key that is only used for signing commits and releases, a reasonable comment on that key could be "dist sig". Daniel Kahn Gillmor takes an even more conservative stance, and argues that even these comments are probably unnecessary [8].

It is also possible to use an image as a user ID. In such cases, the image is stored in a so-called user attribute packet. One problem with images is that they can be fairly large. Since images like old signatures can't be deleted once they are published, and they are downloaded whenever a key

is retrieved, it is currently recommended that images be limited to just a few kilobytes of data.

Images can be useful since many people are able to more quickly associate a person with that person's likeness than with her name. Thus, an image could be shown in a Jabber client or a mail user agent. However, this should probably only be done for validated keys to avoid suggesting authenticity when there is no evidence thereof. Another possible use for images is in a graphical depiction of a path in the web of trust.

User ID Self Signature

By convention, the user ID self-signature immediately follows the user ID. In addition to binding the user ID to the primary key, it also contains additional metadata. As noted above, there may be multiple self-signatures, and normally only the newest is used.

The signature is self-describing. It includes the key that was used to create the signature, the algorithm, etc. The `sigclass` subpacket is `0x13`, which means that this signature is over a user ID.

The signature includes a number of hashed subpackets. Hashed subpackets are effectively key-value pairs that are validated by the signature. The OpenPGP specification includes 22 different subpackets including so-called *notation data*, which can be used to store arbitrary data. (Notations are describing towards the end of this chapter.)

In this example, there are 10 subpackets. Some of the subpackets provide information about the signature itself. This is the case for the `issuer fpr`, `sig created` and `issuer key ID` subpackets. Some of them provide information about the primary key. This is the case for the `key flags`, and `key expires after` subpackets. The `key flags` subpacket is primarily used for indicating the primary key's capabilities. The `key expires after` subpacket indicates when the key expires. An expiration can be extended by creating a new self-signature with a later expiration time. Note: the expiration time is relative to the key's—not the self-signature's—creation time. And, the remaining subpackets describe user and implementation preferences. `pref-sym-algos`, `pref-hash-algos`, and `pref-zip-algos` specify what symmetric, hash and compression algorithms, respectively, the user's OpenPGP implementation supports, and the user wants when using this user ID. `features` describes what advanced features the OpenPGP implementation supports. Currently, there

is only one flag defined, which indicates that the OpenPGP implementation supports the MDC system. And, `keyserver preferences` is a set of flags indicating how the key server should handle the key.

With the exception of the `issuer key ID`, all of the subpackets are prefixed with `hashed`. This indicates that this data is part of the signed data. Subpackets that are not hashed are considered advisory, because an attacker may modify them without detection in transit.

There is also a Preferred Key Server subpacket. But, to avoid leaking metadata, GnuPG ignores this option by default.

Public Subkey Packet

The public subkey packets follow the user ID packets. Other than their type, these packets are effectively identical to the public key packet.

Public Subkey Self Signature

Like user ID packets, a public subkey packet requires a self-signature to validate the key and bind it to the primary key. Typically, a subkey packet contains just a few pieces of meta-data, because preferences are stored in user ID self signatures.

There are two minor differences, which are worth pointing out. First, whereas the `sigclass` field for user ID is `0x13`, the `sigclass` for public subkeys is `0x18`. Second, if the subkey is signing capable, then the self-signature must also have a so-called *back signature* in an embedded signature subpacket created by the signing key over the primary key and the subkey. Obviously, this back signature should not be created for an encryption key based on the aforementioned attacks.

1.7 Key Signing

OpenPGP allows users to validate each other's keys using signatures. Thus, if Romeo is convinced that Juliet controls the key `0x4954FDC67A46B4C5`, then he could certify it (i.e., sign it) using his OpenPGP key. There are two main reasons why Romeo would want to certify someone's key.

First, a certification mechanism of this sort enables the OpenPGP implementation to determine whether a key is valid. This information is critical when Romeo wants to verify a signed document. In that case, Romeo is

not just interested in whether the signature is mathematically valid, and the data has not be corrupted in transit, but he also wants to know whether the signature was really created by Juliet. Unfortunately, there is no way for computers to figure this out without some help from users. Likewise, when Romeo sends an email to Juliet, he wants to be confident that he is really using Juliet's key. It is completely possible that Romeo could have a key that allegedly belongs to Juliet without realizing it (anyone can create a key with any user ID, and upload it to the key servers).

The other reason that a signature is useful is that it provides a mechanism for Romeo's contacts to indirectly verify Juliet's key. That is, when Romeo shares this signature with others (e.g., by publishing it on a key server), then people who trust him (and this is essential!) to validate other people's keys, i.e., to be a so-called *trusted introducer*, could use this signature to find a valid key for Juliet. The network induced on the signatures is referred to as the web of trust although it would be more accurate to refer to is as the web of verifications.

Unfortunately, publishing signatures has the unfortunate side-effect of making the user's social graph public. This can have grave implications beyond the privacy concerns. For instance, it could be used to link a source to a journalist.

1.7.1 Local Signatures

If a signature shouldn't be published, it is possible to mark it as being unexportable. To do this, one would create a local signature. This is done in GnuPG by using `--lsign-key` instead of `--sign-key` to sign the key. At a technical level, this causes an `Exportable Certification` subpacket to be included in the signature with the value of `0`.

Unfortunately, using local signatures is not without problems: it is possible to export local signatures and accidentally upload them to a key server, and the key server implementations do not automatically strip local signatures on import.

1.7.2 Confidence

When someone verifies a key, she doesn't always have the same degree of confidence that the verification is correct. For instance, when Romeo signs Juliet's key, he is almost certainly convinced that Juliet really controls the

stated key. On the other hand, if Romeo is at the pub and meets Iago, and he asks him to sign his key, Romeo is almost certainly less confident that Iago controls the stated key. This is the case even if Iago shows him his government issued identification papers. And, it is also the case if he sends an encrypted email to the email address in Iago's user ID, and receives a signed reply with a shared secret code.

OpenPGP provides a mechanism for expressing different degrees of confidence in the form of three confidence levels ranging from "the person said she controls the key" to "I'm confident she controls the stated key" as well as a generic, "no comment," level. Other than completely ignoring the weakest certification level, this information is not included in web of trust calculations by GnuPG. Thus, for all intents and purposes, it is just gratuitous meta-data. As such, it is better to always use a generic certification level [9]. This is what GnuPG does by default.

1.7.3 Trusted Introducers

When signing a key, it is possible to indicate that the key holder should be a trusted introducer. For instance, an organization may have a single key, say `pgp@company.com`, that they use to sign all of their employees' keys. If employees sign `pgp@company.com` using a trust signature, then anyone who trusts, say, `alice@company.com`, will, as usual, consider `pgp@company.com` to be not only verified, but, due to the trust signature, a trusted introducer. Consequently, that person will also consider any keys that `pgp@company.com` signed to be verified, which, in this case, is everyone in the company. The following example illustrates this idea:

```
juliet@                alice@                 pgp@                   bob
example -- tsign --> company -- tsign --> company -- sign --> @company
.org                   .com                   .com                   .com
```

In GnuPG, Juliet doesn't actually have to use a trust signature to sign `alice@company.com`'s key: she can just use a normal signature and then set the `ownertrust` for `alice@company.com` appropriately.

Trust signatures are very powerful and can also be very dangerous. If Romeo considers Juliet to be a trusted introducer, and Juliet has `tsign` ed her father's key, then any key that Juliet's father signs will be considered verified. Juliet's father could abuse this fact to trick Romeo into trusting a key that he forged for Juliet.

Trust signatures can be constrained. For instance, in the above example, Alice probably wants to limit the scope of her trust signature of `pgp@company.com`'s key to just those user IDs associated with `company.com`. To support this, OpenPGP allows a regular expression to be associated with a trust signature.

A trust signature can also make not just immediate connections trusted, but also indirect connections. This is extremely dangerous and probably only makes sense in very limited situations. For instance, in a very large company, each department might have the equivalent of the above `pgp@company.com` key, and there is a company-wide key that `tsign`s each department's key. In this case, Alice might sign the company-wide key with a depth of `2` instead of `1`. (When Alice uses a trust level of `1`, she means that anyone that the company verifies is considered verified. A trust level of `0` is equivalent to a normal signature; it doesn't create any trusted introducers.)

In GnuPG, it is currently not easy to modify a signature. For instance if you want to convert a normal signature into a trust signature, `gpg` will complain that the key is already signed. To change a signature type or modify a trust signature, it is first necessary to revoke the existing signature using the `revsig` command in the `--edit-key` interface.

1.7.4 Non-Revocable Signatures

Occasionally, it can be useful to make a long-term commitment to a signature. This can be done by setting the non-revocable flag. In GnuPG, this is done using the `nrsign` command in the `--edit-key` interface.

1.7.5 Example

The following example shows Juliet's key including Romeo's signature of her key.

```
$ gpg --export juliet | gpg --list-packets
# off=0 ctb=99 tag=6 hlen=3 plen=269
:public key packet:
version 4, algo 1, created 1499443081, expires 0
pkey[0]: [2048 bits]
pkey[1]: [17 bits]
keyid: 4954FDC67A46B4C5
```

```
# off=272 ctb=b4 tag=13 hlen=2 plen=41
:user ID packet: "Juliet Capulet <juliet.capulet@gnupg.net>"
# off=315 ctb=89 tag=2 hlen=3 plen=340
:signature packet: algo 1, keyid 4954FDC67A46B4C5
version 4, created 1499443081, md5len 0, sigclass 0x13
digest algo 8, begin of digest 59 1a
hashed subpkt 33 len 21 (issuer fpr v4 E5156E507DCB8D63AC89E5334954FDC67A46B4C
hashed subpkt 2 len 4 (sig created 2017-07-07)
hashed subpkt 27 len 1 (key flags: 03)
hashed subpkt 9 len 4 (key expires after 2y0d0h0m)
hashed subpkt 11 len 4 (pref-sym-algos: 9 8 7 2)
hashed subpkt 21 len 5 (pref-hash-algos: 8 9 10 11 2)
hashed subpkt 22 len 3 (pref-zip-algos: 2 3 1)
hashed subpkt 30 len 1 (features: 01)
hashed subpkt 23 len 1 (keyserver preferences: 80)
subpkt 16 len 8 (issuer key ID 4954FDC67A46B4C5)
data: [2047 bits]
# off=658 ctb=89 tag=2 hlen=3 plen=307
:signature packet: algo 1, keyid 6B284A5BE2297415
version 4, created 1499445515, md5len 0, sigclass 0x10
digest algo 8, begin of digest c6 a3
hashed subpkt 33 len 21 (issuer fpr v4 D6636A9EB82A91E94DDEE5066B284A5BE229741
hashed subpkt 2 len 4 (sig created 2017-07-07)
subpkt 16 len 8 (issuer key ID 6B284A5BE2297415)
data: [2046 bits]
# off=968 ctb=b9 tag=14 hlen=3 plen=269
:public sub key packet:
version 4, algo 1, created 1499443081, expires 0
pkey[0]: [2048 bits]
pkey[1]: [17 bits]
keyid: C1A010A1D38C4BB8
# off=1240 ctb=89 tag=2 hlen=3 plen=310
:signature packet: algo 1, keyid 4954FDC67A46B4C5
version 4, created 1499443081, md5len 0, sigclass 0x18
digest algo 8, begin of digest ee 3f
hashed subpkt 33 len 21 (issuer fpr v4 E5156E507DCB8D63AC89E5334954FDC67A46B4C
hashed subpkt 2 len 4 (sig created 2017-07-07)
hashed subpkt 27 len 1 (key flags: 0C)
```

```
subpkt 16 len 8 (issuer key ID 4954FDC67A46B4C5)
data: [2047 bits]
```

The listing follows the usual format described above. The first packet is the public key packet, which is followed by a user ID packet and its self signature. And, at the end comes the subkey key and its self signature.

There is one small difference, however. In this listing, Juliet's user ID is followed by not one, but two signatures. And, the second one is not a self-signature, but Romeo's certification signature: we can see from the `issuer fpr` subpacket that Romeo, not Juliet, created this signature. There are two important things to observe here.

First, Romeo's signature is associated with Juliet's key, not his key. Once it is clear that the signature says something about Juliet's key and not Romeo's, this makes sense. Nevertheless, many beginners don't understand this and think that they somehow own the signature. Unfortunately, this arrangement can lead to denial of service attacks. For instance, vandals could create many signatures on a particular key so that it becomes so large that it can't be imported.

Second, certification signatures are associated with user IDs and not with keys. This avoids bait-and-switch type attacks. Consider Paris who convinces Romeo to sign his key. If Romeo signed the key, and not the user ID, then Paris could simply revoke the user ID and replace it with another, say, Juliet's. Since Romeo would still consider the key to be valid, Paris could possibly trick him into believing a message from the key is from Juliet.

1.8 Revocations

If a key has been compromised or simply retired, it is essential to revoke it so that other people don't accidentally use it. It is also important to revoke a user ID if the identity is no longer valid, e.g., when leaving an organization, but keeping the same key. Occasionally, it can be useful to revoke a user ID certification. For instance, you should revoke a certification if: you find out that you signed the wrong key; the person who controlled the key somehow lost control of it (e.g., he forgot the password, and doesn't have a revocation certificate); or, you find out that you signed an impostor's key.

The following example shows what Juliet's key looks like when she revokes her own key (the output has been truncated):

```
$ gpg --gen-revoke juliet | gpg --import
...
$ gpg --export juliet | gpg --list-packets
# off=0 ctb=99 tag=6 hlen=3 plen=269
:public key packet:
version 4, algo 1, created 1499443081, expires 0
pkey[0]: [2048 bits]
pkey[1]: [17 bits]
keyid: 4954FDC67A46B4C5
# off=272 ctb=89 tag=2 hlen=3 plen=310
:signature packet: algo 1, keyid 4954FDC67A46B4C5
version 4, created 1500052199, md5len 0, sigclass 0x20
digest algo 8, begin of digest 04 ca
hashed subpkt 33 len 21 (issuer fpr v4 E5156E507DCB8D63AC89E5334954FDC67A46B4C
hashed subpkt 2 len 4 (sig created 2017-07-14)
hashed subpkt 29 len 1 (revocation reason 0x02 ())
subpkt 16 len 8 (issuer key ID 4954FDC67A46B4C5)
data: [2048 bits]
# off=585 ctb=b4 tag=13 hlen=2 plen=41
:user ID packet: "Juliet Capulet <juliet.capulet@gnupg.net>"
...
```

The revocation is the second packet. It is a self signature on the primary key. We know that the packet is a revocation certificate based on the `sigclass` (`0x20`) as well as the `revocation reason` subpacket. The `revocation reason` allows the user to say why the key is revoked. Here, the value is `0x2`, which means that the key was compromised. This subpacket can also include a human-readable string. In this case, Juliet did not provide any additional information. But, in the case that the key is being rotated, it might be helpful to include the new key's fingerprint. Of course, this is of limited use, since it is not machine readable.

1.9 Notations

RFC 4880 allows signatures to contain arbitrary data. This mechanism can be extremely useful for extending the OpenPGP system. But, despite its availability, they aren't generally used. One example of how they could be used was considered by the Debian project, which thought about using

notations to store additional information about how a developer's identity was checked [10].

Notations are key value pairs. The key must be of the form `key@example.com`. The domain is included to avoid naming conflicts. Although the value can be any arbitrary data, GnuPG currently only supports free-form strings.

One limitations of notations is that as they are stored in signature subpackets, they must fit into the 64 kilobytes of space available to signature subpackets. (Strictly speaking, the hashed area is limited to 64 kilobytes of subpackets and the unhashed area has the same limitation, but using the unhashed area is not advisable.)

1.10 Summary

This chapter has presented the important details of the OpenPGP standard. This introduction wasn't intended for someone who is planning to write an OpenPGP parser, but to provide a rough overview of the system. Many details have been omitted, as well as several minor features (yes, for better or worse, OpenPGP is that feature rich). For those looking for more information, the RFC is probably the best place to start: it is highly readable, and this introduction should hopefully make it easy to navigate.

Bibliography

[1] OpenPGP Working Group. Charter for working group. https://datatracker.ietf.org/wg/openpgp/charter/.

[2] Peter Gutmann. [openpgp] expiration impending: <draft-ietf-openpgp-rfc4880bis-01.txt>. https://www.ietf.org/mail-archive/web/openpgp/current/msg08863.html, July 2017.

[3] Serge Mister and Robert Zuccherato. An attack on CFB mode encryption as used by OpenPGP. 3897:82–94, 2005.

[4] Jon Callas. Re: A review of hash function brittleness in OpenPGP. https://www.ietf.org/mail-archive/web/openpgp/current/msg00468.html, January 2009.

[5] Holger P. Krekel, Danial Kahn Gillmor, and et al. Autocrypt level 1: Enabling encryption, avoiding annoyances - bad import. https://autocrypt.readthedocs.io/en/latest/bad-import.html.

[6] Werner Koch. Clearsign text document with multiple keys? https://lists.gnupg.org/pipermail/gnupg-users/2013-July/047118.html, July 2013.

[7] Ian Brown, Adam Back, and Ben Laurie. Forward Secrecy Extensions for OpenPGP. Internet-Draft draft-brown-pgp-pfs-03, IETF Secretariat, October 2001. https://tools.ietf.org/html/draft-brown-pgp-pfs-03.

[8] Daniel Kahn Gillmor. Openpgp user id comments considered harmful. https://debian-administration.org/users/dkg/weblog/97, May 2013.

[9] Daniel Kahn Gillmor. gpg –ask-cert-level considered harmful. https://debian-administration.org/users/dkg/weblog/98, May 2013.

[10] Daniel Kahn Gillmor. using openpgp notations to indicate keysigning practices. https://lists.debian.org/debian-devel/2009/06/msg00722.html, June 2009.