An Advanced Introduction to GnuPG

Neal H. Walfield

August 18, 2017

Contents

Part I

Main Matter

Chapter 1

Introduction

GnuPG is an implementation of the OpenPGP protocol, which is used for encryption and authentication.

GnuPG is used to encrypt email. But, this functionality is not just used by individuals to preserve their privacy: political activists rely on it to organize their activities, journalists rely on it to protect their sources, and lawyers rely on it to protect attorney-client conversations. Jason Reich, the director of security for BuzzFeed, describes the importance of GnuPG to journalists this way: "GPG is part of a balanced breakfast of any reporter, especially one who wants to protect their sources, and be able to be reached for leaks and things of that nature." [1]. And, Michal Wozniak, the Chief Information Security Officer at the Organized Crime and Corruption Reporting Project (OCCRP), said, "I do strongly believe that had we not been using GnuPG all of this time, many of our sources and many of our journalists, would be in danger or in jail" [2]. Cindy Cohn, the Executive Director of the Electronic Frontier Foundation, goes further, and says that the privacy and security that GnuPG offers makes it "one of the core tools that we need if we're going to have functioning self-government in the United States or around the world" [3].

But, GnuPG is not only used for encrypting email. GnuPG protects the software updates of nearly all free software-based operating systems including Debian, Ubuntu, Red Hat, and SUSE. Although less common on the desktop, these systems power two-thirds of all web sites [4], and are the dominate platforms used in the cloud computing sector. That means that even if you don't directly use GnuPG, if you use the Internet, your personal data is, in part, being protected by GnuPG.

And, GnuPG is used for much more. People use it to protect data archives, such as, backups. Software distributors sign their software with it so that users can verify the integrity of a copy. Software developers use it to sign their commits [5]. Organizations like Debian use it to secure internal processes, such as making sure that a package upload is authorized, that a vote is legitimate, and that a resignation is authentic. GnuPG is used to secure Bitcoin wallets. And, GnuPG is used to sign documents.

1.1 History

Werner Koch started GnuPG in 1997 [6]. But GnuPG's roots lies in PGP, an encryption program originally written by Phil Zimmermann in 1991 [7]. Zimmermann was a long-time political activist, and wrote PGP to allow activists to securely store messages on BBSs. Although the source code for PGP was available, it wasn't free software. Further, due to its use of RSA for public-key cryptography, and IDEA for symmetric encryption, PGP was patent encumbered.

Around 1996, Richard Stallman, the founder of the Free Software Foundation, started appealing to people to create a free replacement for PGP. Koch was inspired by this speech, and began working on g10, as he initially called it, which was a reference to the tenth *Grundgesetz* (the tenth article of German the constitution), which enshrines the right to private communication in Germany. Since the reference was considered to be too obscure even for most Germans, the name *GNU Privacy Guard* or GnuPG, for short, was adopted soon after the initial release.

As of 2017, Koch has continued to work on GnuPG as the lead developer. Since its start, the project has remained relatively small in terms of the number of contributors. But, it was only in 2012 that Koch found himself working alone on GnuPG. Prior to that, the project received enough funding to employ a couple of developers. In 2012, however, GnuPG had a funding crisis, and Koch was forced to lay off his last employee. The funding situation continued to deteriorate, and in 2014 Koch had to take side jobs unrelated to GnuPG to supplement his income. The situation was unsustainable, and Koch nearly gave up. But, friends convinced him to give a donation campaign one last shot. The response was amazing. Not only did he receive enough money to fund himself, but he pulled in 250,000 euros in small donations, and Stripe, Facebook, and the Linux Foundation each committed to donating about 50,000 euros per year. Along with some partially unexpected contracts from the German BSI (the Federal Office for Information Security), Koch was able to hire five additional developers.

Since then, development of GnuPG has accelerated, and new features are being added on a regular basis. For instance, Koch developed a new key discovery protocol called the Web Key Directory (WKD) [8], there is a new trust model based on TOFU [9], there is official support for a set of Python bindings, and the GnuPG developers are actively contributing to Enigmail.

1.2 OpenPGP Criticism

OpenPGP has been widely criticized. There are three main criticisms: GnuPG isn't easy to use, GnuPG doesn't support deniability, like Off The Record (OTR), and GnuPG doesn't support forward secrecy.

Respond to: `https://medium.com/@mshelton/how-to-lose-friends-and-anger-journal`

Respond a bit more in depth to the Matthew Green blog post: `https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/`

Summarize Filippo articles: `https://arstechnica.com/security/2016/12/op-ed-im-giving-up-on-pgp/` `https://arstechnica.com/information-technology/2016/12/signal-does-not-replace-pgp/`

1.2.1 Usability

GnuPG is infamous for being hard to use. There is a fair amount of truth to this. Nevertheless, the argument can be made that some of the difficulties are required to achieve the support that it wants to achieve. For instance, it is unavoidable that people who are worried about active attackers need to think about authentication.

1.2.2 Deniability

Deniability (or deniable authentication) is the property that participants in a conversation are able to authenticate each other's messages, but they cannot later prove this to a third party. In OTR, this works by having the participants use a shared key for authenticating messages. Thus, if Alice knows that she didn't send a given authenticated message, then it must have come from Bob. This style of authentication is fundamentally different from digital signatures, which provide strong evidence that a particular person created or at least endorsed a signed message.

Why deniability is perhaps not so useful as one might imagine:

`https://debian-administration.org/users/dkg/weblog/104`

1.2.3 Forward Sececy

How important is forward secrecy?

1.3 Modern Chat Protocols

Over the past few years, the amount of activity in the encryption space has increased dramatically. One of the catalysts was almost certainly the Snowden leaks in June 2013, which not only motivated activists to do some work, but also sensitized the public to the work's importance. The area that has probably received the most attention has been in the end-to-end instant messaging space [10]. In particular, Signal, whose protocol has been adopted by WhatsApp and Google Allo, has received very strong endorsements from many prominent members of the InfoSec community. In fact, the creators of the Signal protocol, Moxie Marlinspike and Trevor Perrin, received the 2017 Levchin Prize at the Real World Crypto Symposium for their work on the protocol.

The first major difference between OpenPGP and Signal is with respect to their scope: signal focuses exclusively on real-time communication. This narrow focus has a number of advantages in terms of security. In particular, because communication is near real time, clients can negotiate parameters, and it is possible to implement forward secrecy.

The other major difference is that OpenPGP focuses on a decentralized model whereas these solutions tend to be walled gardens.

Signal uses the telephone number as a stable identifier, which is a strong identifier.

`https://jilliancyork.com/2017/08/03/i-dont-want-to-give-out-my-phon`

Unlike GnuPG, these tools focus on real-time communication.

1.4 Privacy

Address nothing to hide argument (that misses the point—everyone needs privacy).

1.5 Scope

As its title suggests, this book is intended to be an advanced introduction to GnuPG. It is explicitly *not* a reference manual. That is, the focus is not on providing a highly technical, exhaustive guide covering exactly what GnuPG does, but on gradually building up reader's understanding. This isn't a value judgment; I believe that the two are complementary. And, my hope is that after reading this book, you'll have a solid understanding of GnuPG's internals, and can quickly use GnuPG's reference manual to fill in any required details.

Chapter 2

A GnuPG Primer

Examples of how to use gpg from the command line. Cover all of the important stuff and little to none of the esoteric options. E.g., generating an online key, encryption, decryption, signing (inline or detached, clearsign), verifying sigs, using the `--edit-key` interface. Adding a new user id. Retiring a user id. Revoking a key. Signing someone's key. Setting owner trust. To armor or not to armor. Talk about importing and exporting keys (including import and export filters). Some useful options.

Listing keys. Talk about the different search methods, e.g., prefixing `@` to only search on the email.

Note that the right way to interact with GPG is not by screen scraping, but by using `gpg` 's `--status-fd` family of options or using the GPGME library (or one of the many bindings), which remove the need to parse `--status-fd` 's output.

GPG is not a library. Talk about how this arose historically. The tension between providing a user interface and a programming API (former wants convenience and implication, the latter not.) If you want to program GnuPG then it is recommended that you use GPGME (or a binding built on top of GPGME). A lower level interface is `--status-fd`. Has been around since GnuPG 1.2. Example of why it is important to use this interface.

Groups/aliases

Chapter 3

Cryptography

Most readers of this book probably already understand how public-key cryptography works. Perhaps not at the mathematical level, but at least at the conceptual. But, most readers of this book also need to be able to explain public-key cryptography for lay people.

1. What is cryptography? Basically scrambling a text (using permutation and substitution).

2. Example: most people have probably secured a zip file with a password.

3. How does that work? A simple approach is to imagine that each letter is a number—A is 0, B is 1, C is 2, etc.—and then add (without carrying—that is B (2) + Z (25) is 27, 27 is larger than 25, so do: 27 - 26 = 1 and take 1, i.e., do modular 26 arithmetic) the plain text to the password. For instance consider the text "Meet me in Mantua" and the password "tank boil throw letter".

   MEETMEININMANTUA

   - TANKBOILTHROWLETTER

   . . . .

   If you know the password, you can easily reverse the process. But if you don't know the password, it is effectively impossible to recover the plaintext given the ciphertext.

Note: if the password is at least as long the text and password is never reused, this is referred to as a one-time pad and is the strong known cryptography.

4. This approach doesn't scale. If you want to communicate with multiple people, you need a remember a password for each person.

5. Problem solved using public-key cryptography. Instead of sharing a password, each person has a so-called public key and a so-called private key. Using a public key cryptography, for Romeo to encrypt a message to Juliet, he just needs to know her public key. Juliet can decrypt the message using her private key. The nice thing about the public key is that it can be shared with anyone.

6. How does it work? Based on so-called one-way puzzles. Consider factoring `221`. To do this, you could try every number from 2 to the square root of 221 and see if it evenly divides 221. For 221, this doesn't take that long to do, but for a 1000 digit number, it could take forever—even for a computer and although there are some improvements over to the simply method, none are significantly faster. But, if I told you that the factors are 13 and 17, they you can *verify* that very quickly. This is basically how public-key cryptography works. There are also different one-way puzzles.

7. How to imagine public key encryption? We can think of the public key as the blue prints for a safe (or padlock) that anyone can build around a message, but once that message is in the safe, it can only be opened using the recipients corresponding private key.

Explain signing.

Give other examples of how to explain public key cryptography.

Talk about threat modeling. What are you trying to protect? From whom? What resources does the adversary have?

Chapter 4

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 recent version was published in 2007 as RFC 4880. In 2015, the OpenPGP community again reformed the OpenPGP working group to update the specification [11].

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" [12].

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.4.1 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.

4.4.2 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.

4.4.3 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@gnupg.net | \
> gpg --list-packets
gpg: encrypted with 2048-bit RSA key, ID C1A010A1D38C4BB8, created 2017-07-07
```

```
      "Juliet Capulet <juliet@gnupg.net>"
gpg: encrypted with 2048-bit RSA key, ID 5B905AF0423ABB52, created 201
      "Romeo Montague <romeo@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 extracted 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 [13], 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 [14].)

   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:

```
+------+-----------------------------------+-------------+
| SEIP | Data (e.g. a literal data packet) | MDC    hash |
+------+-----------------------------------+-------------+
        \                                      /   ^
         `------------------------------------'    |
                 SHA-1 ----------------------------'
```

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 [15].

4.5 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.

4.5.1 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: [16]). 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@gnupg.net"
gpg: Good signature from "Romeo Montague <romeo@gnupg.net>" [full]
gpg: Signature made Tue 11 Jul 2017 11:52:59 AM CEST
gpg:                using RSA key E5156E507DCB8D63AC89E5334954FDC67A46B4C5
gpg:                issuer "juliet@gnupg.net"
gpg: Good signature from "Juliet Capulet <juliet@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.

4.5.2 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.

4.5.3 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.

```
             ____________________________________________
    ,-----> /                                            \
+-----------+-----------+-----------+------+---------+---------+------
| A, last=1 | B, last=0 | C, last=1 | Data | C's sig | B's sig | A's s
+-----------+-----------+-----------+------+---------+---------+------
                  |        `----> \_____/
                   `----------------^
```

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.

4.6 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.

4.6.1 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 [17].

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.

4.6.2 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.

4.6.3 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@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 [18].

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.

4.7 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.

4.7.1 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.

4.7.2 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 [19]. This is what GnuPG does by default.

4.7.3 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.

4.7.4 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.

4.7.5 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@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.

4.8 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@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.

4.9 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 [20].

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.)

4.10 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.

Chapter 5

Passwords

What are passwords used for (symmetric encryption and protecting private key). Passwords are not used to protect asymmetric encryption. The reason for having a password is to protect the key if the device is compromised (e.g., malware or stolen). Thus, a weak password does not mean weak transport security; the security of the transport is the e.g. RSA encryption. If threat model is typical of a private individual, then using a password manager and a relative weak password is acceptable.

How to generate a strong password: need to be able to measure entropy. Long passphrase doesn't mean anything: if it is a line from a song, it is probably weak. NSA probably tries all of Wikipedia in various forms in the first few hours of trying to crack your password. The only secure way is to use diceware.

Snowden: "Assume your adversary is capable of one trillion guesses per second." To withstand one year, need 65 bits of entropy! How to measure a password's entropy? Need a random password. But that's impossible to memorize. Unless we encode it smartly!

5.1 Diceware

Encode using a simple word list

- /dev/random? 1k words (10-bits entropy per word)
- dice? $6^4 = 1296$ words (10.3-bits entropy)

Secure even if adversary knows the word list!

Examples:

1. able
2. about
3. above

Required length: 80 bits = good = 8 words 120 bits = strong = 12 words
Examples:

- percent burst able smash opposite ready blind stab
- pipe after harm person split seize radar about

Word lists: Diceware (8k). PGP Biometric word list (512). Voice of America's simple English word list (1.5k)

Chapter 6

Key Creation

Today, creating an OpenPGP key could hardly be easier or less error prone. It's as simple as thinking of a password and using `gpg`'s `--quick-gen-key` command:

```
$ gpg --quick-gen-key 'Juliet Capulet <juliet@gnupg.net>'
About to create a key for:
    "Juliet Capulet <juliet@gnupg.net>"

Continue? (Y/n) y
...
gpg: revocation certificate stored as
 '/home/jc/.gnupg/openpgp-revocs.d/98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2.re
public and secret key created and signed.

pub   rsa2048 2017-08-11 [SC] [expires: 2019-08-11]
      98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2
uid                      Juliet Capulet <juliet@gnupg.net>
sub   rsa2048 2017-08-11 [E]
```

(The above confirmation prompt can be suppressed by including the `--batch` option, which, as its name suggests, is designed for batch operations. For batch operations, the raw output of `gpg` shouldn't be parsed, but, instead, `--status-fd` should be used to get a stable interface. Also, in this case, `--pinentry-mode loopback --passphrase-fd X` can be used to supply a password.)

If you have multiple email addresses, then it is useful to also add them to your key if they are in the same trust domain. (For instance, work and private email should often be kept separate.) This can be done just as painlessly using the `--quick-add-uid` option:

```
$ gpg --quick-add-uid 98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2 \
> 'Juliet Capulet <juliet@riseup.net>'
```

For most users, the only important thing left to do is to backup the revocation certificate (this is explained in the next section), and publish the key, so that others can find it:

```
$ gpg --send-key 98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2
gpg: sending key EACAE136B8AF8CC2 to hkps://hkps.pool.sks-keyservers.ne
```

For users with stronger security requirements—those users who are not just worried about protecting their privacy—we recommend that they use a security token instead of an online key so that if the device is compromised, an attacker cannot get access to the secret key material. This is actually very easy: any program can normally access any file. Thus, using an online key, it is necessary to trust *all* programs that run on your system. Although setting up a security token—the focus of Section 6.3—is more work, the day-to-day use of a security token is no more complicated than an online key.

If you are replacing an existing key, then it is strongly recommended that you have the old key sign the new one, and the new key sign the old one so that there is strong, machine readable evidence that the two keys are controlled by the same party. This information is used by `gpg` in the TOFU trust model, for instance, to avoid spurious conflicts.

6.1 Keys Aren't Forever, Revocation Certificates Are

There are several reasons for why you might want to create a new key.

The most security relevant reason is that your current key could be compromised. This is the case if, for example, your device was infected with malware, or it was lost or stolen. In such cases, it is essential to immediately inform your communication partners that any messages encrypted to your key—not only new messages, but also messages encrypted prior to the compromise—could be read by a third party, and that signatures made by your key should no longer be trusted.

A less acute, but still important security relevant reason for creating a new key is that your old key no longer satisfies your security requirements. This could either be because your security requirements have changed, or the key has become weaker due to advances in cryptanalysis (the science of breaking cryptographic systems). An example of the latter is that 1024-bit RSA keys are no longer considered sufficiently strong to stop nation-state attackers. In reaction to this development, organizations like Debian now require members who have such keys to generate new ones. Although the immediate security implications are not as grave as above (unless you really think you are being targeted by a nation-state adversary), communication partners still need to be told to start using the new key.

A practical reason for creating a new key is that your old one has become inaccessible. This can happen if you forget your password, or you forget to migrate your key when reinstalling your system. Novice users are often beset by these problems. This usually leads to mails that can't be decrypted, because someone used the old key, which still appears to be valid. This not only inconveniences the recipient, because she can't decrypt the email, but also the sender, because the recipient will have to ask him to resend the email using her new key. This mistake appears to the users as a usability problem. But, it is worse; it is a security problem: users learn that encrypting email is fragile. And, taking this lesson to heart, they will reserve encryption for messages that "really" need protection. But, when the time comes, these users will be out of practice, which increases the chance that they mistakenly leave some important data unencrypted.

There is only one way to deal with these issues: the user needs to somehow inform her communication partners that her old key is no longer valid, and that a new one should be used instead. This can be done by talking to each person. But, this is inconvenient for everyone involved. Instead, it is easier, and less error prone to simply codify this into the system, which is what a revocation certificate does: it states that a particular key is no longer valid. And, a user's communication partners don't normally have to take any special actions: once uploaded to a key server, it will automatically be respected the next time the software refreshes the key.

A revocation certificate is only considered valid if it includes a self-signature. This presents a problem for users who lost access to their key: they can't create a revocation certificate! To mitigate this problem, version 2.1 of GnuPG automatically creates a revocation certificate when creating a new key. (The revocation certificate is stored in the `$GNUPGHOME/openpgp-revocs.d` directory.)

Because `gpg` doesn't know how the certificate will be used, it creates a generic revocation certificate. In some cases, it is useful to provide more details. But, in practice, this is rarely necessary: most users won't see the reason, and GnuPG treats the different reasons in the same way. Other OpenPGP implementations appear to do the same thing.

Automatically creating the revocation certificate when the key is created ensures that users who forget their password can still revoke their key. But, this doesn't help users who accidentally delete their key, e.g., by reinstalling their system, or forgetting to migrate it to a new computer. To protect against this mistake, it is essential that the revocation certificate be backed up on a separate system.

6.1.1 Backing Up a Revocation Certificate

To help ensure that the revocation certificate remains accessible when it is finally needed, it should be stored in multiple places, but not someplace that is easily accessible to an adversary. This is hard to do automatically.

One easy way to make a backup is to store the revocation certificate on the user's mail server. Of course, anyone with access to the mail account, such as the mail provider, could publish it. But, this is unlikely to happen in practice, and the consequences are more an inconvenience than a security issue: the user has to create a new key, and should tell her contacts what happened; the attacker is not able to forge signatures, or decrypt any messages.

Another way to make a backup is to display the revocation certificate as a QR code, and have the user photograph it. On Debian, this can be done using `qrencode`:

```
$ apt-get install qrencode
$ qrencode -o $FINGERPRINT.png < $FINGERPRINT.rev
$ eog $FINGERPRINT.png
```

Even if the user doesn't have a QR code scanner installed on her camera, it is possible to decode it later. This can be done, for instance, using ZBar, which is available on Debian as part of the `zbar-tools` package. Because the user probably won't know what the QR code is for after a few weeks, it is essential to add text next to the QR code to explain that the QR code contains a revocation certificate. The main security problem here is that many phones automatically backup data to the cloud, and, as above, the provider needs to be trusted to not publish it.

It is also reasonable to print the revocation certificate. Paper, for instance, has much better archival properties than many digital storage mediums, such as CD-ROMs. This can either be in text form (but this form is a pain to reenter) or as a QR code. Unfortunately, printers are no longer as secure as they once were: to simplify printing, some local printers are accessed via the cloud! But, even if this is not the case, in the very least, most are connected to the internet, and don't receive software updates. But, as before, the damage that an attacker who has a revocation certificate can cause is limited.

6.1.2 Publishing a Revocation Certificate

A key isn't really revoked until all communication partners have a copy of the revocation certificate. The easiest way to accomplish this is to import the revocation certificate locally, and then publish it on the public key servers.

```
$ gpg --import 98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2.rev
gpg: key EACAE136B8AF8CC2: "Juliet Capulet <juliet@riseup.net>" revocation cer
gpg: Total number processed: 1
gpg:    new key revocations: 1
$ gpg --send-key 98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2
gpg: sending key EACAE136B8AF8CC2 to hkps://hkps.pool.sks-keyservers.net
```

When using the revocation certificate that `gpg` automatically generated at key creation time, one more step is required: to prevent the user from accidentally importing the revocation certificate, GnuPG requires that the user first edit the file to remove a comment character. The contents of the file explain this, but importing the file does not provide a helpful error message (`gpg` just indicates that the file contains "no valid OpenPGP data"), and most users won't know to look at the files, which explains what to do:

```
...
To avoid an accidental use of this file, a colon has been inserted
before the 5 dashes below.  Remove this colon with a text editor
before importing and publishing this revocation certificate.

:-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: This is a revocation certificate

iQE2BCABCAAgFiEEmNuExW9W21z0czzN6srhNrivjMIFAlmNo+4CHQAACgkQ6srh
...
```

Assuming the user's communication partners have configured their software to automatically refresh keys, this should be enough. But it is nevertheless recommended that the updated key be sent to frequent communication partners to ensure a timely notification. This is best done by attaching a minimal key, which can be generated as follows:

```
$ gpg -a --export --export-options export-minimal \
> 98DB84C56F56DB5CF4733CCDEACAE136B8AF8CC2 > juliet-key.gpg
```

6.1.3 Recruiting Your Friends

Another way to deal with the revocation problem is to make a trusted third party a so-called *designated revoker*. This can be done by using the `addrevoker` subcommand in the `--edit-key` interface. For instance, Romeo could designate Juliet as a revoker for his key:

```
$ gpg --edit-key romeo@gnupg.net
Secret key is available.

sec  rsa2048/B003B1463C7B41BE
     created: 2017-08-11  expires: 2019-08-11  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa2048/50A1A6C84DBDEA6F
     created: 2017-08-11  expires: never       usage: E
[ultimate] (1). Romeo Montague <romeo@gnupg.net>

gpg> addrevoker

Enter the user ID of the designated revoker: 98DB84C56F56DB5CF4733CCDE
pub  rsa2048/EACAE136B8AF8CC2 2017-08-11 Juliet Capulet <juliet@riseup
 Primary key fingerprint: 98DB 84C5 6F56 DB5C F473  3CCD EACA E136 B8A
```

```
WARNING: appointing a key as a designated revoker cannot be undone!

Are you sure you want to appoint this key as a designated revoker? (y/N) y

This key may be revoked by RSA key EACAE136B8AF8CC2 Juliet Capulet <juliet@ris
sec  rsa2048/B003B1463C7B41BE
     created: 2017-08-11  expires: 2019-08-11  usage: SC
     trust: full          validity: full
ssb  rsa2048/50A1A6C84DBDEA6F
     created: 2017-08-11  expires: never       usage: E
[  full  ] (1). Romeo Montague <romeo@gnupg.net>

gpg> quit
Save changes? (y/N) y
```

Now, Juliet can generate a revocation certificate for Romeo's key using `gpg` 's `--generate-designated-revocation` command. The resulting revocation certificate is just like a normal revocation certificate. So, as above, Juliet would import this certificate, and then publish Romeo's key to cause it to be revoked.

6.2 Tweaking, Twiddling, and Frobbing

Prior to GnuPG 2.1, the `--quick-gen-key` command did not exist. To generate a key, users instead had to use the `--gen-key` command. The `--gen-key` command is like `--quick-gen-key`, but prompts the user to set a number of parameters. (This command, as well as its even more flexible variant, `--full-key-gen`, are still available.) But, in practice, few users need to use anything but the defaults. In fact, the defaults are even reasonable for almost all experts. Unfortunately, because the `--gen-key` command asks for user input, users appear to assume that the defaults need to be tweaked. And, this idea that the defaults are not sufficient is further reinforced by most GnuPG guides that suggest "better," more "secure" settings. In almost all cases, these guides only confuse, overwhelm, and scare users.

Now, it is almost certainly true that a larger key is harder to brute force than a smaller key. That is, a larger key increases the strength of the cryptography. But, larger key sizes have a cost. In particular, verifying a signature generated by a 4096-bit RSA key doesn't take twice as long as a 2048-bit RSA key, but orders of magnitude longer. So, the question is: does the stronger cryptography actually increase the system's security? Bruce Schneier argues that the Snowden leaks provide strong evidence that the NSA has not broken strong cryptography: when the NSA wants to access someone's data, they compromise the infrastructure and the endpoints, which is more costly, and more likely to be noticed [21]. Assuming Schneier is correct, since the strongest potential adversary in the world isn't breaking strong cryptography, further increasing the strength of the cryptography will *not* increase the system's security. Instead, to increase the system's security, it is better to protect the endpoint, and improve the user's operational security.

There are several things that can be done to better protect an endpoint. These include encrypting the storage device, using a good password for logging in, enabling a screen locker, promptly installing system updates, and avoiding malware. The next step is to better protect the cryptographic keys. This can be done by using a security token, which is a small piece of hardware that stores the cryptographic keys, and performs the basic cryptographic operations. In this way, if the end point is compromised, the keys are still safe. Although any data that was decrypted while the adversary controlled the computer could have been recovered.

For those few cases where it is necessary to override some defaults, it is still possible to use the `--key-gen` command or `--full-key-gen` command. And, it is also possible to modify a key using `gpg` 's `--edit-key` interface.

`gpg` also provides an interface for batch operations. See the "Unattended key generation" chapter of the GnuPG manual for details.

6.3 Security Tokens

A security token is a small piece of hardware that is typically connected to a computer via USB or NFC. The security token holds the keys and performs the primitive crypto operations, such as, decrypting a session key, or generating a signature. In this way, the secret key material never needs to be on the internet-connected device.

Not having the secret key material on the main device has a number of advantages: if an adversary wants to decrypt or sign messages, it is not sufficient to compromise the endpoint and exfiltrate the keys; the attacker needs to maintain a presence on the device, and wait for the user to insert the security token to decrypt or sign a message. For smartcard readers with an integrated PIN pad, the attacker also has to convince the user to enter the security token's PIN. This is a great defense, because although a user might enter the PIN a few times, most people will quickly become suspicious of many spurious prompts. Unfortunately, readers with an integrated PIN pad are bulky, which makes them inconvenient for mobile users. But, even without a PIN pad, security tokens significantly increase security. Of course, a security token can't prevent an attacker who has control over the endpoint from seeing any messages that are decrypted on the device.

A security token also has the advantage that if the device is compromised, it is not necessary to completely revoke the OpenPGP key. At most the signing subkey needs to be rotated (see Section 6.5), if the attacker might have signed a message. This is a great advantage, because the user's fingerprint stays the same, and certifications remain valid. Normally, creating a new OpenPGP key means that the user not only has to inform all of her contacts that she has a new key, and print new business cards, but she also has to recertify all of the keys that she signed, and convince people who signed her old key to sign her new one.

Given their security properties, and the relative ease of use once they are setup, we strongly recommend that anyone who relies on GnuPG to protect their security use a security token. Using an online key is reasonable for people who use GnuPG to protect their privacy or protect themselves from phishing excursions, or who want to hinder mass surveillance.

6.3.1 Hardware

There are a variety of security tokens that support OpenPGP. These have different advantages and disadvantages in terms of the degree to which they respect the user's freedom, their security properties, their feature sets, and their commercial availability. Below, we introduce a few that are known to work well with GnuPG.

OpenPGP Smartcard

The OpenPGP smartcard has been around since 2003. Although the software is proprietary, the specification is freely available and usable without constraints [22], and it has become the de facto interface for interacting with OpenPGP security tokens.

The main distributor is Kernel Concepts. They sell them to end users in their online shop, and they ship worldwide (`https://www.floss-shop.de/en`). These smartcards are relatively inexpensive. At the time of this writing, the FLOSS Shop sells them for 16.40 euros. But, because most systems don't include a smartcard reader, this hardware also needs to be purchased. Depending on the required features, in particular, whether an external PIN entry pad is desired, a smartcard reader currently costs between 20 euros and 50 euros.

Gnuk

The Gnuk security token (`https://www.fsij.org/category/gnuk.html`) was created by NIIBE Yutaka in 2012. His goal is to create a security token that not only uses free software, but whose schematics are freely available, and whose parts can be sourced by anyone. These constraints mean that the Gnuk does not use specialized security hardware that has been strengthened to prevent physical key extraction attacks, because this hardware's documentation is typically only available by signing an NDA, and can not be easily bought by individuals. But, although commodity hardware can't protect the user from key extraction attacks half as well as specially hardened hardware, it is much more difficult for an attacker to introduce a backdoor on all versions of the product. For instance, the MCU that Gnuk uses is produced by many manufacturers.

Using an alternate official firmware, the Gnuk can also function as a true random number generator (TRNG).

Nitrokey

The Nitrokey (`https://www.nitrokey.com/`) is based on the Gnuk, but is more feature rich. For instance, it supports OTP and U2F. And, some versions have on-board storage. The code is open source, but the schematics are not freely available.

YubiKey

YubiKey (`https://yubikey.com`) is a popular security token that has been adopted by some large organizations including Google and GitHub, and is sold by many major retailers. Like the Nitrokey, YubiKeys support several different security systems. But, not all YubiKeys support the OpenPGP card interface: support for this functionality is in the current product line is limited to the YubiKey NEO, and the YubiKey 4. The YubiKey NEO also supports NFC. This wireless interface makes it easy to use the YubiKey with a mobile phone. And, OpenKeychain, which is an OpenPGP implementation that integrates with the K-9 mailer on Android, supports it.

Like the OpenPGP smartcard, the YubiKey's firmware is proprietary. YubiKey used to release their OpenPGP applet as open source, but decided that due to their focus on security, and the inability to reflash the devices, making the source code available provides "little practical value" [23].

6.3.2 Creating a Key

Most security tokens include functionality to create an OpenPGP key. But, using this functionality is dangerous. First, given the limited hardware, and the typically proprietary software, the quality of the entropy is questionable. Unfortunately, the importance of high quality entropy for the security of the system can not be understated. For instance, the NSA is known to have backdoored the Dual_EC_DRBG pseudorandom number generator, a standard adopted by NIST, to allow them to recover encryption keys, and paid companies like RSA to make it the default in their products [24]. Second, security tokens do not provide an option to export keys. This limitation is highly desirable to prevent an attacker from recovering keys if the device is stolen or lost. But, it also means that it is not possible to backup the keys. Since OpenPGP is used to protect long-lived data, this is a serious limitation.

The alternative to creating an OpenPGP key on the security token is to create it on a computer, and then copy it to the security token. GnuPG supports this out of the box. But, with some distributions, such as Debian, GnuPG's smartcard support is packaged separately and not installed by default. On Debian and Ubuntu, GnuPG's smartcard support is included in the `scdaemon` package. Depending on the security token, it may also be necessary to install `pcscd`, which includes additional drivers.

Using An Offline Computer

If you are going to take the trouble to use a smartcard to separate your secret keys and your main system, then you can't use your main system to manage your keys.

There are two ways around this: you can either use a dedicated offline computer, or a live CD. The former is more secure, particularly, if you are willing to completely cut off its network access (i.e., air gap it by removing all network cards, bluetooth module, etc.). Given that used IBM Thinkpads are readily available for under 50 euros, this is the preferred solution. That said, for the majority of people who have modest security requirements, managing keys from a live CD that is booted from their main computer is reasonable.

Unfortunately, most live CDs are not appropriate for working with offline keys. Tails, however, is a GNU/Linux distribution that takes the necessary precautions. First, Tails starts as few services as possible to reduce the attack surface, and, with one clearly marked exception, it doesn't allow outbound network access except over Tor. And, when Tails shuts down, it wipes the system's memory. This is essential to make sure the keys are not accidentally exposed to an attacker. This protection is not just necessary to prevent cold boot attacks [25], which only those who require the highest levels of security have to worry about, but also to prevent the system after it restarts from accessing, and perhaps accidentally leaking the keys in uninitialized memory.

Whether you use a dedicated computer or reboot your normal computer into Tails, you need:

- A security token,
- The Tails distribution,
- A bootable storage device for Tails (at least 2 GB large),
- A storage device for your secret key material, and
- A storage device for exchanging files with your main system.

It doesn't matter whether the storage devices are USB keys, SD cards, hard drives, or something else. The important bit is that your computer can boot from the one for Tails, and the security token, and the storage devices can all be attached to the computer at the same time.

6.3.3 Tails

Tails is available from `https://tails.boum.org/`. The key used to sign the distribution is:

```
A490 D0F4 D311 A415 3E2B  B7CA DBB8 02B2 58AC D84F
```

First, download the ISO image, and the corresponding signature file. The latest version is linked to from `https://tails.boum.org/install/download/openpgp/`, and the files are named like `tails-amd64-VERSION.iso` and `tails-amd64-VERSION.iso.sig`.

Next, verify the signature. To do this, you need to first fetch the aforementioned key:

```
$ gpg --recv-key A490D0F4D311A4153E2BB7CADBB802B258ACD84F
gpg: requesting key 58ACD84F from hkp server keys.gnupg.net
gpg: key 58ACD84F: public key "Tails developers (offline long-term identity ke
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
$ gpg --verify tails-amd64-VERSION.iso.sig tails-amd64-VERSION.iso
gpg: Signature made Wed 09 Aug 2017 02:06:36 AM CEST
gpg:                using RSA key 79192EE220449071F589AC00AF292B44A0EDAA41
                                  ----------------------------------------
gpg: Good signature from "Tails developers (offline long-term identity key) <t
gpg:                 aka "Tails developers <tails@boum.org>" [undefined]
```

The `--verify` command shows us that the signature is "good" in the sense that the signature was really over the ISO image. But, it also shows us that the signature was generated by the key `79192EE220449071F589AC00AF292B44A0EDAA41`, not by the key `A490D0F4D311A4153E2BB7CADBB802B258ACD84F`! A close examination reveals that this is because the Tails developers used a signing subkey to make the signature. We can see this by using the `--list-keys` command to show more details about the key used to create the signature:

```
$ gpg -k 79192EE220449071F589AC00AF292B44A0EDAA41
pub   rsa4096/0xDBB802B258ACD84F 2015-01-18 [C] [expires: 2018-01-11]
      Key fingerprint = A490 D0F4 D311 A415 3E2B  B7CA DBB8 02B2 58AC D
                        ------------------------------------------
uid                   [  undef ] Tails developers (offline long-term i
uid                   [  undef ] Tails developers <tails@boum.org>
sub   rsa4096/0x98FEC6BC752A3DB6 2015-01-18 [S] [expires: 2018-01-11]
sub   rsa4096/0x3C83DCB52F699C56 2015-01-18 [S] [expires: 2018-01-11]
sub   rsa4096/0xAF292B44A0EDAA41 2016-08-30 [S] [expires: 2018-01-11]
```

Note: when you try this, you might see a different signing key. This is okay. What is important is that the main key is correct in the sense that the fingerprint matches: the user ID is not enough to prove the download's authenticity; creating a key with an arbitrary user ID, and uploading it to a key server is easy. By rotating keys, the Tails developers can reduce the amount of time that an undetected compromise of the signing key is useful to an attacker.

If `gpg --verify` indicates that the signature is bad, or `gpg -k` indicates that the signing key is not associated with `A490D0F4D311A4153E2BB7CADBB802B258AC` then there is a problem with the download. You should first try again from a different network. If the problem persists, seek help from an expert. Since Tails is used by people who are trying to protect sensitive information, there are bound to be copies that have been modified to include malware; **do not use the ISO image if you (or someone you trust) can't verify it**.

Assuming the data is okay, it is now possible to copy the Tails ISO to a USB key. This can be done using `dd`:

```
$ sudo dd if=tails-amd64-VERSION.iso of=/dev/sdX bs=4096
```

(`sdX` should be replaced by the name of the actual storage device.)

Then, boot into Tails.

Before logging it, you can set the `root` password so that you can, for instance, install additional (optional) software. You can do this at the Tails login screen by going to *Additional Settings » Administration Password*. Note: if you are worried about a targeted attack, you should not connect the computer to the network, and this step can be skipped.

The Tails website has alternate installation and verification instructions. It is reasonable to follow them. Particularly, if you don't have a Unix-like system with `gpg` already installed, which these instructions take for granted.

6.3.4 Initializing the Security Token

Once Tails has started, the first thing to do is to make sure that it recognizes the security token. Insert the security token, and then issue `gpg` 's `--card-status` command. The output should be similar to the following:

```
$ gpg --card-status
Reader ...........: 04E6:E003:21251019201732:0
Application ID ...: D2760001240102010005000002D9D0000
Version ..........: 2.1
Manufacturer .....: ZeitControl
Serial number ....: 00002D9D
Name of cardholder: [not set]
Language prefs ...: de
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: rsa2048 rsa2048 rsa2048
Max. PIN lengths .: 32 32 32
PIN retry counter : 3 0 3
Signature counter : 0
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
General key info..: [none]
```

Looking at the `Manufacturer` field, we see that the security token is an OpenPGP smartcard. Specifically, we know from the `Version` field that it implements version 2.1 of the OpenPGP card specification. Like all OpenPGP smartcards, it supports three keys: a signing key, an encryption key, and an authentication key. The `Signature key`, `Encryption key` and `Authentication key` fields tell us that no keys are currently stored on the card. The `Key attributes` field indicates that this particular

OpenPGP card supports up to 2048-bit RSA keys. Newer versions of the card support 4096-bit RSA keys.

If the card has already been used, then it first needs to be reset. This can be done using `--card-edit` 's `factory-reset` subcommand. If the factory reset command indicates that the card is not supported, as below, you may need to consult your security token's documentation:

```
$ gpg --card-edit
...
gpg/card> admin
Admin commands are allowed

gpg/card> factory-reset
gpg: OpenPGP card no. D276000124010200FFFE50FF6C060000 detected
gpg: This command is not supported by this card
```

But, the following commands usually work:

```
$ gpg-connect-agent
/hex
scd apdu 00 20 00 81 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 81 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 81 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 81 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 83 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 83 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 83 08 40 40 40 40 40 40 40 40
scd apdu 00 20 00 83 08 40 40 40 40 40 40 40 40
scd apdu 00 e6 00 00
scd apdu 00 44 00 00
```

The first four lines after the `/hex` directive enter a bad user PIN. The next four enter a bad admin PIN. Then, the last two lines terminate, and reactivate the card, respectively. After removing the security token and reconnecting it, it should be reset. You can verify this by running `gpg --card-edit`, checking that the card contains no keys, and then using the `verify` subcommand to make sure the user PIN has been reset to the default (normally `123456`).

Next, we need to change the default PINs. The OpenPGP card has two PINs: a user PIN and an admin PIN. The user PIN is used on a day-to-day basis to authorize decryption and signing; the admin PIN allows adding new keys, among other things. The admin PIN should not be used on the main computer, and it should be different from the user PIN. The default PINs are usually `123456` and `12345678`, respectively.

To change the PINs, we need to first enable admin mode. Then, we can use the `passwd` command to change each PIN. GnuPG will prompt you to enter the old PIN and the new PIN. If you are using a PIN pad, this can be confusing: you need to enter the new PIN twice.

```
$ gpg --card-edit
...
gpg/card> admin
Admin commands are allowed

gpg/card> passwd
gpg: OpenPGP card no. D2760001240102010005000002D9D0000 detected

1 - change PIN
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection? 1
PIN changed.

1 - change PIN
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection? 3
PIN changed.

1 - change PIN
```

```
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection? q
```

You'll almost certainly want to write the admin PIN down someplace. Given how often it is normally used, most people forget it. A good place to hide it is at a physically different location from where you hide the USB key with your secret keys on it.

6.3.5 Formatting the Removable Storage Devices

As mentioned above, we need to format two storage devices: one for the secret keys, and one for the public keys. Assuming the partition holding the private key material is encrypted with a strong passphrase, it is reasonable to just have two partitions on a single memory card. But, if possible, it is better to never insert the USB key with the secret key material into an insecure computer.

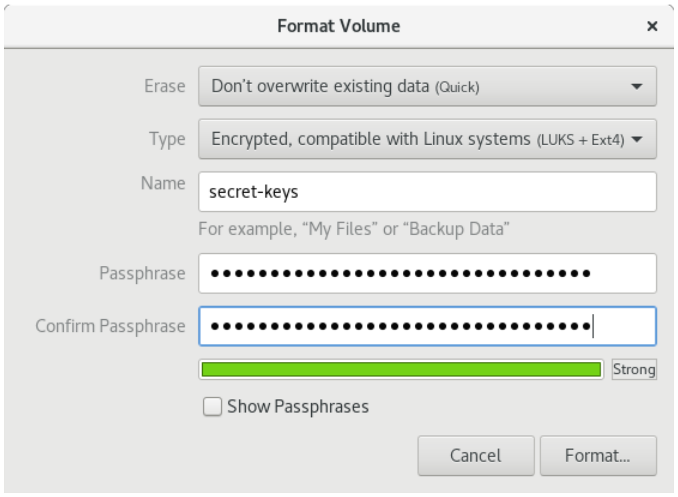
The Encrypted File System

When we create the OpenPGP key, we need to store it somewhere. Although it is possible to install Tails and create a persistent volume, it is better to use a separate storage device. This way, upgrading your Tails "installation" is trivial: you just need to download a new live CD, and copy it to your dedicated Tails device; there is no data to migrate.

The amount of storage space for secret keys material doesn't need to be large: 10 megabytes is more than sufficient. But, it should be encrypted. The easiest way to create an encrypted file system in Tails is to use GNOME Disks (*Applications » Utilities » Disks*).

In GNOME Disks, select the USB key, create a new volume, set the label to "secret-keys", and format it as a "LUKS + Ext4" file system. See Figure 6.1. GNOME Disks does not automatically mount the new file system. But, you can do this by selecting the partition, and clicking on the "play" button. The file system will be mounted under `/media/amnesia/secret-keys`.

You'll probably also want to write this passphrase on the piece of paper with your admin PIN.


Format Volume

Erase: Don't overwrite existing data (Quick)

Type: Encrypted, compatible with Linux systems (LUKS + Ext4)

Name: secret-keys

For example, "My Files" or "Backup Data"

Passphrase: ••••••••••••••••••••••••••••••••

Confirm Passphrase: ••••••••••••••••••••••••••••••••

Strong

Figure 6.1: Formatting an encrypted partition in GNOME Disks.

The Sneaker Net File System

Unfortunately, most security tokens don't store the public key or have any storage space on them. Thus, to transfer the public key to the main computer, we need a third storage device.

This time, when partitioning the file system, name the file system `sneaker-net`, and use an unencrypted `ext4` file system. After it has been formatted, mount the partition. The file system will be mounted under `/media/amnesia/sneaker-net`.

6.3.6 Generating the Keys

We can finally generate the keys. At the beginning of this chapter, we generated a simple key that had a single subkey. When using a security token, we also want at least a signing subkey. If you plan to use your security token to authenticate `ssh` connections, then you'll also need an authentication subkey. The reason to have a signing subkey is to further isolate the powerful certification-capable key from your online devices: the certification key not only determines your key's identity, but it is also used to create new subkeys. Thus, using the master key not only as a certification key, but also a signing key would mean that if the security token is lost, then we'd have to create a new OpenPGP key. If this happens when using a separate signing subkey, it is only necessary to revoke the signing subkey, and create a new one.

The following transcript shows how to create a certification-capable master key, and three subkeys. Note that we first change `gpg` 's home directory to be on the encrypted file system.

```
$ mkdir -p /media/amnesia/secret-keys/gnupg
$ export GNUPGHOME=/media/amnesia/secret-keys/gnupg
$ gpg --quick-gen-key 'Juliet Capulet <juliet@gnupg.net>' rsa cert 2y
gpg: WARNING: unsafe permissions on homedir '/media/amnesia/secret-keys
gpg: keybox '/media/amnesia/secret-keys/gnupg/pubring.kbx' created
gpg: /media/amnesia/secret-keys/gnupg/trustdb.gpg: trustdb created
gpg: key E9794A89BDB70380 marked as ultimately trusted
gpg: directory '/media/amnesia/secret-keys/gnupg/openpgp-revocs.d' crea
gpg: revocation certificate stored as '/media/amnesia/secret-keys/gnupg
public and secret key created and signed.

Note that this key cannot be used for encryption.  You may want to use
the command "--edit-key" to generate a subkey for this purpose.
pub   rsa2048 2017-08-14 [C] [expires: 2019-08-14]
      635D6A0EA043F835A1FFD9A7E9794A89BDB70380
uid                      Juliet Capulet <juliet@gnupg.net>

$ gpg --quick-addkey 635D6A0EA043F835A1FFD9A7E9794A89BDB70380 rsa encr
$ gpg --quick-addkey 635D6A0EA043F835A1FFD9A7E9794A89BDB70380 rsa sign
$ gpg --quick-addkey 635D6A0EA043F835A1FFD9A7E9794A89BDB70380 rsa auth
```

Listing the key, we can see that we got the desired structure:

```
$ gpg -K
gpg: WARNING: unsafe permissions on homedir '/media/amnesia/secret-keys
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2019-08-14
/media/amnesia/secret-keys/gnupg/pubring.kbx
--------------------------------------------
sec   rsa2048 2017-08-14 [C] [expires: 2019-08-14]
      635D6A0EA043F835A1FFD9A7E9794A89BDB70380
```

```
uid           [ultimate] Juliet Capulet <juliet@gnupg.net>
ssb   rsa2048 2017-08-14 [E] [expires: 2018-08-14]
ssb   rsa2048 2017-08-14 [S] [expires: 2018-08-14]
ssb   rsa2048 2017-08-14 [A] [expires: 2018-08-14]
```

Because the OpenPGP card doesn't include the public key, it is necessary to use the sneaker net storage device to transfer the public key from the offline machine to the online machine:

```
$ gpg -a --export 635D6A0EA043F835A1FFD9A7E9794A89BDB70380 > \
> /media/amnesia/sneaker-net/635D6A0EA043F835A1FFD9A7E9794A89BDB70380.pub
```

If your Tails machine is connected to the Internet, you could transfer the public key by uploading it to a key server or a website, and then retrieving it with the online machine.

6.3.7 Saving Your Progress

Just in case, we make a mistake, it is useful to create a snapshot of the secret key material:

```
$ gpg --export-secret-keys > /media/amnesia/secret-keys/secret-keys.gpg
```

To restore, we can import the key and mark it as ultimately trusted:

```
$ rm -rf /media/amnesia/secret-keys/gnupg
$ mkdir /media/amnesia/secret-keys/gnupg
$ gpg --import secret-keys.gpg
gpg: WARNING: unsafe permissions on homedir '/media/amnesia/secret-keys/gnupg'
gpg: keybox '/media/amnesia/secret-keys/gnupg/pubring.kbx' created
gpg: /media/amnesia/secret-keys/gnupg/trustdb.gpg: trustdb created
gpg: key E9794A89BDB70380: public key "Juliet Capulet <juliet@gnupg.net>" impo
gpg: key E9794A89BDB70380: secret key imported
gpg: Total number processed: 1
gpg:               imported: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1
$ gpg --edit-key juliet@gnupg.net
...
```

```
Please decide how far you trust this user to correctly verify other use
(by looking at passports, checking fingerprints from different sources,

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y
```

6.3.8 Creating a Backup

The Revocation Certificate

It is almost certainly reasonable to store the revocation certificate on the main computer. Thus, we can copy it to our sneaker net device:

```
$ cp /media/amnesia/secret-keys/gnupg/openpgp-revocs.d/635D6A0EA043F835
> /media/amnesia/sneaker-net
```

Just remember to actually copy it to someplace that is regularly backed up.

The Secret Key

Backing up the secret key is more difficult, because its security requirements are much higher given the consequences of a compromise. Paperkey is a relatively convenient way to back up secret key material on paper. To do this, you have to to attach and configure a printer. Make sure the printer is not connected to the network, and reset the printer afterwards.

```
$ paperkey --secret-key /media/amnesia/secret-keys/secret-keys.gpg
# Secret portions of key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
# Base16 data extracted Mon Aug 14 13:36:43 2017
# Created with paperkey 1.3 by David Shaw
#
# File format:
```

```
# a) 1 octet:  Version of the paperkey format (currently 0).
# b) 1 octet:  OpenPGP key or subkey version (currently 4)
# c) n octets: Key fingerprint (20 octets for a version 4 key or subkey)
# d) 2 octets: 16-bit big endian length of the following secret data
# e) n octets: Secret data: a partial OpenPGP secret key or subkey packet as
#              specified in RFC 4880, starting with the string-to-key usage
#              octet and continuing until the end of the packet.
# Repeat fields b through e as needed to cover all subkeys.
#
# To recover a secret key without using the paperkey program, use the
# key fingerprint to match an existing public key packet with the
# corresponding secret data from the paper key.  Next, append this secret
# data to the public key packet.  Finally, switch the public key packet tag
# from 6 to 5 (14 to 7 for subkeys).  This will recreate the original secret
# key or secret subkey packet.  Repeat as needed for all public key or subkey
# packets in the public key.  All other packets (user IDs, signatures, etc.)
# may simply be copied from the public key.
#
# Each base16 line ends with a CRC-24 of that line.
# The entire block of data ends with a CRC-24 of the entire block of data.

  1: 00 04 63 5D 6A 0E A0 43 F8 35 A1 FF D9 A7 E9 79 4A 89 BD B7 03 80 1D7942
  2: 02 B9 FE 07 03 02 E7 84 42 CB 86 E4 73 CA DB 47 A2 C0 4F 0A BB 57 2C46C8
  3: 04 63 BF E6 11 52 C4 F3 7A BB 12 34 66 DB 79 5A 89 E1 C2 8D 2E 10 603062
  4: 0B 3D 57 0A FD ED 8A 97 71 0B 51 EB 31 C4 02 28 C1 6E 64 18 B9 2C 8470F7
...
132: C5CD3A
```

Note: when you restore the key, you'll still need the key's passphrase.

Another way to backup the secret key is to copy it to another encrypted storage device.

6.3.9 Copying the Keys to the Security Token

We can finally copy the keys to the security token. This is done using `gpg` 's `--card-edit` interface. To add keys to the card, you'll need to enter the admin PIN, not the user PIN.

The subcommand to move a key to a security token is `keytocard`. `keytocard` works on the currently active subkey. The `key` subcommand is used to select or deselect a key. Since some operations can operate on multiple keys, it is necessary to explicitly deselect keys. Selected keys are shown with a ⋆.

```
$ gpg --edit-key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
Secret key is available.

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb  rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb  rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> key 1    # Select the first subkey.

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb* rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb  rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> keytocard
Please select where to store the key:
   (2) Encryption key
Your selection? 2
...
gpg> key 2    # Select the second subkey.
```

```
sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb* rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb* rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> key 1    # Deselect the first subkey.

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb  rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb* rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> keytocard
Please select where to store the key:
   (1) Signature key
   (3) Authentication key
Your selection? 1
...
gpg> key 3    # Select the third subkey.
...
gpg> key 2    # Deselect the second subkey.
...
gpg> keytocard
Please select where to store the key:
   (3) Authentication key
Your selection? 3
```

```
...
gpg> quit
Save changes? (y/N) n
Quit without saving? (y/N) y
```

At the end of the transcript, we explicitly do *not* save the changes. Saving the changes would cause the secret key material for the corresponding subkey to be deleted. The reason for this is that the `keytocard` command doesn't copy, but *moves* the secret key material to the card. Saving without quitting inhibits this side effect. In practice, this isn't a problem if you backed up the secret key material, as recommended above.

Using `--card-status`, we can see that the keys were successfully loaded on to the security token:

```
$ gpg --card-status
Reader ...........: SCM Microsystems Inc. SPR 532 [Vendor Interface] (
Application ID ...: D276000124010201000500002D9D0000
Version ..........: 2.1
Manufacturer .....: ZeitControl
Serial number ....: 00002D9D
Name of cardholder: [not set]
Language prefs ...: de
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: rsa2048 rsa2048 rsa2048
Max. PIN lengths .: 32 32 32
PIN retry counter : 3 0 3
Signature counter : 0
Signature key ....: A17A D462 C473 51AD D0E8  988B 305A 8468 03A9 1753
      created ....: 2017-08-14 13:07:49
Encryption key....: C9CD 8F3D ECDE BB7E 720A  7CD9 F8D8 ED7B B1A2 A8F6
      created ....: 2017-08-14 13:07:32
Authentication key: 9308 3590 BCD9 3CC5 044C  BEAD 300B A8EE 1B5E DEED
      created ....: 2017-08-14 13:08:04
General key info..: sub  rsa2048/305A846803A91753 2017-08-14 Juliet Ca
sec   rsa2048/E9794A89BDB70380  created: 2017-08-14  expires: 2019-08-
```

```
ssb   rsa2048/F8D8ED7BB1A2A8F6  created: 2017-08-14  expires: 2018-08-14
ssb   rsa2048/305A846803A91753  created: 2017-08-14  expires: 2018-08-14
ssb   rsa2048/300BA8EE1B5EDEED  created: 2017-08-14  expires: 2018-08-14
```

You can now shut Tails down.

6.3.10 Using the Keys

To actually use the keys on the security token, we need to do four more minor things.

First, plug the security token into your computer, and run `--card-status`. This command makes sure that `gpg` can actually see the card:

```
$ gpg --card-status
Reader ...........: SCM Microsystems Inc. SPR 532 [Vendor Interface] (21251019
Application ID ...: D2760001240102010005000002D9D0000
Version ..........: 2.1
Manufacturer .....: ZeitControl
Serial number ....: 00002D9D
Name of cardholder: [not set]
Language prefs ...: de
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: forced
Key attributes ...: rsa2048 rsa2048 rsa2048
Max. PIN lengths .: 32 32 32
PIN retry counter : 3 0 3
Signature counter : 0
Signature key ....: A17A D462 C473 51AD D0E8  988B 305A 8468 03A9 1753
      created ....: 2017-08-14 13:07:49
Encryption key....: C9CD 8F3D ECDE BB7E 720A  7CD9 F8D8 ED7B B1A2 A8F6
      created ....: 2017-08-14 13:07:32
Authentication key: 9308 3590 BCD9 3CC5 044C  BEAD 300B A8EE 1B5E DEED
      created ....: 2017-08-14 13:08:04
General key info..: [none]
```

Although the security token is recognized (and the keys are loaded), you can't yet use the keys, because `gpg` is missing the public keys. Insert and mount the sneaker net device, and import them:

```
$ gpg --import /media/juliet/sneaker-net/635D6A0EA043F835A1FFD9A7E9794A
gpg: Total number processed: 1
gpg:               imported: 1
```

Listing the secret key, we can see that it is now available.

```
$ gpg -K 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
sec#  rsa2048/0xE9794A89BDB70380 2017-08-14 [C] [expires: 2019-08-14]
      Key fingerprint = 635D 6A0E A043 F835 A1FF  D9A7 E979 4A89 BDB7 
uid                   [unknown] Juliet Capulet <juliet@gnupg.net>
ssb>  rsa2048/0xF8D8ED7BB1A2A8F6 2017-08-14 [E] [expires: 2018-08-14]
ssb>  rsa2048/0x305A846803A91753 2017-08-14 [S] [expires: 2018-08-14]
ssb>  rsa2048/0x300BA8EE1B5EDEED 2017-08-14 [A] [expires: 2018-08-14]
```

The # after the `sec` header for the master key means that the master key is not available; the > next to the `ssb` keys means that the keys are on a security token.

Looking at the above output, we also see that the key is not marked as trusted. In order for certifications by this key to be respected, it is necessary to mark the key as ultimately trusted. This can be done using the `--edit-key` interface.

```
$ gpg --edit-key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
Secret key is available.
...
gpg> trust
...
Please decide how far you trust this user to correctly verify other use
(by looking at passports, checking fingerprints from different sources,

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5
Do you really want to set this key to ultimate trust? (y/N) y
...
```

Finally, you'll also want to publish your key so that others can more easily find it.

```
$ gpg --send-key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
```

6.3.11 Saving the Revocation Certificate

Before unmounting the device, there is one last thing that we need to do: we need to copy the revocation certificate someplace that is backed up.

```
$ gpg --import /media/juliet/sneaker-net/635D6A0EA043F835A1FFD9A7E9794A89BDB70
> ~/.gnupg
gpg: Total number processed: 1
gpg:               imported: 1
```

6.3.12 Signing Keys with an Offline Master

When it comes to signing keys, having an offline master key is a pain: it is necessary to download the public keys on a network-connected computer, transfer them to a removable storage device, sign them on the offline computer, and then move the signatures back to the main computer. The key signing tool `caff` partially automates this workflow, but it is still inconvenient.

For most users, it would be nice to somehow separate the "modify the key" and "certify other keys" capabilities: only the former capability is highly sensitive.

With a few small tricks, this is possible. The basic idea is to create a secondary, certification-only key, which is not offline, and to have the offline key designate it as a trusted introducer using a so-called *trust signature*. Then, you sign other people's keys using the second key, and ask people sign your main key. If anyone sets your main key as a trusted introducer, then they will automatically trust your secondary key by way of the trust signature. If the second key is somehow compromised, it can be revoked, and replaced with a new key. And, any signatures can be recreated with the new key.

To create a certification-only key, do the following:

```
$ gpg --quick-gen-key 'Juliet Capulet (certification key)' rsa cert 2y
gpg: key 0xCD6AF594BAA8EF38 marked as ultimately trusted
```

```
gpg: revocation certificate stored as '/home/us/.gnupg/openpgp-revocs.c
public and secret key created and signed.

Note that this key cannot be used for encryption.  You may want to use
the command "--edit-key" to generate a subkey for this purpose.
pub   rsa2048/0xCD6AF594BAA8EF38 2017-08-14 [C] [expires: 2019-08-14]
      Key fingerprint = 149D 0735 A25E 63B1 EC9F  EEBD CD6A F594 BAA8 E
uid                              Juliet Capulet (certification key)
```

There are two things to note about this key. First, the user ID doesn't include an email address. This is useful to prevent people who search the key servers by email address from finding the wrong key. Although searching a key server by email address is strongly discouraged, there is no need to make such users' lives worse than necessary. Second, the key only includes a certification-capable key; there are no signing or encryption subkeys. This prevents people from accidentally encrypting data to your secondary key, or a misconfigured GnuPG from accidentally using it to generate a signature.

After creating the key, we need to sign it using the main key. This requires transferring the public key to the offline computer.

```
$ gpg --export 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38 > \
> /media/juliet/sneaker-net/149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38.pu
```

Then running the following on the offline computer after remounting the encrypted file system with the secret key, and the sneaker net file system:

```
$ export GNUPGHOME=/media/amnesia/secret-keys/gnupg
$ gpg --import /media/amnesia/sneaker-net/149D0735A25E63B1EC9FEEBDCD6AF
gpg: key CD6AF594BAA8EF38: public key "Juliet Capulet (certification ke
gpg: Total number processed: 1
gpg:               imported: 1
$ gpg --edit-key 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38
pub  rsa2048/CD6AF594BAA8EF38
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: unknown       validity: unknown
[ unknown] (1). Juliet Capulet (certification key)
```

```
gpg> tsign

pub  rsa2048/CD6AF594BAA8EF38
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: unknown       validity: unknown
 Primary key fingerprint: 149D 0735 A25E 63B1 EC9F  EEBD CD6A F594 BAA8 EF38

     Juliet Capulet (certification key)

This key is due to expire on 2019-08-14.
Please decide how far you trust this user to correctly verify other users' key
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I trust marginally
  2 = I trust fully

Your selection? 2

Please enter the depth of this trust signature.
A depth greater than 1 allows the key you are signing to make
trust signatures on your behalf.

Your selection? 2

Please enter a domain to restrict this signature, or enter for none.

Your selection?

Are you sure that you want to sign this key with your
key "Juliet Capulet <juliet@gnupg.net>" (E9794A89BDB70380)

Really sign? (y/N) y

gpg> Save changes? (y/N) y
$ gpg --export-options backup \
> --export 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38 > \
> /media/amnesia/sneaker-net/149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38.pub
```

Using a value of 2 for the depth parameter to the `tsign` subcommand means that anyone who considers the main key to be a trusted introducer will also consider the certification key to be a trusted introducer. (Using a value of 1 is equivalent to a normal signature, which simply verifies the target, but does not claim that the key's own should be treated as a trusted introducer.)

It is possible to restrict the domains over which the trusted signature is valid. But, support for this in GnuPG is incomplete. For instance, this feature is currently not supported on Windows.

After creating the signature, move the signed public key back to the main computer, import it, and publish it.

```
$ gpg --import /media/juliet/sneaker-net/149D0735A25E63B1EC9FEEBDCD6AF5
gpg: key 0xCD6AF594BAA8EF38: "Juliet Capulet (certification key)" 1 new
gpg: WARNING: server 'gpg-agent' is older than us (2.1.18 < 2.1.23-beta
gpg: Note: Outdated servers may lack important security fixes.
gpg: Note: Use the command "gpgconf --kill all" to restart them.
gpg: Total number processed: 1
gpg:          new signatures: 1
# gpg --send-key 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38
```

It is also a good idea to have the certification key cross sign the main key. (The `-u` option indicates what key to use for the signature. This is useful if you have multiple keys.)

```
$ gpg -u 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38 \
> --edit-key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
Secret key is available.

pub  rsa2048/0xE9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb  rsa2048/0xF8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
     card-no: 0005 00002D9D
ssb  rsa2048/0x305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
     card-no: 0005 00002D9D
ssb  rsa2048/0x300BA8EE1B5EDEED
```

```
     created: 2017-08-14  expires: 2018-08-14  usage: A
     card-no: 0005 00002D9D
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> sign

pub  rsa2048/0xE9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
 Primary key fingerprint: 635D 6A0E A043 F835 A1FF  D9A7 E979 4A89 BDB7 0380

     Juliet Capulet <juliet@gnupg.net>

This key is due to expire on 2019-08-14.
Are you sure that you want to sign this key with your
key "Juliet Capulet (certification key)" (0xCD6AF594BAA8EF38)

Really sign? (y/N) y

gpg> Save changes? (y/N) y
```

6.4 Key Expiration

OpenPGP includes a key expiry mechanism. An expired key is like a revoked key: `gpg` won't encrypt a message to it or rely on it. But, unlike a revocation certificate, the expiry information is published immediately. Thus, if the user loses access to the key *and* the revocation certificate, the key will still eventually be considered invalid. This is particularly useful for users who uninstall GnuPG without first publishing a revocation certificate. Another difference is that whereas revocation certificates can't be rescinded once they are published, it is possible to change when a key expires. This is important as keys are intended to be long-term identities, but the time until a key expires should be relatively short.

As of GnuPG 2.1, GnuPG automatically sets newly created primary keys to expire in two years. The easiest way to change the expiration time is to use the `--quick-set-expire` command. The following command sets the primary key to expire in two years relative to when the command was issued:

```
$ gpg --quick-set-expire 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38 2y
```

If the expiration of a subkey needs to be extended, this can be done as follows:

```
$ gpg --quick-set-expire 149D0735A25E63B1EC9FEEBDCD6AF594BAA8EF38 2y \
> C9CD8F3DECDEBB7E720A7CD9F8D8ED7BB1A2A8F6
```

Unfortunately, using `--quick-set-expire` to extend a subkey's expiration is only supported since 2.1.22, which was released in July 2017. As such, this functionality is not supported by the version of GnuPG shipped with Debian 9 (stretch). In this case, it is necessary to use the `expire` command from the `--edit-key` interface to extend a subkey's expiration.

After extending a key's expiration, don't forget to publish the updated key (e.g., using `--send-key`) so that your communication partners see the change. If the expiration is adjusted on an offline computer, you'll need to first export the updated key (`--export`) to a removable storage device, and then import it on an online computer.

To better avoid the problems associated with an expired key, it is better to extend the expiration date two or three months before the expiry so that any automatic update mechanism picks up the change, before the user sees an error.

6.5 Subkey Rotation

A user can approximate forward secrecy by regularly rotating her encryption and signing subkeys [17]. Unfortunately, since endpoint security—not the cryptography—tends to be the weak point in the system, this only really makes sense for people who store their keys on a security token. Unfortunately, if you are using a security token, then you won't be able to store both your old keys and your new keys on the same token: the OpenPGP card specification only supports a single encryption key. Although it is possible to carry multiple security tokens, this is often inconvenient. In particular, it becomes unmanageable after a few key rotations. Thus, in practice, this only makes sense if you are willing to forego access to old encrypted data, which is not how most people use OpenPGP.

Rotating keys is as simple as revoking the old keys, and generating new subkeys. To revoke a subkey, it is necessary to use the `--edit-key` interface, select the subkeys to revoke using the `key` subcommand, and use the `revkey` subcommand to revoke the selected subkeys. This is illustrated below:

```
$ gpg --edit-key 635D6A0EA043F835A1FFD9A7E9794A89BDB70380
Secret key is available.

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb  rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb  rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> key *   # Select all keys.  You can also use key N
             # to select the Nth subkey.

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
ssb* rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  expires: 2018-08-14  usage: E
ssb* rsa2048/305A846803A91753
     created: 2017-08-14  expires: 2018-08-14  usage: S
ssb* rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  expires: 2018-08-14  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>

gpg> revkey
Do you really want to revoke the selected subkeys? (y/N) y
Please select the reason for the revocation:
  0 = No reason specified
  1 = Key has been compromised
  2 = Key is superseded
  3 = Key is no longer used
  Q = Cancel
Your decision? 2
Enter an optional description; end it with an empty line:
>
Reason for revocation: Key is superseded
(No description given)
Is this okay? (y/N) y

sec  rsa2048/E9794A89BDB70380
     created: 2017-08-14  expires: 2019-08-14  usage: C
     trust: ultimate      validity: ultimate
The following key was revoked on 2017-08-15 by RSA key E9794A89BDB7038
ssb  rsa2048/F8D8ED7BB1A2A8F6
     created: 2017-08-14  revoked: 2017-08-15  usage: E
The following key was revoked on 2017-08-15 by RSA key E9794A89BDB7038
ssb  rsa2048/305A846803A91753
     created: 2017-08-14  revoked: 2017-08-15  usage: S
The following key was revoked on 2017-08-15 by RSA key E9794A89BDB7038
ssb  rsa2048/300BA8EE1B5EDEED
     created: 2017-08-14  revoked: 2017-08-15  usage: A
[ultimate] (1). Juliet Capulet <juliet@gnupg.net>
```

Chapter 7

Validating Keys

Why is validation important? (What is a MitM attack? Why can't keys be validated by a machine?)

How validation works on the web: x509—centralized and completely broken.

Traditional way to do this in the OpenPGP world is to use WoT. Describe how the WoT works. Talk about why it is hard to use: Key signing parties are for geeks. Even exchanging fingerprints in person is inconvenient.

Alternatives? If you don't already have the key on a business card, just pick up the phone (note: `--with-icao-spelling`). Talk about why using the same medium for getting fingerprint is not good. If you want to send an email, then it might be reasonable to use, say, twitter direct messages to boot strap a conversation. Both are **much** more secure than no check. How to sign (`--sign-key` and `--lsign-key`).

Talk about TOFU as an alternative. It's limitation. Nevertheless, in practice, probably good. New trust model (since v2.1.10, Dec. 2015). Checks identity / key consistency. Model used by `ssh`. No user support required. To enable, add the following to `gpg.conf`: `trust-model tofu+pgp`. Can also set `tofu-default-policy good`.

Talk about direct trust and trust always and what they are good for.

Talk about how to verify: a specific short key ID can be faked in just a few seconds. Even a long key ids are not immune to collisions. Talk about evil32 / scallion tool.

7.1 Key Discovery

How do you find a key? Traditionally, there are two ways: either via a business card or web site or by looking on a key server. The former is good, but inconvenient the latter is very, very bad. Key servers are not trusted. Anyone can forge a user id, etc. Talk about WKD and how it works. Given examples of how to deploy WKD.

Talk about keybase.io, Autocrypt, and pEp.

Chapter 8

GnuPG's Architecture

Talk about GnuPG's split architecture. Explain why it is important to separate gpg from gpg-agent, scdaemon, pinentry and dirmngr: Components in their own address spaces, which reduces impact of bugs (think heartbleed).

This is different from 1.4.

GPG is for low security—session encryption, encoding, etc. GPG Agent for security operations: password manager, private keys, etc. Similar similar to PC and smartcard. (In fact, possible to run gpg-agent as a different user id or on a different machine.)

Smartcard Daemon: Interacts with smartcards (directly or via PC/SC). Note: typically packaged separately as `scdaemon`.

Pinentry: for interacting with the user (not only passwords, but also questions). Started by gpg-agent. Talk about trusted windows and why it is important. Several different implementations provide tighter integration with a desktop environment. But, these come at a cost to security (much more complicated).

Pinentry fallback if there is no GUI. How to set `GPG_TTY` and why it is necessary. Talk about different pinentry configuration options and what they are good for. In particular, talk about how password caching works. Talk about `gpg-preset-passphraase` and `--loopback` related stuff in this context. Also talk about `--keep-tty` / `--keep-display`

Directory manager is really the network component. Interacts with key servers (HKP, ldap, http) (`--search-keys email@example.com`, `--recv-key keyid`). Certificate and CRL cache for gpgsm. Talk about different options. In particular, how to best configure `dirmngr` with `use-tor`, etc.

Give some details about the different sockets.

8.1 gpg-connect-agent

What it does (communicate with the different components). How to use it. Fact that it exposes a command line interface. Use help to figure out what to do.

Show how to script with gpg-connect-agent, e.g., shutting down a server.

8.2 signals

Talk about how e.g. SIGUSR1 can be used to cause gpg-agent to dump debugging information.

8.3 Assuan

Talk about Assuan. IPC protocol Pipe / socket based. Very simple, text-based interface. No interface definition language (IDL). Show example of a pinentry session calling `getpin`.

Can use `gpg-connect-agent` to connect to the running GPG Agent.

Assuan is a separate package from gpg. Anyone can use it.

8.4 Debugging

Due to the distributed nature of the architecture, it can be hard to figure out what went wrong (error messages become more generic as they are passed further along the stack).

`watchgnupg` helps. Tool for gathering log entries.

In `gpg-agent.conf`, add:

- `log-file socket:///home/USER/.gnupg/S.log`
- `debug-level basic` (or advanced or expert)

Run: `watchgnupg -\:\!-force /home/USER/.gnupg/S.log`

How to setup a test environment. (Talk about `GNUPGHOME` / `--homedir` and where the daemons live.)

8.5 configuration

Talk about gpg.conf, gpg-agent.conf, etc.

Talk about `gpgconf`.

Chapter 9

Good Practices and Tips

9.1 Refresh keys.

When you get a signed message, fetch the key.

Refresh keys regularly. Why? New preferences. Revocation certificates. WoT updates.

Note: Don't use `gpg --refresh-keys`. Install parcimonie. Uses tor. Random intervals between each key refresh reduces chance of targetted attacks and leaking who you are sending messages to.

9.2 Key Disclosure

What to do if you have to disclose the encryption key for a message? Don't disclose your private key! This allows decryption of all messages. Just disclose the session key. Show example of `--show-session-key`.

9.3 Backups

Don't backup the RNG's seed! Exclude `.gnupg/random_seed` from backups!

9.4 ssh

Keys Instead of Passwords. Using keys means password is not sent to server. Ever enter password for a different server? You've just disclosed your password!

OpenSSH stores private keys on hard drive. Keys are protected by a passphrase. Passphrase is cached by ssh agent.

GnuPG implements the ssh agent protocol. GnuPG can use keys stored on a smart card.

GnuPG's ssh agent: configuration:

Set `SSH_AUTH_SOCK` in `.bashrc`:

```
export SSH_AUTH_SOCK=$HOME/.gnupg/S.gpg-agent.ssh
```

Add `enable-ssh-support` to `~.gnupg/gpg-agent.conf`. Restart gpg agent. Add public key to `.ssh/authorized_keys` file. public key obtained by doing:

```
$ ssh-add -L
ssh-rsa AAAAB3NzaC1...zyt cardno:000603016636
```

9.5 Remote gpg-agent

gpg can use a remote gpg-agent. Running on another computer or as a different user.

- Create a new user, `gpg`
- On secure pc, add the following to `.gnupg/gpg-agent.conf`:

```
extra-socket /home/gpg/.gnupg/S.gpg-agent-remote
```

On insecure pc, run the following to forward the port:

```
$ ssh -f -o ExitOnForwardFailure=yes -o StreamLocalBindUnlink=yes \
> -L /home/neal/.gnupg/S.gpg-agent:/home/gpg/.gnupg/S.gpg-agent-remote
> gpg@localhost bash -c '{ while sleep 5; do echo NOP; done } | gpg-co
```

Requires at least version 6.7 OpenSSH, which supports forwarding Unix Domain Sockets.

Note: If forwarding fails, exit. If the socket to be forwarded already exists.

Forwards the file `.../S.gpg-agent` on the insecure host to the file `.../S.gpg-agent-remote` on the secure host.

Note: ssh won't expand tildes.

Loop keeps connection opened and port forwarded. (Could also use `autossh`.) Exits when gpg-agent exits.

Chapter 10

MUA Integration

This chapter contains guidelines on integrating GnuPG into a mail user agent (MUA). Other good sources of information on this topic are existing MUAs, in particular, KMail and Enigmail, which probably have the best GnuPG integration. This is not to say that our recommendations or what KMail and Enigmail implement are optimal. Far from it. A common criticism of GnuPG is how difficult it is to use. We acknowledge these criticisms, and we particularly welcome help in this area. Nevertheless, we suspect that some of the user interactions cannot be significantly simplified without compromising the security of the system, which has traditionally been designed to protect the user from an active adversary.

Most people do not have active adversaries. This is particularly true in democratic countries. People who live in these places primarily turn to a technology like GnuPG to protect their privacy, thwart phishing excursions, or fight mass surveillance. These users do not have the same security requirements as journalists, activists, or lawyers operating in regimes where civil rights are not respected, and a single unencrypted message can result in jail time, or worse.

Given these different classes of users, it is entirely reasonable to simplify some of the proposed interaction patterns for those who are only interested in protecting their privacy by using encryption opportunistically [26]. This is precisely what the Autocrypt project is trying to accomplish. Their hope is that trading protection from active adversaries for increased ease of use will result in greater adoption of encrypted email by people looking to protect their privacy, and fight mass surveillance, but don't want to be bothered with security issues [27].

Simplifying user interactions needs to be done carefully. People currently associate GnuPG and related tools as providing high levels of protection, and may assume that because these new interfaces use GnuPG that they provide the same level of protection. As such, we recommend the MUA make clear to users what level of protection the interface can offer. This could be done using a warning, but text that resembles an EULA is unlikely to be read [28]. Another approach to this problem is to ask the user to choose a profile that best matches their needs (i.e., their threat model), and then adjust defaults accordingly. This is the approach that the Tor Browser Bundle takes. This has the added benefit of causing users to think about risk assessment. The MUA need not support all of the profiles that it shows. Then, if the MUA does not support the user's threat model, the user should be warned.

In the GnuPG context, three profiles appear to be called for:

- **Very Strong Security**: Some users turn to GnuPG, because they fear targeted attacks from a nation state adversary including rubber-hose cryptoanalysis (i.e., the use of torture to recover passwords). These users should almost certainly use a security token, which the MUA should help them configure, HTML should be disabled, and all operations that could leak sensitive information should require explicit confirmation. The MUA should also help these users implement forward secrecy (by regularly rotating subkeys), provide a mechanism to automatically purge old emails, and disable indexing encrypted emails.

- **Strong Security**: Some users need protection from less sophisticated adversaries. For instance, lawyers worry that their communication with their clients may be spied on by criminal groups or corrupt government organizations. Although these users rely on encryption to protect sensitive communication, they also send and receive a lot of unencrypted email, and they don't want to be overly inconvenienced when processing those messages. Consequently, these users should have to confirm sending unencrypted mails when keys appear to be available, and using unverified keys should require confirmation.

- **Privacy Preserving**: Many people, especially those living in functioning democracies, aren't particularly worried about their safety. Instead, they turn to a tool like GnuPG, because they are concerned about their privacy, and mass surveillance. Other reasons include the need to occasionally send a password by email, and a desire for protection from drive-by phishing expeditions (although since few organizations currently sign their email, this is more wishful thinking than practical protection).

With few exceptions, the MUA should avoid interrupting these users with security questions. One exception is when the user follows up to an encrypted email, but the reply won't be sent in an encrypted manner. Since the sender encrypted the email, it might be for a good reason and, consistent with the do no harm principle, the user should not accidentally endanger her communication partner, or the subject of the mail.

This doesn't mean that the encryption should entirely disappear into the background. The MUA should still help the user understand what is going on, and allow the user to provide input, if desired. For instance, like a web browser, the MUA should indicate whether a message is secure. And, if the user clicks on the icon, she should get more information, and have the option to verify her communication partner's identity. In other words, security should largely be opt-in.

The trade off that these profiles make is straightforward: someone who requires more security is more sensitive to a mistake, and is more willing to interact with the system to ensure this security. For people who have lower security requirements, not only are these interactions annoying, they can actually hurt security elsewhere: showing dialog boxes that are simply clicked away results in habituation [29, 28].

Communication, of course, necessarily involves multiple parties. Thus, if a user with high security requirements communicates with a user with low security requirements, the casual user could accidentally compromise the careful user by forgetting to encrypt an email. Thus, consistent with the do-no-harm principle, it is important that even an implementation designed for users with low security requirements not be too lax.

10.1 Integration

There are two basic ways to add GnuPG support to a MUA: it can be added natively, or it can be added via a plug-in. KMail, and Claws are examples of MUAs that have native GnuPG support; Enigmail, GPGTools, and gpgol are examples of plug-ins.

One approach isn't necessarily better than the other. But, the development of plug-ins tends to be highly divorced from the actual development of the MUA with the practical result that the needs of the plug-in are often not sufficiently taken into account by the MUA developers. This has been a problem for the Enigmail developers, for instance.

One common problem is controlling how messages are rendered: the GnuPG support code needs a lot of control over this. This control is necessary to prevent mimicry attacks. For instance, it is necessary to not only show when a message is verified, but also prevent an attacker from crafting a message that appears to be verified. One way to accomplish this is to style not only the message, but also the chrome around the message.

The things that need to be added to a MUA for reasonable GnuPG support is not long: there needs to be a way to create a key, encrypt messages, verify messages, and do some basic key management. But, all of these things have numerous gotchas that can negatively impact both the user experience, and the security of the system. The point of this chapter is to point out these issues to avoid making developers—or worse, their users—rediscover these problems the hard way.

10.2 Key Creation

When a GnuPG-enabled MUA is started, it would seem logical to prompt the user to create or import a key if the user has not already done so. This behavior is reasonable if the user has explicitly enabled GnuPG support by installing a plug-in. However, if the MUA has native GnuPG support, and it is not certain that all users want to use GnuPG, it may be best to wait to avoid overwhelming the user during the initial setup.

If a key is not generated immediately, this doesn't mean that the GnuPG-related functionality should somehow be hidden or disabled. Even without a key, it is still possible to verify signatures, and show unsigned messages as being insecure. Then, if a user clicks on such a security notice, the MUA can explain why the message is considered insecure, and provide an option for the user to configure the GnuPG support. Similarly, it is reasonable to present an option to encrypt a message before a key has been created. If the user selects this option, and there is no key associated with the sending email address, then the MUA should show the key creation wizard. This significantly improves discoverability.

The key generation wizard should not only allow the user to generate a new key, but also provide an option to import an existing one. When the user enters or selects a user ID, the wizard should look for an existing key with that email address both in the appropriate WKD, and on any configured key servers. If there is a matching key, the wizard should ask the user if she wants to import the key or really create a new one. Importing the key might not be possible if the key is a fake, or if the user lost access to the key, e.g., by formatting the computer, or forgetting the key's passphrase. Both are unfortunately rather common for novice users.

When the key generation wizard starts, the user ID should default to the current identity. For instance, if the user has the email addresses `alice@posteo.de` and `alice@gnupg.net`, and clicks on encrypt while composing an email from `alice@gnupg.net`, the wizard should default to creating a key for `alice@gnupg.net`. If Alice selects a different identity, then the wizard should explain why the key won't be usable for the email she is currently composing.

If the user already has a key, but not one for the current identity, it is reasonable for the key creation wizard to offer to add the identity to the existing key. However, current thinking in the GnuPG project is that users require less training when there is a one-to-one mapping of keys and email addresses than when multiple user IDs are associated with a single key. For instance, if the MUA offers to add the user ID to an existing key, it becomes necessary to explain why this might be undesirable, e.g., most people probably want separate keys for their private, and their work email. And later, if the user retires her email address, it will become necessary to explain the difference between revoking the key and revoking a user ID. Of course, since many users do use keys with multiple user IDs, it is necessary for the MUA to support such keys, and explain their meaning when signing keys, for instance.

The key generation wizard should make key creation as easy as possible by prompting the user for as little information as reasonable. In particular, the user should *not* have the option to enter a comment; adding a comment is almost always inappropriate [18]. Likewise, key generation parameters should not be configurable. But, the user should be allowed to choose whether the key is published on the Internet. This requires an explanation, which can be made by simile: publishing a key on the Internet is like publishing a telephone number in a phone book, and no one is checking the submitted entries.

If it is deemed absolutely necessary that the user be able to tweak key parameters, then the options should be hidden unless the user explicitly enables some sort of expert mode. The reason is simple: for the most part changing these parameters doesn't actually improve the overall security. For instance, using a 2048-bit RSA key is currently considered sufficiently secure by multiple authorities [30]. If more security is really needed, then the user should start by improving their weakest defense, which is almost certainly their opsec and not the cryptography. Bruce Schneier, for instance, argues that the Snowden leaks provide strong evidence that the NSA has not broken strong cryptography. Instead, the NSA appears to get the information they want by compromising infrastructure and endpoints [21]. The easiest and probably most effective measure is to use a smartcard instead of storing the private key material on the computer.

There are also practical reasons for not using an overly large key. Perhaps the most important one is simply based on performance: it does not take twice as long to verify a signature generated with a 4096-bit RSA key than one generated with a 2048-bit RSA key, but about an order of magnitude longer. This performance penalty becomes particularly noticeable for 16,384-bit keys.

10.2.1 Revocation Certificate

After creating a key, the wizard should prompt the user to save the key's revocation certificate, or offer to print it out (or both!). For users with low security requirements, it is also reasonable to send the revocation certificate to the user in an email (along with an explanation of what a revocation certificate is, and how to publish it). This is the easiest way to make sure the revocation certificate is stored in multiple places, but it has the disadvantage that it gives anyone who can access the user's mail the power to revoke her key. This weakness is problematic, but it is not disastrous: that person would be able to perform a denial of service attack (other people would no longer be able to send encrypted messages to the user, and signatures generated by the key would no longer be considered valid), but could not assume the user's identity, or read encrypted messages. And, creating a new key is straightforward. So, the potential damage is limited, and for most users probably represents a net win given the benefits of being able to revoke a lost or inaccessible key.

10.3 Expiration

When GnuPG 2.1 creates a new key, the default is to set the key to expire in two years. Just because a key expires does not mean that the user needs a new key: the expiration is just an emergency brake if the user loses access to her key, and can't publish a revocation certificate. Consequently, the MUA should support extending a key's expiration date. This can be done when the MUA starts. But, since many users rarely restart their MUA, it may be better to check whenever the key is used.

If the key is about to expire (within, say, three months), the MUA should extend the expiration. Once the expiration is extended, the key needs to be uploaded to the key servers or otherwise distributed to the user's communication partners so that their OpenPGP implementation can take the change into account.

Since extending a key's expiry requires making a self-signature, the user will need to unlock the secret key. This interaction can be hidden by piggy backing the operation onto some other operation that requires the user to unlock the key.

For security sensitive users, it may make sense to ask the user if this is desired. For very high risk users, there should also be an option to rotate the keys.

10.4 Sending Mail

The mail composition window should have a toggle to "secure" or "encrypt" the current message. When active, this toggle should actually cause the message to be encrypted *and* signed. There should *not* be a separate toggle for signing the message. As explained previously in Section 4.4, most users assume that encrypting includes signing, and don't understand signing at all.

The button may have a menu that becomes visible after, for instance, a long press, which allows the user to select between "Encrypt and Sign", "Sign-only", "Encrypt-only" and "No protection." However this menu is activated, it should be reserved for advanced users, which justifies the poor discoverability of this feature: needing to only encrypt or only sign a message is relatively specialized, and these users can be expected to have had training; normal users should only have to choose between a secure, and an insecure option.

The Mailpile MUA always signs messages, even if they are not encrypted. To avoid confusing users who do not have an OpenPGP capable MUA, Mailpile uses inline signatures when possible, because, with the exception of one line, the signature shows up at the bottom of the message, and users have learned to ignore mumbo jumbo at the end of messages. Anecdotal evidence suggests that this approach doesn't impose any cognative load on users whose MUAs don't support OpenPGP. When an inline signature can't be used, Mailpile exports the signature as an ASCII-armored blob, adds a description explaining the purpose of the signature, and names the attachment `signature.asc.html`. The naming is essential: if a recipient open the attachment, she sees the explanation, and knows that she can ignore it. Anecdotal evidence suggests that this also significantly reduces the amount of confusion that signatures typically cause.

For users with high security requirements, it makes sense to always enable encryption by default, and then require that the user explicitly disable it if encryption is not desired. This avoids mistakenly sending a message unencrypted when it should have been encrypted. However, this default can be annoying for users who do not normally encrypt their mail.

As mentioned earlier, a MUA can deal with this dilemma by setting appropriate defaults for the user's threat model. But even for low security users, there are cases in which it is clear that encryption should be enabled by default. For instance, if the user is replying to an encrypted message, then encryption should be enabled. In fact, if the user tries to disable encryption, it is reasonable to show a warning of the form: "you are replying to an encrypted message, do you really want to disable encryption for your reply?" Similarly, if a recipient consistently sends encrypted mail, or there is a verified key available, then encryption should probably be turned on. Although it is appealing to encrypt whenever possible, encryption can sometimes decrease usability. This is particularly the case for users who process email on multiple devices, but only a subset of them are able to decrypt the messages.

An appropriate default can be more difficult to find when there are multiple recipients. For instance, when a user replies to an encrypted message, she might not have keys for all of the recipients. But, the application can help the user find the keys, and, in this case, finding appropriate keys is actually straightforward: due to the way that OpenPGP encrypts data, the long key ID of the sender and any recipients will normally be embedded in the message (specifically, in the PK-ESK packets). Unfortunately, the key IDs are subject to tampering, but since this requires a more determined adversary, they are almost certainly much more reliable than simply searching a key server for keys with a particular email address. It is also possible to try and find the key using WKD, which provides a basic verification check. Another reason to avoid key servers is that using a key found on a key server may cause more problems than it solves: the message may be encrypted, but because it is the wrong key, the intended recipient can't decrypt it. Making decryption unreliable is a sure way to discourage the use of encryption. Key discovery is covered in more detail in Section 10.6.1.

Sometimes mails include keys as attachments, or references to them. In such cases, the MUA should either import them automatically or provide a button to allow the user to import them. But, the keys should always be imported if they are already available locally: the keys might contain updates, such as new subkeys, an extended expiration, or a revocation certificate. This topic is discussed further in Section 10.6.1.

10.4.1 Encryption Keys

To make it clear whether there is a key for a particular recipient, the MUA should add a small icon, e.g., a padlock, next to each email address. As usual, to improve discoverability, and provide a reminder to encrypt, this should always be done, even if encryption for a draft has not yet been enabled. In that case, the padlock should also be crossed out. The coloring and the icon should vary according to the degree to which the key is verified. (It is important to not only change the color to support colorblind users.)

We recommend that the UI distinguish between the different degrees of verification. The web of trust provides three verification levels: a key can either by fully verified, marginally verified or not verified. (Note: for historical reasons, GnuPG uses the term "trusted" here instead of "verified." To reduce confusion in this document, we reserve the term trusted for when a key is not just verified to be controlled by the stated entity, but may act as an introducer. MUAs should do the same.) And, the TOFU trust model provides even finer grained verification levels. These distinctions are important for security conscious users, and, as a rule of thumb, marginally verified keys should **not** be shown as having the same level of security as fully verified keys. Instead, fully verified keys should be shown in, say, green, and partially verified keys should be shown in, say, yellow. If it is somehow desirable that marginally verified keys have the same security level as fully verified keys, then the user should explicitly set the `marginals-needed` option in her `gpg.conf` file to `1`. In the very least, the UI should distinguish between fully verified keys, and not fully verified keys, i.e., if the UI only shows two states, it should show marginally verified keys the same way it shows completely unverified keys.

If the TOFU trust model is enabled, the number of days on which a message has been encrypted to the key plus the number of days on which a message signed by the key has been verified should be shown next to the icon. This can be shown in a small bubble subscripting the icon, which is similar to what Twitter does for showing counts. For large numbers, it is reasonable to show approximate numbers (e.g., rounding `1132` to `1.1k`).

Showing these statistics is important to help users to detect mimicry attacks, which are often employed by phishers. For instance, if a bank normally signs their emails, then users hopefully become used to seeing the count slowly increase. Then, if they get an email that appears to be from their bank, but the count is `0`, they will hopefully become suspicious.

If the user hovers the mouse over the padlock icon or clicks on it, the MUA should show a short, tweet-length message explaining why the key is considered verified (or not). If the key is not fully verified, an option to start a key verification wizard should be provided. If there is a TOFU conflict, there should be an option to start a TOFU conflict resolution wizard. And, if there is no key associated with the email address, there should be an option to start a key discovery wizard. (The wizards are described in Section 10.6.)

10.4.2 BCC Recipients

When sending a mail, if there are any `bcc` recipients, the MUA should create a separate mail for each `bcc` recipient, and one for the rest. This avoids having the OpenPGP implementation leak the `bcc` recipients to the other recipients. Although it is possible to hide a recipient's key ID in a message by using a speculative key ID (e.g., using `gpg`'s `--throw-keyids` option), this still reveals to the recipients that the message was probably encrypted to other people. Using separate emails avoids this leak.

10.4.3 Saving Drafts

In general, when a draft—whether it has been marked to be encrypted *or* not—is saved on the IMAP server, it should be encrypted to the user. It should not be encrypted to any recipients; they should only be able to decrypt the final version.

It is important to encrypt all drafts even if they that have not been marked for encryption, because the user's intent is only known once the mail has been sent. It may be reasonable to relax this requirement in cases where it is clear that the user is only using the encryption for privacy purposes. But a safer way to avoid using the private key to decrypt the drafts is to *also* either save the session key or an unencrypted copy locally.

10.4.4 Sent Mails

When sending a mail, it is important to also encrypt the mail to the user. Given the near universal prevalence of a sent folder in MUAs, most users clearly expect to occasionally be able to later read the mails that they send. This can be done using `gpg` 's `encrypt-to` option, or, when encrypting an email, the sender can be specified explicitly.

10.4.5 Attaching Keys

To make it easier for a recipient to reply to a message in an encrypted manner, the MUA should provide an option to attach all public keys she would need to do so.

Receiving a key can be surprising to users who don't use or know about GnuPG. But, if you are encrypting, this is not a concern: you know the recipient's MUA understands OpenPGP messages. As such, in these cases, the keys can be attached automatically.

When attaching a key, it is reasonable to just include a minimal version of the key. In particular, it doesn't need to include any certifications, because once the recipient has the key, it is easy to get the rest of the data from a key server. A minimal key can be created by providing the option `--export-options export-minimal` when exporting a key using `gpg`.

The user's key should also always be specified in the OpenPGP header [31]. This is the case whether the mail is encrypted or not. This provides a strong hint to recipients that the user can work with OpenPGP messages.

10.5 Reading Mail

When the user opens an email message, it is necessary to identify if the message is encrypted or signed and to act accordingly. This is relatively straightforward, but does require a robust MIME parser to to handle all email. In particular, emails that have been transformed during transport can be problematic. The more challenging issue is making sure the user understands whether a message has been transferred securely. As a general rule of thumb, it is better to be conservative, and indicate that a message has been transferred insecurely than to incorrectly claim that a message has been transferred securely when that might not be the case. For instance, instead of attempting to interpret all possible structures, it is better to white list acceptable structures, and treat deviations as being insecure. Other issues include avoiding unnecessary passphrase prompts, and searching encrypted email.

10.5.1 Verifying Messages

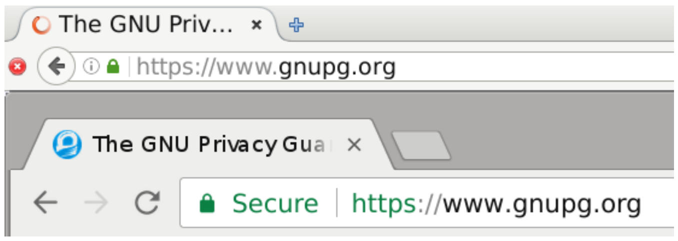


Figure 10.1: Padlock icons shown by Firefox and Chromium when a website is transferred securely.

When a user views an email, it is important to communicate whether the contents were transferred in a secure fashion. In web browsers, this type of information is usually shown using a small padlock icon in the address bar.

Firefox, for instance, shows a green padlock if it transferred the website in an encrypted manner, *and* it could authenticate the end-point. It uses a gray padlock with a yellow warning triangle if some—but not all—of the content was encrypted, and eavesdropping was possible, or if the website used a self-signed certificate. It uses a gray padlock with red strikethrough if a man-in-the-middle attack was possible. And it just shows a neutral, "more information" icon if TLS was not used at all [32]. There are two important issues with this scheme.

The first issue is that this scheme conflates encryption and authentication. Although it might be reasonable to demand that websites that use authentication also use encryption to be considered secure—it simplifies user training, and doesn't impose a significant deployment cost—this argument doesn't apply in an email setting. Consider, for instance, a company that wants to sign all of its outgoing emails to help mitigate phishing. In this scenario, encryption is more of a hindrance than a help: requiring encryption would mean that the company would have to somehow find the right encryption key for each of its correspondents. When only providing an authentication mechanism, not only are the customers' keys not required, the customers don't even need to have a key: they just need the ability to validate the signature.

The second problem is that a TLS connection that can't be authenticated is shown to be worse than a connection that is completely insecure. For instance, until the recent introduction of *Let's Encrypt*, website operators who wanted to offer an encrypted connection to their website, but didn't want to pay for a certificate could use a self-signed certificate. Although data protected by such certificates is not secure in the sense that the end point can't be authenticated without user intervention, such certificates enable encryption, which does protect users from passive surveillance. In other words, self-signed certificates provide more protection than nothing at all, but websites that use self-signed certificates are shown as being less secure than sites that use no protection at all! (Although encrypting is better than not encrypting, we nevertheless recommend that MUAs show encrypted and unsigned mails in the same way that they show unencrypted and unsigned mails to avoid confusing users.)

Happily, at least the Chrome browser does not make this distinction. And, like Chrome, we strongly recommend that whatever mechanism is used to show that a mail can't be authenticated be used for *both* unsigned mails, and mails with a signature that can't be verified. Specifically, we recommend considering an unencrypted and unsigned email to be the baseline, and that an email is never displayed in such a way that the user would consider it to be less secure than the baseline, unless there is strong evidence of an attack.

It is reasonable to show unverified messages, and unsigned messages in a neutral manner, and to show verified messages in a positive manner. However, it may also be reasonable to show unverified messages, and unsigned messages in a negative manner. This is how MS Outlook behaves when S/MIME is enabled. This has the added advantage that it may prompt the user to learn why the MUA showed the message as being unsafe.

The first step to checking whether a message is authentic is to check whether the signing key is verified according to some trust model, e.g., the web of trust. When verifying an email, another step is required: it is also necessary to make sure the key is controlled by the sender. This can be done by checking that the email address in the email's `From` header actually appears in one of the key's verified user IDs. This is necessary to prevent an attacker from reusing a message in a different context. For instance, assuming Romeo trusts his father, his father could write an email that appears to come from Juliet, but sign it with his own key. If the MUA doesn't check that the `From` header and the signer field agree, then the MUA would show Romeo that the message is verified. Unfortunately, some mailing lists rewrite the `From` header, which will cause this test to gratuitously fail. One reason for doing this is to improve DMARC compatibility.

Just checking that the sender matches a verified user ID is not actually enough to prevent all replay and mimicry attacks. It is also necessary to make sure the embedded timestamp is similar to (i.e., within a few hours of) the email's timestamp. If the timestamp in the email is years later than the one embedded in the signature, then the email may be part of an attempted replay attack. Similarly, it is possible to change the recipient. For instance, Juliet might send the following signed message to Paris: "Go away, I do not love you!" But, Paris, realizing that Romeo and Juliet are in love, and hoping to trick Romeo, might simply send a copy of the message to him with the `From` header set to Juliet. These types of attacks can be mitigated by also verifying the mail headers. The Memory Hole project was started to do exactly this [33]. Unfortunately, the standard isn't finished, and work on it appears to have stalled. Nevertheless, there is enough information to understand the intent, and several mail clients including Enigmail and Mailpile implement it.

Sometimes, a message may include multiple signatures. Any signatures from keys that match the email address in the `From` header should be used for verification purposes. Other keys may be listed when showing the verification details.

If the TOFU trust model is enabled, then the TOFU statistics should be shown as in the encryption case.

10.5.2 Multi-part Emails

Thanks to inline signatures, it is trivial to make a message that is only partially verifiable.

For simplicity's sake—we don't want to confuse the user—it is tempting to treat such messages as insecure like web browsers do. However, some companies, and some mailing lists automatically append a footer to all messages. This modification would change a message that is otherwise completely verifiable to one that contains a part that isn't signed. Thus, messages coming from these sources would never show up as secure.

A straightforward *technical* solution is to show each section individually. This can be done using a frame. The frame should be part of the MUAs chrome and not the message to avoid mimicry attacks. Further, each part should have an icon, e.g., a padlock, that shows information about the part's verification (the degree to which it is verified, and the key's TOFU statistics), and that, when clicked, shows a menu allowing the user to get more information, and find the key if it is missing, verify the key if it is present, or resolve a TOFU conflict, as appropriate.

To further distinguish between verified and unverified parts, a special background can be used. Ideally, the background should be unique for each user to further frustrate any attempt at a mimicry attack.

Unfortunately, this information rich technical solution may overwhelm many users. An alternative is to show a single icon at the top that shows the minimum security level of the individual parts. Since some corporations and mailing lists attach a small footer to all mails, this should be excluded from the calculation, but it should not be shown as verified.

The issues raised so far are manageable. Unfortunately, MIME makes things much more complicated: MIME can not only encode multi-part documents, but it can also encode rich content that logically consists of multiple MIME parts only some of which are signed, such as an HTML document and images that it references.

If a message includes at least one verified part, then the MUA should only show those parts that are verified, and warn the user that the message contained unverified content that is hidden. It is reasonable for the warning to include an option to show the unverified parts anyway. At that point the message should be displayed as insecure.

This suggestion conflicts with our earlier suggestion of showing unsigned messages in the same way as unverified messages. The best suggestion we have is to show a warning along the lines of "this message is unverified, show anyway" for unsigned messages. But, since most users will primarily deal with unsigned mail, this warning will very quickly get annoying, and lose its value. If security profiles are supported, this option should only be enabled for users who have very high security requirements.

10.5.3 Unencrypted Cache

The OpenPGP email workflow assumes that messages are stored on an untrusted server, and thus continue to need protection even after the mail has been delivered. Supporting this type of workflow is one of the primary reasons that OpenPGP doesn't provide forward secrecy. There are two major consequences of this workflow.

First, every time a message is accessed, it needs to be decrypted. This can lead to many passphrase prompts. These can be largely mitigated by increasing the amount of time `gpg-agent` caches passphrases, or by using a password manager. But, it is also annoying for smartcard users who need to basically always leave their smartcard inserted, which effectively nullifies a nice security property of smartcards: the user can observe operations, because they can only be done when the card is inserted.

Second, it is not possible to search encrypted mails. This is a major usability problem, particularly when the subject line is also obscured as it should be to avoid accidentally leaking the message's contents.

Both of these issues can be largely mitigated by caching the unencrypted version of each message locally. This assumes, of course, that the local device is secure. At a minimum, the user should have the mail stored on an encrypted partition.

10.6 Key Management

There are three main aspects to key management: key discovery, key verification, and key organization.

10.6.1 Key Discovery

The first requirement for encrypting or verifying a message is having the appropriate key. There are several ways to find the right key. Unfortunately, most of them make no guarantee that the key that is returned is the correct key. But some are significantly more difficult for an adversary to corrupt than others making them at least appropriate for opportunistic encryption.

Exchanging Fingerprints in Person

The most secure way to find a person's key is to get it from that person directly. If a physical meeting is possible, this can be done by exchanging fingerprints in person. At least in the business world, the cost of this exchange can be driven to zero: because exchanging business cards is a common practice in this world, adding your fingerprint to your business card makes securely exchange fingerprints a free byproduct of a well-established ritual.

Having a fingerprint on a business card is not quite enough to use it: it still needs to be entered into the system. The key discovery wizard can make this process easier by suggesting possible matches based on what the user has entered so far. (Possible matches can be found by querying a key server.)

We recommend having the user enter at least 64-bits worth of the fingerprint before enabling auto completion to ensure that the user checked a minimal amount of the fingerprint. For instance, it is possible to create a key with a specific 32-bit key ID in just a few *seconds* on modern desktop computers [34].

If the email address is known (and it is probably reasonable to first ask the user to specify a contact if this is not clear from the context), and there is at least one matching key, an alternative approach is to show a series of buttons with fragments of the matching fingerprints, and have the user select the matching fragments. This idea is illustrated below:

```
[ 8F17 ] [ 5200 ] [ 18A3 ]
[ 3DDA ] [ 6396 ] [ 8723 ]
[ AACB ] [ 6388 ] [ 0BAD ]

[ None of the above ]
```

The "none of the above" option is useful if the right key is not on the key servers, for whatever reason.

A more user-friendly technique could use a webcam and OCR to read in the fingerprint. From an implementation perspective, this is more demanding than scanning a QR code, for instance, but there are many fewer people who add a QR code containing their fingerprint to their business card than those who add their fingerprint. But, providing an option to display a public key using a QR code on screen can be helpful: someone could scan it.

Picking up the Phone

Exchanging keys in person requires that people actually meet face to face. This is often not practical. The next best alternative is to pick up the phone. This approach is appropriate for all but those people who have the highest security concerns—those whose threat model includes a real-time voice imitator. Although this has been technically feasible for years. It requires precise timing that only a nation-state adversary could afford.

Again, assuming the email address is known, the button grid can be used to facilitate transcription of the fingerprint.

Searching a Website

Calling someone is not always possible or desirable. In this case, it is sometimes possible to find the person's key on her website. The caveats are that even a relatively unsophisticated attacker can often own a website or spoof it, and because there hasn't traditionally been a standard place to publish keys, the MUA can't actually help the user find it.

In 2016, the GnuPG project published a new key discovery protocol called the web key directory (WKD). WKD automates, and hardens this key discovery process. The basic idea is that to find `romeo@posteo.de` 's key, Juliet looks for the key associated with `romeo@posteo.de` in a database on `posteo.de` [8]. This protocol relies on the security of TLS, and the mail provider. The mail provider can currently be held in check by periodically auditing the database, e.g., periodically fetching your own key via Tor and making sure it hasn't been replaced. Eventually, something like certificate transparency [35] could be added to catch abuse or detect things like national security letters. The reliance on TLS and its centralized infrastructure goes against the philosophy of OpenPGP, but it is acceptable for people whose threat model is limited to privacy violations, and phishing excursions.

Currently, the only commercial mail provider that supports WKD is Posteo, but, as of 2017, there are discussions underway with other mail providers to add support for this feature.

Searching Key Servers

A seemingly convenient way to find someone's key is to search for it using that person's email address on a public key server. Unfortunately, this method has very bad security properties: anyone can upload a key to a key server with any user ID. It is trivial to forge a user ID. In fact, in 2014, all known keys were cloned with identical short key IDs [34]. But, even if you are only interested in the privacy aspects of encryption, the key servers are a bad place to look for keys. Because many people forget their passphrase or forget to migrate their key to a new computer, the key servers are littered with seemingly valid keys that are practically unusable. Since someone searching the key servers doesn't know what key is correct, these people often get emails that they can't decrypt. This is annoying, and causes people to avoid encryption. Consequently, if a MUA decides to provide support for looking up keys by their user ID, we strongly advise adding a prominent warning about the possible problems. Further, if this approach must be used, it is better to encrypt to all matching keys. When the recipient replies, it is then possible to narrow down the set of potential keys based on the signature or the PK-ESK packets—assuming there was no man in the middle attack.

Note: these problems don't mean that key servers are completely useless. Far from it. The problem with key servers is that user IDs are not authoritative. But, if you have already verified someone's key, then key servers are the perfect place to get any updates (e.g., new signatures, revocation certificates, etc.), because cryptography can be used to determine whether the information really belongs to the key in question.

Exploiting Context and Hints

There are two main places where context can be used to discover potentially useful keys: a signed message indicates what key was used to sign it, and an encrypted message usually includes the key IDs of the sender and other recipients in the PK-ESK packets. Emails also sometimes include hints about the right keys to use. For instance, some people attach either their key to the emails that they send (pEp does this by default), or the keys of all recipients in order to make it easier for people to reply in a multi-party discuss. Another hint can sometimes be found among a mail's headers: the `OpenPGP` header allows the sender to advertise a key [31].

In theory, there is no reason to not import these keys. Simply importing a key will not cause it to be considered verified: whether a key is considered to be verified, is determined exclusively by the trust model, not whether it happens to be available locally. But, having what is probably the right key available locally is useful for opportunistic encryption. And, used in conjunction with the TOFU trust model, it is even possible to bootstrap some trust over time.

Unfortunately, in practice there are two important issues with harvesting keys.

The first issue is that automatically fetching keys via the network can be used as a back channel. A sophisticated attacker could create a new key for each message. When a user fetches the key, the attacker can potentially learn not only that the user opened the message, but also the user's IP address. This attack can be mitigated by routing this type of traffic via Tor (to do this, Tor must be installed and GnuPG configured to use it by adding `use-tor` to `dirmngr.conf`). Using Tor not only hides the user's IP address, but also requires the attacker to actually control the user's preferred key servers to observe the fetch. This is only feasible by an adversary with a lot of resources.

Even if automatically fetching keys is disabled, the MUA can still harvest this information, and save it in a local database. Then when the user explicitly searches for the key associated with an email address, say, the hints can be exploited.

The second issue is that GnuPG doesn't handle very large key rings (those with thousands of keys) very well. This manifests itself in two ways. It shows up as longer random access times: `gpg` does a linear scan of the key ring the first time it is accessed. Also, GnuPG's trust calculations are done on demand when `gpg` starts. These calculations can take minutes on large key rings. And, they are done whenever a new key or signature is imported, or a key expires or is revoked. When harvesting keys, this can happen very often. Happily, the trust calculations can be deferred by setting `no-auto-check-trustdb` in `gpg.conf` and then running `gpg --check-trustdb` periodically, e.g., from something like `cron`. But obviously, this means the trust model may not be completely up to date. However, the only long-term fix is to improve the way that keys are stored on disk.

Note: `gpg` can automatically fetch keys needed for verifying signatures by setting the `auto-key-retrieve` option in `gpg.conf`, and for encrypting messages by setting the `auto-key-locate` option. These options have the disadvantage that they can potentially block the `gpg` process for a relatively long time. Consequently, it is often more appropriate to attempt to fetch the key in the background. In the verification case, the message can be rerendered if the key becomes available. And, in the encryption case, a key should be located in the background when the recipient is added, not when the message is sent.

Taking Advantage of Trusted Introducers

Designating someone as a trusted introducer means that the user trusts that person to correctly verify others. Since friends of friends are likely to be friends as well, it makes sense to proactively fetch any keys that trusted introducers have signed.

`gpg` does not do this itself. And, unfortunately, the key servers do not provide a mechanism to find all keys signed by a particular key. But, since verification is usually mutual, it is possible to approximate this by fetching all keys that signed a trusted introducer's key. The MUA can do this periodically in the background.

10.6.2 Key Verification

Key verification is essential to the security of the system. Although people who are primarily interested in preserving their privacy will not spend much time on this task, it is essential that the key verification support is robust for those who depend on it for its security properties.

This requirement first means that it should be easy to start the key verification wizard in appropriate contexts. For instance, when the user adds a recipient to an email, as explained above, an icon should be displayed showing whether there is a key associated with the contact, and, if so, the degree to which the key is considered verified. Clicking on the icon should allow the user to verify the key.

When the key verification wizard is started, it should not just prompt the user to check the fingerprint, but actually guide the user through the different ways to obtain a fingerprint. For instance, the following is a bad idea:

```
Certify this key?

  8F17 7771 18A3 3DDA 9BA4  8E62 AACB 3243 6300 52D9

                                [ Ok ]  [ Cancel ]
```

Instead, the key verification wizard should ask the user how she wants to confirm the key: using a business card or other printout, or via phone. This approach educates the user without being patronizing: the user learns how to verify a fingerprint, and that it is not okay to just click verify without actually verifying the key.

To prevent the user from simply clicking okay without checking the fingerprint, we recommend requiring that the user enter at least part of the fingerprint. This can be done by using the buttons with the fingerprint fragments, as described above.

Ownertrust

It is strongly recommended that an option to set a key's `ownertrust` be well hidden relative to the key verification option. In fact, it should only be possible to set the `ownertrust` if the key in question is already fully verified (e.g., directly signed). Also, even though there are a few rare cases where it makes sense, it shouldn't be possible to set a key to be ultimately trusted if no secret key material is available.

When the `ownertrust` option is shown, it must be well explained that this option is not only about trusting the person, but also trusting how she verifies keys. For instance, I might trust my best friend when he introduces me to people in the physical world, but without understanding how he verifies keys (does he just click on yes to make the padlock green?), I probably should not set him as a trusted introducer. In practice, the latter is generally much more difficult for people to judge, because they don't understand the process very well themselves, and, given how hard it is to get people to exchange fingerprints, it is unlikely that we will ever convince them to discuss their security practices.

Publishing Signatures

The key verification wizard should provide an option to publish the signature. This should be accompanied by an explanation of what this means and why this is useful (people who trust you won't need to manually verify this fingerprint).

It is also reasonable to provide an option to make a trusted signature instead of a simple certification. Again, this requires an explanation. This option should probably only be hidden unless expert mode is enabled.

10.6.3 TOFU Conflict Resolution

Like the web of trust, TOFU is a trust model. The major difference between the two is that the web of trust provides strong guarantees, but requires a lot of upfront verification work whereas TOFU builds up trust slowly over time and is only secure in an asymptotic sense, but requires little user support. The `tofu+pgp` trust model should be the default for users with low security requirements. For backwards compatibility reasons, TOFU has not been made the default in GnuPG.

Normally, the user only needs to interact with the TOFU trust model to resolve conflicts—when multiple valid keys have the same email address. A conflict is a strong sign that a man-in-the-middle attack is underway. But, it can also just be because the user replaced a key that she could no longer access or revoke. The only way to resolve this is by asking the user to verify the key. (When creating a new key, a conflict can be avoided by either promptly revoking the old one or cross signing the two keys.)

When the user starts the conflict resolution wizard, the wizard should explain what a conflict is, show the conflicting keys and their statistics, and explain how to resolve the problem (ideally, the user should call the contact to verify the fingerprint). Because the user might not be able to resolve the conflict immediately, it is better to provide a resolve later option, which is the default, rather than have the user simply accept the key without validating it.

Note: just because a key has a lot of past usage does not mean that it is the right key: the man in the middle might just have failed to intercept the most recent message. Likewise, the new key is not necessarily the right one: the man in the middle might just have started the attack.

10.6.4 Address Book Integration

A key ring is effectively a backwards address book: instead of names being the primary keys, and OpenPGP keys being associated with contacts, a key ring reverses this. This unusual arrangement can cause novice users significant confusion. As such, it is better to avoid the key ring as much as possible, and instead directly integrate keys into the user's address book.

If the address book supports identities with multiple email addresses, then it should be possible to associate each email address with a different key. Also, it should be possible to force messages sent to a particular contact to be encrypted to multiple keys. This is useful in the case where an email address acts as an exploder.

It is also useful to keep track of users who appear to use GnuPG. A recent encrypted or signed email is the best indicator, but the presence of the OpenPGP mail header is also an excellent hint. The presence of a key with the user's email address is, however, not sufficient proof that the user can use GnuPG. This functionality can also be exposed as an option: "always encrypt to this user."

Chapter 11

Programming with GnuPG

`--batch`, `--status-fd` and `--command-fd`.

Writing tests: use `--faked-system-time`. (Talk about how it works.) `gpg-compose` for creating test data.

Talk about GPGME.

Chapter 12

Misc.

Topics are probably better integrated someplace else:

- gpgv
- `/etc/skel/.gnupg`
- keyring vs. keybox. Talk about kbxutil.
- What's a keygrip.
- More tools: encrypted mailing lists (schleuder), form encryption (Kuvert),

Bibliography

[1] Neal H. Walfield. GnuPG Stories: Jason Reich from BuzzFeed. https://youtu.be/oQvP9SXm-ek?t=1m46s, June 2017.

[2] Neal H. Walfield. GnuPG Stories: Micha 'Rysiek' Woniak from OCCRP. https://www.youtube.com/watch?v=6DqfWz-KHSI&feature=youtu.be&t=5m50s, June 2017.

[3] Neal H. Walfield. GnuPG Stories: Cindy Cohn, executive director of the EFF. https://www.youtube.com/watch?v=IdCiJMc3q80&feature=youtu.be&t=4m30s, June 2017.

[4] Wikipedia. Linux — wikipedia, the free encyclopedia, 2017. [Online; accessed 19-July-2017].

[5] Mike Gerwitz. A git horror story: Repository integrity with signed commits. https://mikegerwitz.com/papers/git-horror-story.html, February 2017.

[6] Werner Koch. First release. https://lists.gnupg.org/pipermail/gnupg-devel/1997-December/014131.html, December 1997.

[7] Wikipedia. Pretty good privacy — wikipedia, the free encyclopedia, 2017. [Online; accessed 19-July-2017].

[8] Werner Koch. OpenPGP Web Key Service. Internet-Draft draft-koch-openpgp-webkey-service-02, IETF Secretariat, October 2016. https://tools.ietf.org/id/draft-koch-openpgp-webkey-service-02.txt.

[9] Neal H. Walfield and Werner Koch. TOFU for OpenPGP. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, pages 2:1–2:6, New York, NY, USA, 2016. ACM.

[10] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. End-to-End Encrypted Messaging Protocols: An Overview. In Franco Bagnoli, Anna Satsiou, Ioannis Stavrakakis, Paolo Nesi, Giovanna Pacini, Yanina Welp, Thanassis Tiropanis, and Dominic DiFranzo, editors, *Third International Conference, INSCI 2016 - Internet Science*, volume 9934 of *Lecture Notes in Computer Science (LNCS)*, pages 244 – 254, Florence, Italy, September 2016. Springer.

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

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

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

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

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

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

[17] 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`.

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

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

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

[21] Bruce Schneier. NSA surveillance: A guide to staying secure. `https://www.theguardian.com/world/2013/sep/05/nsa-how-to-remain-secure-surveillance`, September 2013.

[22] Achim Pietig. Functional specification of the openpgp application on iso smart card operating systems. `https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.3.pdf`, June 2017.

[23] Jakob Ehrensvärd. Secure hardware vs. open source. `https://www.yubico.com/2016/05/secure-hardware-vs-open-source/`, May 2016. Last accessed: August 2, 2017.

[24] Wikipedia. Dual ec drbg — wikipedia, the free encyclopedia, 2017. [Online; accessed 2-August-2017].

[25] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[26] V. Dukhovni. Opportunistic Security: Some Protection Most of the Time. RFC 7435, RFC Editor, December 2014. `https://www.rfc-editor.org/rfc/rfc7435.txt`.

[27] Holger P. Krekel, Danial Kahn Gillmor, et al. Autocrypt level 1. `https://autocrypt.readthedocs.io/en/latest/`.

[28] Rainer Böhme and Stefan Köpsell. Trained to accept?: A field experiment on consent dialogs. In *Proceedings of the SIGCHI Conference on*

Human Factors in Computing Systems, CHI '10, pages 2403–2406, New York, NY, USA, 2010. ACM.

[29] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *Proceedings of the 2011 Workshop on New Security Paradigms Workshop*, NSPW '11, pages 67–82, New York, NY, USA, 2011. ACM.

[30] Damien Giry. Keylength - cryptographic key length recommendation. `https://www.keylength.com/`. Last accessed: July 25, 2017.

[31] Atom Smasher and Simon Josefsson. The "OpenPGP" mail and news header field. Internet-Draft draft-josefsson-openpgp-mailnews-header-07, IETF Secretariat, August 2014. `https://tools.ietf.org/html/draft-josefsson-openpgp-mailnews-header-07`.

[32] mozilla support. How do i tell if my connection to a website is secure? `https://support.mozilla.org/en-US/kb/how-do-i-tell-if-my-connection-is-secure`. Last accessed: July 23, 2017.

[33] Daniel Kahn Gillmor et al. Memory hole. `http://modernpgp.org/memoryhole/`,`https://github.com/ModernPGP/memoryhole`. Last accessed: July 23, 2017.

[34] Richard Klafter and Eric Swanson. Evil 32: Check your gpg fingerprints. `https://evil32.com/`, `https://www.defcon.org/html/defcon-22/dc-22-speakers.html#Klafter`, August 2014. Last accessed: July 28, 2017.

[35] Ben Laurie, Adan Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, RFC Editor, June 2013. `https://www.rfc-editor.org/rfc/rfc6962.txt`, `https://www.certificate-transparency.org/`.