Cryptography for Mail and Data

Lars Packschies

Difficulty

● ○ ○

Would you put confidential information on a postcard and send it to your friends, colleagues, or business partners? Well, no. But why would you put confidential information in an e-mail and send it around the world?

Cryptography not only makes your Internet communication more secure by giving you the opportunity to encrypt and/or sign messages, it also guarantees your own privacy. You may for example be aware of the fact that the European Union now regulated the retention of connection data by Internet Service Providers and Mobile Phone Companies for at least 6 months. Together with credit and bonus card data and all the information that lies around, this allows the generation of complete personal profiles not only from the primary data but also from data derived from data mining algorithms. They may already have gathered quite a lot of information about you and your habits but you now could start doing something about it.

Symmetric and asymmetric ciphers

The term *cryptography* originates from the Greek words *kryptós* for *hidden* and *gráphein* for *writing*. In general, we distinguish *symmetric* and *asymmetric* cryptographic *ciphers*. The terms symmetric or asymmetric relate to the structure of the *key*. To encrypt data, or a message respectively, you need the information of how to encrypt or decrypt data (the cipher) and a key, which is the secret parameter in the cipher. The knowledge of that key enables you to encrypt or decrypt information. The key can be used for a a longer period of time or you may use one key for every single message you send.

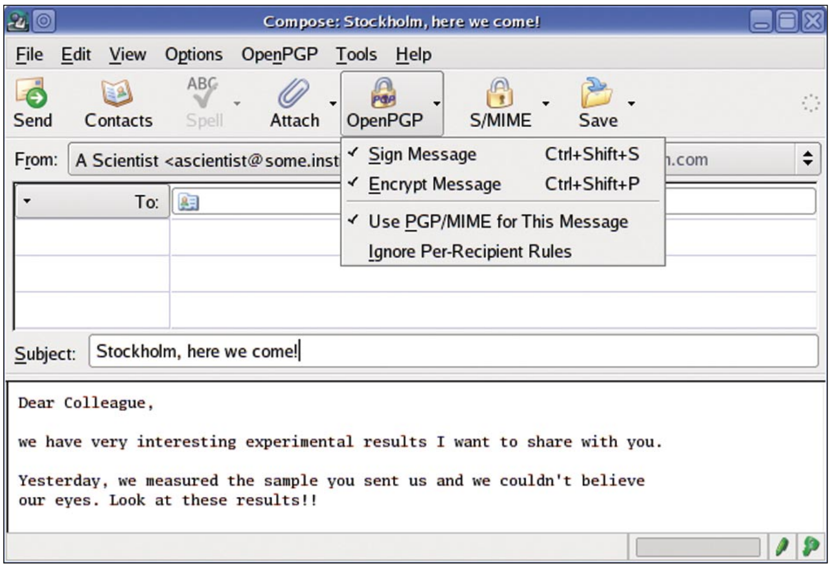
Symmetric cryptographic keys are characterized by the fact that the keys for encrypting and decrypting data are identical (or the key for encrypting and decrypting messages can be calculated from each other). In other words, sender and receiver of the message to be exchanged have to have the same key. And they have to exchange that key prior to sending messages around. This has always been the major

What you will learn...

- how you set up and use your keys Gnu PG,
- how you can encrypt data on the filesystem level.

What you should know...

- symmetric and asymmetric cryptography basics,
- algorithm basics.



From: A Scientist <ascientist@some.inst ….com

✓ Sign Message Ctrl+Shift+S
✓ Encrypt Message Ctrl+Shift+P
✓ Use PGP/MIME for This Message
Ignore Per-Recipient Rules

Subject: Stockholm, here we come!

```
Dear Colleague,

we have very interesting experimental results I want to share with you.

Yesterday, we measured the sample you sent us and we couldn't believe
our eyes. Look at these results!!
```

Figure 1. *Enigmail adds the OpenPGP button that allows you to sign and/or encrypt your messages*

drawback of symmetric methods: The so-called *key exchange problem*.

One of the first ciphers is called the *caesar cipher.* Julius Caesar used to encrypt messages by exchanging each letter of the original message by its third successor in the alphabet. A becomes D, B becomes E and Z becomes C. The algorithm is to exchange every clear text letter by the letter of a shifted alphabet, and the key is *3*; *Shift* the alphabet by 3.

The methods of substituting and transposing letters to generate more sophisticated ciphers have of course evolved over the years, some of which have involved the use of mechanical devices. One very prominent example is the ENIGMA used by the german troops in the second world war (there is a very good article on that machine and its cryptoanalysis in the *Wikipedia*). There were more than 200,000 of these machines in use, and every operator had to be equipped with a list of keys called codebooks every month. By the way: it was successfully cryptoanalysed (*cracked*, so to say) by a group of researchers around the Polish mathematician Marian Rajewski and Alan Turing in Bletchley Park, Milton Keynes, England, already in the mid 1930s. The public was informed about that in the 1970s (they called it *the ultra secret*).

Used with the help of modern computers, there is quite a number of symmetric ciphers available that are considered as *secure*, for example AES (Advanced Encryption Standard or Rijndael by Joan Daemen and Vincent Rijmen), 3DES (triple-DES, Data Encryption Standard, based on works by Horst Feistel) or IDEA (International Data Encryption Algorithm), only to name a few. All these modern symmetric ciphers have been developed roughly after the 1950s. Ciphers developed before that are more generally referred to as *classical*.

But it took until the 1970s before cryptographers solved that key exchange problem (by Whitfield Diffie, Martin Hellman and Ralph Merkle) and by developing the idea of asymmetric keys by Ron Rivest, Adi Shamir and Leonard Adleman in 1977, based on that key exchange idea.

Asymmetric cryptographic methods or algorithms use different keys for encrypting and decrypting data or messages. Both keys together are called the *Key Pair* (often referred simply to as *key* or *asymmetric key*). One of the two parts is always kept secret after the generation of the key pair. This is called the *private key,* while the corresponding counterpart is made available to the public, therefore called *public key.* One key is used to decrypt the message, and due to the underlying mathematical construct, only the other key can be used to reconstruct the original message. It is practically impossible to calculate the secret key from the public key (an vice versa). Moreover, it is also impossible to try to decrypt a message by trying every possible key. The latter attempt is called *brute force attack*. It would, by common knowledge, take a couple of billions of years.

Secret and Public Keys

The concept of secret and public keys generally allows two modes of operation: (1) Encryption/Decryption and (2) the generation and verification of electronic signatures.

Encryption/Decryption

Imagine two people, Alice and Bob. Alice generates a key pair (she does this only once, it can be reused) and makes the public key available to the public, and Bob can pick up that key. This public key can then be used by Bob to encrypt a message destined to Alice. However, only Alice's private key is able to decrypt Bobs message. Only the owner of that private key, Alice, can read it. Every person who has access to Alice's public key can write a message to her only she can read. When Alice intends to write a secret message to Bob, she could use a public key generated by Bob.

Signature

The second mode of operation uses the same keys of Alice in reverse order. Imagine Alice writing a message and encrypting it *with her private key.* Then, everybody with access to the matching public key can read the message after decrypting it. In that case, the reader can be sure that the message has been encrypted by Alice's private key, and, therefore, Alice must have written that message. Only Alice, by definition, is the only person to have that private key. We call this electronic signature.

Generally, there are two major asymmetric methods available today that you will have to do with and that are considered as *secure*:

RSA (Rivest, Shamir, Adleman, was patented) and ElGamal (by Taher ElGamal). Furthermore, there is the Digital Signature Algorithm (DSA).

PGP, OpenPGP, S/MIME

To put that together: RSA, ElGamal and DSA are asymmetric algorithms or ciphers. AES, 3DES or IDEA (IDEA was patented as well) are symmetric ciphers. You can simply use them to encrypt or decrypt or even electronically sign data. But to really be able to use these algorithms in real world applications, you need to know quite a lot of other things like how to handle data, what algorithms to use for the generation of key pairs, what to do when a message has to be encrypted or decrypted and so on.

To make it quite a bit more complex, there are not only asymmetric ciphers involved in the encryption of a message in modern applications. It takes a lot of time to encrypt large lumps of data using an asymmetric cipher. Much longer than it would take to use a symmetric cipher.

Therefore, for practical reasons, a symmetric session key is generated for each message to encrypt the data. After that, the symmetric key is encrypted using the actual key from the asymmetric key pair. You end up with two things: the symmetrically encrypted data block, and the asymmetrically encrypted symmetric key. The receiver then just uses the corresponding asymmetric key to dissect the symmetric key which in turn is then used to encrypt the data block.

After all these algorithms had been available to the public, the first application implementing these algorithms was PGP (Pretty Good Privacy) by Phil Zimmerman, released on an bulletin board in 1991. It became very popular but also more and more commercial. Not every PGP program version was released in source code. Furthermore, PGP was not allowed to be exported from the US in form of a computer program (there were some international Version (e.g. 5.0i). In fact, they were printed on paper and could be exported legally as a book. The code was then scanned an OCR'ed outside the US), and it contained patented algorithms. Since the community was therefore not always able to review the source code, those versions could not be completely trusted. There could for example have been backdoors or master key algorithms implemented without telling the public; Using cryptographic codes is a matter of trust. To avoid patent and license issues, the development of GnuPG (by Werner Koch) was started. GnuPG implements the so called OpenPGP Standard (RFC 2440, often referred to as PGP/MIME) that is based on PGP (the way Phil Zimmerman did it).

But it would be too easy if there was only one standard: there is also S/MIME (Secure MIME, RFC 2822). S/MIME uses (some) ciphers that are also used in OpenPGP, but both standards have different key and message formats and therefore are incompatible. Moreover, both standards use different trust models. While OpenPGP allows you to set up a large *web of trust* (we come back to that??) whereas S/MIME uses X.509 v3 (X.509 specifies, amongst other things, standard formats for public key certificates and a certification path validation algorithm – see *Wikipedia*) based certificates that are strongly hierarchical.

Hash algorithms

Cryptographic protocols make use of algorithms, that generate so-called *finger prints* or *hash values* of data. Such a hash is very short, you can not reconstruct the data from that hash value (otherwise this would be the best compression algorithm ever) and that hash value should be definite. Furthermore, it must be impossible (or at least nearly impossible) to generate two different documents with the same hash value. This is called *the generation of collisions*.

You may have seen hashes before when you tested the integrity of downloaded software packages (for example using md5sum or sha1sum). In cryptography, especially in electronic signatures, the MD5 and SHA1 algorithms are widely used. However, researchers have found ways to reduce the number of tests to find collisions by some numbers of magnitude. There is one example for MD5 where researchers have generated two different postscript files with the same MD5 hash value. The first is a letter of recommendation of Alice's Boss while the second document is an order of the roman emperor Gaius Caesar.

Thus, MD5 should be considered insecure, the same is true for SHA1. However, MD5 and SHA1 are still in use since they are part of the DSS algorithm. As long this is the case, MD5 and SHA1 will still be used in the program GnuPG, for example. There are better algorithms implemented in GnuPG, but SHA256 for instance uses RSA and not DSS keys. Unfortunately, one seems to have to live with this until an official NIST standard allows to circumvent that problem. However, it's possible to setup GnuPG keys so that they avoid the use of MD5. SHA-1 instead is obligatory with the OpenPGP standard, but one can reduce the probability of use by changing priorities for different hash algorithms. We come back to this later.

Key Generation

GnuPG may already be installed on your Linux box. Try `gpg --version`. If GnuPG is available, you should get the version number and the cryptographic and compression algorithms implemented in the actual version of GnuPG (shortened)

```
......> gpg (GnuPG) 1.4.2.2
[..]
Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E,
 RSA-S, ELG-E, DSA
Cipher: 3DES, CAST5, BLOWFISH,
 AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160,
SHA256, SHA384, SHA512
Compression:
 Uncompressed, ZIP, ZLIB, BZIP2
```

You are now ready for the generation of you first GnuPG keypair. To start the key generation process type

Table 1. *The codes list*

Code	Algorithm
Symmetric Ciphers	
S1	IDEA
S2	3DES
S3	CAST5
S4	BLOWFISH
S7	AES128
S8	AES192
S9	AES256
S10	TWOFISH
Hash Algorithms	
H1	MD5
H2	SHA1
H3	RipeMD160
H8	SHA256
H9	SHA384
H10	SHA512
Compression Algorithms	
Z1	ZIP
Z2	ZLIB
Z3	BZIP2

```
...........> gpg --gen-key
Please select
what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
Your selection?
```

Use the default here. The DSA keypair (used for signatures) will have 1024 bits in length, but you can change the key size of the ElGamal keypair. Normally, 2048 is enough. At some point it does not make sense to make the key longer and longer, because it's easier to torture you get the private key than to try to crack it. Unfortunately, the user is the weakest link of the chain.

```
DSA keypair will have 1024 bits.
ELG-E keys may be
between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

So just press *Enter*.

```
Requested keysize is 2048 bits
Please specify how long the key
should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

Normally you would enter 0 here. If you want to change the key every year, feel free to change that. But if you use so-called keyservers for the distribution of your key(s), you are accumulating expired keys on that servers. You can not delete keys from servers; You can only revoke them.

The next step then is to put some personal information into that key if you want. If you are about to take part in a web of trust and let others *sign* your public key and thereby stating they trust you, it makes sense to put in your e-mail address and your real name. Generally, you are free to put in there whatever you want.

```
You need a user ID to identify your
 key;
 the software constructs the user ID
from the Real Name,
 Comment and Email Address in this
 form:
    "Heinrich Heine (Der Dichter)
 <heinrichh@duesseldorf.de>"
Real name: Alice C
mail address: alice@example.com
Comment:
You selected this USER-ID:
    "Alice C <alice@example.com>"
Change (N)ame, (C)omment,
 (E)mail or (O)kay/(Q)uit?
```

Type (O). Now you enter a passphrase or *mantra*, as it is called. It should be as long as possible but you should be able to memorize it. 30-40 characters should be o.k., but do not use words from a dictionary or phrases out of books if possible. The mantra is the last bastion between the secret key and the world outside, so take a good one. If you want to write it down, place the piece of paper in a safe. You have to put in the mantra twice before the key generation process starts. GnuPG tells you that it may be a good idea to move the mouse around or do some other actions with the keyboard and so on. GnuPG needs random numbers to generate the key. The quality of these numbers is crucial. Modern Linux systems use random number generators that are suitable for your purpose.

This finishes the key generation process. GnuPG shows a summary of the key's features and the key identification information, for example:

```
pub   1024D/E7318B79 2006-03-17
      Key fingerprint
 = 6DB6 3657 EE80 E74D 164B
 C978 6500 F1EF E731 8B79
uid Alice C <alice@example.com>
sub   2048g/2B381D4B 2006-03-17
```

The line starting with pub 1024D gives you information about the primary key of length 1024 bit (DSA keys always are 1024 bits long), it's a DSA key (marked D) and it has the *key-ID* E7318B79. This number will identify your key on the world's key servers.

The next line shows the fingerprint of your key. When you let your key signed by other users, this fingerprint is used for identification (note that the key ID resembles the last four Bytes of the fingerprint). The line with sub 2048g tells you the subkey is an ElGamal (g) of length 2048 bit. The whole construct, which can hold additional subkeys or user identities (e.g. e-mail address information etc.), will always be identified as *key-ID E7318B79*.

Generating a key revocation certificate

It is really important to generate a so-called revocation certificate as the next step. It allows you to revoke your key, which means to put a tag on it like *expired* or *do not use it any more*. If your key gets compromised somehow or stolen, revoking is the only way to tell the world that this key should not be used again. But you have to be careful with this certificate. If this certificate gets stolen, the thief can revoke your key, upload it to a keyserver and make it unusable for you. And he does not even need your private key to do this, and he does not need your mantra either. And you can not remove this revocation certificate from you key once it is uploaded and distributed. To generate your personal revocation certificate, issue the command:

```
...> gpg --gen-revoke <your key-ID>
```

and give the information requested. Normally, you will generate a revocation certificate that revokes the key with no specific reason given, so *the key is not used anymore* will be o.k. After entering your mantra, GnuPG writes the certificate to stdout. The best thing is to write that down on a piece of paper and place it in a safe. If you want to print it, be aware of the fact that this document may run through print servers that may store data. You can write the certificate to a disk and place it in the safe as well, but disks may lose the data over the years.

In case that you have problems with your key (you lost it, or it got stolen or you just don't want to use it anymore), just import the certificate into your public key chain and upload it to a key server. There is more about im- and exporting keys below. The revocation certificate can just be handled like a public keyfile (but do not do this now).

```
 > gpg --import
<rev_certificate_filename>
```

Keyservers

Now your key is ready to use. The public part of that key can be exported into a file to be passed around to your friends or can be exported to international key servers. It is however strongly recommended not to upload your public key unless you have some working experience with your new key pair. To upload the public key issue the following command:

```
 > gpg --send-keys <key-ID>
```

To import a key from a key server use:

```
 > gpg --recv-keys <key-ID>
```

It may be necessary to specify a key server. Werner Koch recommends to use so-called SKS key servers since they can cope with all the information a keyfile can contain. A key server can be specified with the option --keyserver. In Poland for example you can use *sks.keyserver.penguin.pl*, in Germany there is *sks.keyserver.penguin.de*. Keyservers exchange their information, so keys are distributed automatically.

Key chains

The secret and public key rings or key chains can be found in the directory *~/.gnupg*. Make sure no other users can access the files in this directory.

Importing and exporting keys

To export your public key into a file named *mykey.txt* issue the command:

```
 > gpg --export --armor
     <your key-ID> > mykey.txt
```

The option `--armor` makes sure the key file is human readable. The key-ID may be replaced by any user information stored in the key, like your name or e-mail address.

To import the key of another user just take the file you got and import it into your public keyring. Here it is done with a key Alice got from Bob (the file *bobpublic.txt* contains the key):

```
 > gpg --import bobpublic.txt
gpg: key 20ACB216:
 public key "Bob B <bob@example.com>"
 imported
gpg: Total number processed: 1
gpg:       imported: 1
```

Editing, trusting and signing keys

There is only one tiny flaw: Alice should not just use Bob's key, but to express her trust first. To do so, GnuPG has a key editor. It is started with `gpg --edit-key <key-ID>`. Again, the `key-ID` may be replaced for example with the Name *Bob* or his e-mail-address.

At the prompt, you get an overview of the editor commands with `help`. For trusting Bobs key, Alice starts the editor and opens Bobs key. The key information is listed and contains unknown trust and validity levels:

```
pub  1024D/20ACB216
created: 2006-03-17
expires:  never     usage: CS
                    trust: unknown
                    validity: unknown
sub  2048g/6B99CC08
created: 2006-03-17
expires:  never     usage: E
[ unknown] (1). Bob B <bob@example.com>
```

It is really important to check whether this key really belongs to the person Alice thinks of as Bob. And, one step further, if Bob is really the person Alice thinks he is. It may as well be, that a third person, lets traditionally name him Mallory, who poses as Bob, offers Alice a wrong key. If Alice now would trust that key and encrypt messages for Bob, then Mallory instead of Bob could read these messages. To avoid this attack, Alice could ask Bob to show his `ID-card` and to hand over the key personally or she could check the fingerprint and ask Bob if the fingerprint is o.k.:

```
 > gpg --fingerprint bob
[..]
 Key fingerprint
 = 6871 3E47 AEEE 7424 10EF
 B544 3EC0 383B 20AC B216
```

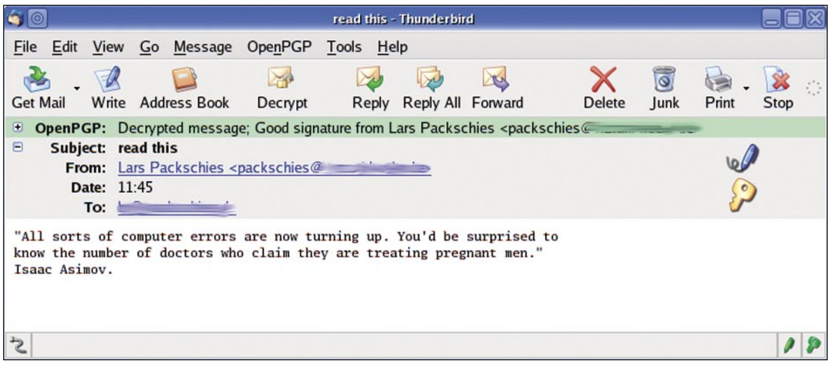
When she has clarified the identity of Bob and his key, she issues the command `trust`.

Please decide how far you trust this user to correctly verify other users' keys (by looking at passports, checking fingerprints from different sources, etc.).

```
  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
```

GnuPG expects Alice to decide how she rates Bobs experience with the handling of keys and if she thinks he is trustworthy personally. The value 5 is reserved for personal keys, not for those of other users. Alice trusts him fully.

```
pub  1024D/20ACB216
created: 2006-03-17
expires: never       usage: CS
                    trust: full
                    validity: unknown
sub  2048g/6B99CC08
created: 2006-03-17
expires: never       usage: E
[ unknown] (1). Bob B <bob@example.com>
Please note that the shown key validity
 is not necessarily correct
unless you restart the program.
```



OpenPGP: Decrypted message; Good signature from Lars Packschies <packschies[redacted]

Subject: read this
From: Lars Packschies <packschies@[redacted]
Date: 11:45
To: [redacted]

```
"All sorts of computer errors are now turning up. You'd be surprised to
know the number of doctors who claim they are treating pregnant men."
Isaac Asimov.
```

Figure 2. *The green line shows the signature status. The message was encrypted and signed, as shown by the key and pen icons on the right hand side. You can click on these to get additional information about the keys used*

Still, the key is not valid. To make it a valid, Alice has the opportunity to sign it with her own private key. This can be done within the key editor using the command `sign`. The key can then be exported into a file (see above) and sent back to Bob who can then import it into his public key chain. The signing of other user's keys is used to set up trust networks better known as the *web of trust*. If Alice does not want to hand over the signed key to Bob and just wants it to be valid in her own keychain, she can just sign it locally using the command `lsign`.

```
 > lsign pub  1024D/20ACB216
created: 2006-03-17
expires: never       usage: CS
                    trust: full
                    validity: unknown
 Primary key fingerprint:
 6871 3E47 AEEE 7424 10EF
 B544 3EC0 383B 20AC B216
  Bob B <bob@example.com>
Are you sure that you want to sign
 this key with your
key "Alice C <alice@example.com>"
 (E7318B79)
The signature will be marked
 as non-exportable.
Really sign? (Y/N)
```

The last information is due to the local signature. Alice types `Y` to sign it and she has to type in her mantra because her private key has to be unlocked.

```
You need a passphrase
to unlock the secret key for
user: "Alice C <alice@example.com>"
1024-bit DSA key,
 ID E7318B79, created 2006-03-17
Enter passphrase:
```

After entering the correct passphrase or mantra, Bobs key is valid for Alice. It may be that the key editor has to be restarted (use quit) to update the internal trust database of GnuPG.

```
pub  1024D/20ACB216
created: 2006-03-17
expires: never       usage: CS
                    trust: full
                    validity: full
sub  2048g/6B99CC08  created:
 2006-03-17
expires: never       usage: E
[  full  ] (1). Bob B <bob@example.com>
```

If Bob imports and signs Alice's key using a scheme as above (maybe with a non-local signature), they can start sending secret messages to each other. In general, sending secret messages is just writing the message and encrypting it with the receiver's public key. If you use GnuPG-enabled Mail clients (like Mozilla Thunderbird with the Enigmail plugin), the Mail client does this for you. But first, we will use the command line interface to GnuPG.

The Web of Trust

Signing and trusting other user's keys builds up a web of trust. Imagine, you want to write a secret message to a recipient, and you received his key from a public keyserver. You have never met him personally and you want to know, if the key really belongs to him. You have not gone through this process of checking the key fingerprint and sending test mails etc., but someone else may have done so. And if you really trust that person, the *unknown* key is automatically valid for you.

The web of trust has some easy rules. You can tune these rules a little, but generally speaking they are like this. A key is valid for you, if:

- you signed it or,
- it was signed by a key you trust fully or,
- it was signed by three keys you trust marginally,
- and the path between your key and the recipient's key is not longer than five steps.

Editing your key preferences

As stated above, you can set up your key in a way that the use of special algorithms like SHA-1 or MD5 are avoided, or at least other algorithms are more highly prioritized: if you want to use blowfish as your preferred symmetric cipher, and switch off DES, this is also possible.

These settings are also made using the key editor. Alice, for example, would fire up the editor using the command:

```
 > gpg --edit-key alice
Secret key is available.
pub  1024D/E7318B79
created: 2006-03-17
expires: never       usage: CS
                    trust: ultimate
                    validity: ultimate
sub  2048g/2B381D4B
created: 2006-03-17
expires: never       usage: E
[ultimate] (1).
Alice C <alice@example.com>
```

Which algorithms are used in this key can be displayed using the editor command `showpref` or `pref`. The former displays the key settings in a more verbose way, the latter replaces algorithms by their code. Alice's key has the following setup:

```
Command> showpref
pub  1024D/E7318B79
created: 2006-03-17
expires: never       usage: CS
                    trust: ultimate
                    validity: ultimate
[ultimate] (1).
Alice C <alice@example.com>
     Cipher: AES256, AES192,
 AES, CAST5, 3DES
     Digest: SHA1, RIPEMD160,
 SHA256, MD5
   Compression: ZLIB, BZIP2, ZIP,
 Uncompressed
  Features: MDC, Keyserver no-modify
```

You can easily see that SHA1 is the first hash algorithm in the *Digest* line, but you cannot set preferences naming the algorithms, you have to replace them by their identification code. The command `pref` lists the code line of the actual key:

```
Command> pref
pub  1024D/E7318B79
created: 2006-03-17
expires: never     usage: CS
                    trust: ultimate
                    validity: ultimate
[ultimate] (1).
Alice C <alice@example.com>
     S9 S8 S7 S3 S2 H2 H3
 H8 H1 Z2 Z3 Z1
 [mdc] [no-ks-modify]
```

S names the symmetric cipher algorithms, *H* are hash algorithms and *Z* names compression algorithms.

To set preferences, you can use the command `setpref`. This command uses a line of codes as input. If Alice wants to get rid of MD5, but wants to keep the other settings unchanged, she would copy and paste the code line from the `pref`-output above but would just leave away *H1* and put *H2* to the end of the hash algorithms.

```
Command> setpref S9 S8 S7 S3 S2
H3 H8 H2 Z2 Z3 Z1
Set preference list to:
  Cipher: AES256, AES192,
 AES, CAST5, 3DES
 Digest: RIPEMD160, SHA256, SHA1
  Compression: ZLIB, BZIP2, ZIP,
 Uncompressed
  Features: MDC, Keyserver no-modify
Really update the preferences? (Y/N)
```

Say `Yes` here:

```
You need a passphrase to unlock
 the secret key for user:
"Alice C <alice@example.com>"
1024-bit DSA key,
 ID E7318B79, created 2006-03-17
Enter passphrase:
```

After you entered the correct passphrase, the key attributes are updated. The command `showpref` shows the result:

```
Command> showpref
pub  1024D/E7318B79
created: 2006-03-17
expires: never       usage: CS
                    trust: ultimate
                    validity: ultimate
[ultimate] (1).
Alice C <alice@example.com>
     Cipher: AES256, AES192,
AES, CAST5, 3DES
    Digest: RIPEMD160,
SHA256, SHA1
     Compression: ZLIB, BZIP2, ZIP,
 Uncompressed
   Features: MDC, Keyserver no-modify
```

You can see that MD5 is switched off, and SHA1 is set to the end of the line, but you can't switch it off completely. This shows the user of Alice's key, that she prefers to use RIPEMD160 or SHA256 over SHA1. This already avoids the use of SHA1 in the most cases.

The settings of preferences can also important when you happen to import a PGP key into GnuPG or vice versa. To export a GnuPG key to use it with PGP for example (note that you can not use ElGamal keys with PGP), the preference settings have to be set to *S9 S8 S7 S3 S2 S10 H2 H3 Z1 Z0.*

Note: some versions of GnuPG use the command `updpref` to activate the settings made with `setpref`. Have a look at the editor `help` command output. To end the session, leave the editor with `quit`.

Encrypting and decrypting data

Imagine that Alice wants to write a message to Bob containing confidential information. She can write that in a file (*secret.txt*) and encrypt it using the command:

```
.  > gpg --recipient bob
 --encrypt --armor secret.txt
```

This generates a file *secret.txt.asc.* Not even Alice can now decrypt that file again, but she still has the original. However it is possible to generate an encrypted file that can be decrypted by two or more users. It has then to be encrypted using two or more recipients. Alice could as well do this:

```
 > gpg --recipient bob
 --encrypt --recipient alice --armor
                     secret.txt
```

Or, in short form,

```
 > gpg -r bob -r alice -e -a secret.txt
```

What happens there, is that the original message is encrypted with a session key and this session key is then encrypted, one at a time, with both public keys of Alice and Bob. All information together is then stored in the file *secret.txt.asc*.

Alice can send this file to Bob who is then able to decrypt the message with the decrypt option. His private key is then automatically used but Bob has to enter his mantra to unlock it.

```
 > gpg --decrypt secret.txt.asc
You need a passphrase to unlock
 the secret key for
user: "Bob B <bob@example.com>"
2048-bit ELG-E key, ID 6B99CC08,
 created 2006-03-17
(main key ID 20ACB216)
Enter passphrase:
```

Bob enters his passphrase here.

```
gpg: encrypted with 2048-bit ELG-E key,
 ID 2B381D4B, created 2006-03-17
  "Alice C <alice@example.com>"
gpg: encrypted with 2048-bit ELG-E key,
 ID 6B99CC08, created 2006-03-17
  "Bob B <bob@example.com>"
Hi Bob, come to the willow tree tonight
 at 8. we have to talk, Alice.
```

In this special case the last line is the original message of Alice.

Additionally, you can use GnuPG to encrypt data using symmetric ciphers. GnuPG uses the option *conventional* to do this. You can choose the cipher algorithm as well. To encrypt the file `mail.tgz` using AES256, just type

```
 > gpg --cipher-algo aes256
 -c mail.tgz
Enter passphrase:
Repeat passphrase:
```

You have to type the passphrase twice to avoid typos. If you do not provide an output filename with the `-o` option, GnuPG uses the input filename with the *appendix .gpg*. To decrypt the data, just type

```
 > gpg -o mail2.tgz mail.tgz.gpg
gpg: AES256 encrypted data
Enter passphrase:
```

The file *mail2.tgz* contains the original data.

Signature and signature validation

Bob can read the message now and he knows for sure that the message is meant to be *for him* (it has been encrypted with his public key) but he cannot be sure that it has been written and sent *from Alice* since the message has no signature attached. To do this, Alice can encrypt the message with an additional sign option. Here Alice uses the `--sign` option. It puts the signature and the signed text in one file named *secret.txt.asc*. Assumed that Bob trusted and signed Alice's key, he will get the following output from `gpg --decrypt secret.txt.asc`:

```
2048-bit ELG-E key,
ID 6B99CC08,
created 2006-03-17
(main key ID 20ACB216)
gpg: encrypted with 2048-bit ELG-E key,
 ID 2B381D4B, created 2006-03-17
  "Alice C <alice@example.com>"
gpg: encrypted with 2048-bit ELG-E key,
 ID 6B99CC08, created 2006-03-17
  "Bob B <bob@example.com>"
Hi Bob, come to the willow
tree tonight at 8. we have to talk,
 Alice.
gpg: Signature made
Fri 17 Mar 2006 04:05:26 PM CET
 using DSA key ID E7318B79
gpg: Good signature from
 "Alice C <alice@example.com>"
```

Now, Bob can be sure that Alice wrote the message because the signature could be verified.

Encrypt your mails: Thunderbird and Enigmail

Using GnuPG as described above can be annoying, especially when you are willing to use cryptographic functions like signing or encrypting your mail messages on a regular basis. In that case every message has to be saved to the file system, processed by GnuPG and then reopened in your mail client software to be sent.

To make life easier and more comfortable, nearly all mail user agents (mail client programs) have cryptographic functions implemented or give access to cryptography programs by special plugins, e.g. KMail, Mutt, Pine, Sylpheed, Emacs and Balsa, only to name a few.

One of the most powerful and reliable combinations is Thunderbird (the standalone mail client from the Mozilla project) together with the GnuPG plugin Enigmail. Enigmail adds an *OpenPGP* menu to your mail client. It is available for Linux, Mac OS X and Windows, you have to have GnuPG installed. You find GnuPG for your platform on *http://www.gnupg.org/download*. If you want to use GnuPG with Windows, have a look at the GnuPG.README.Windows file in your GnuPG Start menu entry; you can access gpg from the Windows command line or using the Windows Privacy Tray WinPT.

Enigmail can be downloaded from this web *http://enigmail.mozdev.org/*. To install the plugin, go to the *Tools* menu and select *Extensions* and then *Install*. Point the file browser to the Enigmail plugin file you just downloaded and select *Install*. Enigmail will be available after a restart of Thunderbird.

Enigmail has to be activated for every e-mail Identity you want to use. This is done in the *Edit* Menu, *Account settings* (*Tools* Menu in Windows). You have to check the box *Enable OpenPGP support (Enigmail) with this* identity. This window allows you to set your default Key ID manually if the Enigmail plugin cannot derive the ID from your e-mail address. The *Advanced* button opens the *OpenPGP Preferences* dialog, which can be accessed from the *OpenPGP* menu (entry *Preferences*) as well. You may have to enter the path to the gpg binary in the *Basic* tab if it was not set automatically. Furthermore, it's important to check if *Encrypt to self* is set in the *Sending* tab, otherwise you would not be able to read mail you sent encrypted. Another setting you should check is *Always use PGP/MIME* in the *PGP/MIME* tab. If PGP/MIME is not activated, Enigmail uses the so called inline PGP format where attachments are not encrypted.

To encrpyt or sign mail, use the OpenPGP button in the compose window. You can select *Sign Message, Encrypt Message* or both. If you want sign a message, Enigmail pops up the passphrase entry dialog to get access to your private key. GnuPG then processes the data before it is sent by your mail client. If you want to encrypt a message, GnuPG has to know the public key of the recipient.

When you receive a message that is encrypted to you, you are prompted to enter you passphrase. Thunderbird also tests the signature if provided and informs you if it is valid.

Encrypted File Containers and Filesystems

Cryptography is not only about GnuPG and encrypting files or messages. Beside many other things you can do with it like securing your Internet communication (SSH, SCP), securing mail and web servers etc. (not shown here), it is fairly easy to encrypt file containers and file systems under Linux as well. LoopAES and DM-Crypt are shortly introduced here, but there are more, of course, and it is possible to do similar things under other operating systems. Here are two examples.

Encrypt a partition with LoopAES

LoopAES uses the Linux cryptography enabled loopback device to set up a file system within a container or a partition as a device. I want to show a quick and easy scenario where you use a free disk partiton (here it is /dev/sdc3) to set up a file system within a loopback encrypted device. This file system is going to be en- and de-crypted on the fly, but you will need a key to get access to it. This actual key will be stored in a symmetrically encrypted keyfile. This sounds a bit complicated, but you will see how it works as we go through that example.

It may possibly not work on all systems, but it was successfully tested on Fedora Core 4. Some systems need a patched version of the loopback device. Some hints about this can be found on *http://loop-aes.sourceforge.net/*.

Firstly, choose a partition. It may be on an USB stick, an external harddrive or just a partition on your built-in hard drive, but it has to be empty.

Secondly, create the random key. In this example, we take 2925 bytes of /dev/random, convert them to base64 (with `uuencode`, usually in the *sharutils* package) and use head and `tail` to take 65 lines of that random block. Finally, we encrypt these numbers with AES256 using GnuPG:

```
> head -c 2925 /dev/random |
   uuencode -m | head -n 66 |
   tail -n 65  | gpg --symmetric
   -cipher-algo aes256
   -a > keyfile.gpg
```

This may take a time, depending on the entropy content available on your system (you need a lot entropy to create random numbers with /dev/random!). The keyfile now can be stored on an USB stick or a smart card, for example. Your encrypted file system will then only be available if you plug in that physical device.

The next step is to initialize the data partition. It will filled with pseudo random numbers once, using /dev/zero to produce a stream of zeros that are encrypted by the encrypting loopback device. This is done only once:

```
> head -c 15 /dev/urandom |
  uuencode -m - | head -n 2 |
  tail -n 1 |
  losetup -p 0 -e aes256
  /dev/loop3 /dev/sdc3
```

This sets up a loopback device /dev/loop3 using the partition /dev/sdc3, which is initialized with some random numbers and AES256. Everything that is now put to /dev/loop3 will be encrypted. A stream of zeros so becomes a long list of pseudo random numbers. It's just a lot faster than generating random numbers, that's why we do it this way. This has been done once only.

```
> dd if=/dev/zero of=/dev/loop3
  bs=4k conv=notrunc 2>/dev/null
```

The initialization is finished when the loopback device is released:

```
> losetup -d /dev/loop3
```

We now have to initialize the file system on our parition:

```
> losetup -K /path/to/your/keyfile.gpg
  -e AES256 /dev/loop3 /dev/sdc3
```

and

```
> mkfs -t ext2 /dev/loop3
```

To release the device again, use

```
> losetup -d /dev/loop3
```

Every time you want to use that partition, set up a loopback device and mount it into your file system. This becomes fairly easy, if you append the following line into your /etc/fstab (all in one line):

```
/dev/sdc3 /mnt/loopdev ext2
   defaults,noauto,loop=/dev/loop3,
   encryption=AES256,
   gpgkey=yourkeyfile 0 0
```

Then it's just:

```
 > mount /mnt/loopdev
Password: keyfile passphrase
```

Data Container Encrypted With DM-Crypt

Another example I want to show here is how you use a container file (just a block of random data on your hard disk) to put in encrypted data. We use DM-Crypt here, which should be available on your system provided you use a kernel 2.6 architecture. If this does not work on your system, have a look at *http://www.saout.de/misc/dm-crypt.*

DM-Crypt is the Device Mapper Target for the encryption of data from Christophe Saout. Since kernel 2.6.4, DM-Crypt replaces Cryptoloop. The Device Mapper administers virtual block devices which in turn can access physical devices like hard disks or partitions. There are quite a number of Device Mapper Targets introducing striping to several block devices for instance. This intermediate layer so to say may as well be equipped with cryptographic features: DM-Crypt.

Make sure that both *Device mapper support* and *Crypt target support* are switched on in your kernel (you find them under *Device Drivers/Multi-Device-Support(RAID and LVM)*). Furthermore, *Device Drivers/Block Devices/Loopback device support* and *Cryptographic Options/AES cipher algorithms* have to be switched on as well.

If you are using Fedora Core, install the *device-mapper* package, Debian users need *dmcrypt* and *cryptsetup* packages. Additionally, some kernel modules have to be loaded. Red Hat or Fedora users can add these three lines to /etc/rc.local:

```
modprobe aes
modprobe dm_mod
modprobe dm_crypt
```

When you are using Debian, use `modconf` and choose *kernel/drivers/md* and *kernel/crypto*.

We will use a 200 MB Data container. It is set up like this:

```
 > dd if=/dev/urandom
 of=container bs=1024k count=200
```

The superuser can now connect the container via DM-Crypt to the device mapper. We use /dev/loop4 here.

```
 > losetup /dev/loop4 container
 > cryptsetup -y create
        secret /dev/loop4
```

You are asked for a passphrase twice (option `-y`) to avoid accidental mistyping. `Container` is the name of the container file, you may have to enter a full path, and `secret` is the name of the device mapper file (feel free to choose different names). You will then find it under `/dev/mapper/secret`. The device does not yet contain a file system, it is set up like this:

```
 > mkfs.ext2 /dev/mapper/secret
```

Now you can mount it:

```
 > mount /dev/mapper/secret /mnt/secret
```

When you are finished using it, type

```
 > umount /mnt/secret
 > cryptsetup remove secret
 > losetup -d /dev/loop4
```

DM-Crypt can not only handle data containers but also whole partitions, and you can encrypt your swap partition easily. This example just showed the data container encryption, you find more information on the DM-Crypt homepage *http://www.saout.de/misc/dm-crypt/.*

Conclusion

In the beginning of the nineties of the last Millennium, the first program for cryptographic purposes was released to the public by Phil R. Zimmerman. This program, PGP, was later referred to as *crpytography for the masses.* The free and Open Source alternative GnuPG allows you to easily encrypt and decrypt your data and e-mail, sign data or both. This article explains the basics of symmetric and asymmetric cryptography and shows you in practice how you set up your keys and how you use them. The second part of the article introduces you to the art of data encryption on the filesystem level in a few easy steps. ●

About the author

Dr. Lars Packschies works as a research associate at the regional computer center of the University of Cologne and is the contact person for chemistry related software and databases as well as for cryptographic applications. He administrates the software and takes care of the privacy protection under Linux, SunOS/Solaris, IRX and AIX. He is the author of *Praktische Kyptografie unter Linux* (*Practical cryptography under Linux*). Contact with the author: *packschies@rrz.uni-koeln.de.*

On the Net

- *http://downlode.org/Etext/alicebob.html*
- *http://www.gnupg.org*
- *http://rfc2440.x42.com* – OpenPGP, RFC 2440
- *http://rfc2822.x42.com* – RFC 2822, S/MIME
- *http://www.cits.rub.de/MD5Collisions* – The Story of Alice and her Boss
- *http://www.heise.de/newsticker/meldung/56624* – Werner Kochs comment in German
- *http://www.gnupg.org/(de)/documentation/faqs.html*
- *http://www.stud.uni-hannover.de/~twoaday/winpt.html*