Systems, Tools, and Terminal Science

BLOG ARCHIVES

GNU/Linux Crypto: Introduction

Posted on

Most of this series has been independently by Rafael Beraldo. Thanks very much, Rafael!

With the growing popularity of operating systems with Linux kernels that do not primarily use GNU components in the base system, this series was retitled to "GNU/Linux Crypto" in May 2017 for less ambiguity and to give .

Cryptography for authentication and encryption is a complex and frequently changing field, and for somebody new to using it, it can be hard to know where to start. If you're a GNU/Linux user comfortable with the terminal, but unfamiliar with the cryptographic tools available to you on open source UNIX-like operating systems, this series of posts aims at getting you set up with some basic tools that will allow you to keep your own information secure, to authenticate conveniently and safely with remote servers, and to work with signed and encrypted files online.

I'll be working on Debian GNU/Linux, but most of these tools should adapt well to other open source UNIX-likes, including BSD. Please feel free to comment on the articles with details relevant to your own implementations, or with extra security considerations for interested readers.

As a disclaimer, I'm not myself an expert on cryptographic algorithms or key security. If you are, and you find an error or security problem with any of my explanations or suggestions, please let me know and I will correct it and credit you.

I'll be covering the following topics:

-
-
-
- `gpg-agent(1)` `ssh-agent(1)` `keychain(1)`
- `pass(1)`
- `mutt(1)`
- `duplicity(1)`
-
-

If you already know about a specific topic, feel free to skip around through the other articles.

This entry is part 1 of 10 in the series .

Posted in | Tagged , , , , , , , , ,
, ,

GNU/Linux Crypto: GnuPG Keys

Posted on

Many tools that use cryptography on GNU/Linux and the internet revolve around the software standard (OpenPGP). The GNU Privacy Guard (GnuPG or GPG) is a popular free software implementation of this standard.

You can install GnuPG with its `gpg(1)` frontend on Debian like so:

```
# apt-get install gnupg
```

You can do a lot of very cool things with GPG, but it boils down to four central ideas:

- Generation of **keypairs**, randomly-generated and mathematically linked pairs of files, one of which is kept permanently secret (the **private key**) and one of which is published (the **public key**). This is the basis of **asymmetric key cryptography**.
- **Managing** keys, both your own public and private key, along with other people's public keys, so that you can verify others' messages and files, or encrypt them so that only those people can read them. This might include publishing your public key to online keyservers, and getting people to sign it to confirm that the key is really yours.
- **Signing** files and messages with your private key to enable others to verify that a file or message was authored or sighted by you, and not edited in transmission over untrusted channels like the internet. The message itself remains readable to everybody.
- **Encrypting** files and messages with other people's public keys, so that only those people can decrypt and read them with their private keys. You can also sign such messages with your own private key so that people can verify that it was sent by you.

We'll run through the fundamentals of each of these. We won't concern ourselves too much with the mathematics or algorithms behind these operations; the Wikipedia article for explains this very well for those curious for further details.

Let's start by generating a 4096-bit RSA keypair, which should be more than sufficient for almost everyone at the time of writing. We'll observe a few of the recommended for the Debian developers.

Doing this on a private, up-to-date desktop machine is best, as it's easier to generate entropy this way. It's still possible on an SSH-only headless server, but you may have to resort to less cryptographically sound methods to generate proper randomness.

Create or edit the file `~/.gnupg/gpg.conf` on your system, and add the following lines:

```
personal-digest-preferences SHA256
cert-digest-algo SHA256
default-preference-list SHA512 SHA384 SHA256 SHA224 AES256 AES192 AES CAST5 ZLIB BZIP2 ZIP Unc
```

These lines tell GnuPG to use the cryptographically stronger SHA256 hashing algorithm for signatures in preference to the .

With that done, we can get down to generating some keys:

```
$ gpg --gen-key
```

You will be prompted to choose the type of keypair you want. The default ought to be `RSA and RSA`, which means we'll generate one master key for signing, and one **subkey** for encryption:

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
```

For the key length, choose the maximum 4096 bit RSA:

```
What keysize do you want? (2048) 4096
Requested keysize is 4096 bits
```

The expiry date is up to you. Good practice is to set an expiry date about a year out, because as long as you have access to the private key material, you can update the expiry date indefinitely, even if it's already expired. For this particular example, we'll set an expiry date one year out:

```
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) 1y
Key expires at Wed 21 Jan 2015 12:24:57 NZDT
Is this correct? (y/N) y
```

Next, we're prompted for some basic information to name the key. In almost all circumstances you should use your real name, as without a real-world means to actually verify your identity, public keys are much less useful long-term. For the comment, you can include the key's purpose, or your public aliases, or any other information relevant to the key:

```
Real name: Tom Ryder
Email address: tom@sanctum.geek.nz
Comment: Test Key Only
You selected this USER-ID:
    "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
```

Next, we're prompted for a passphrase to encrypt the key, so that if it ever falls into the wrong hands, nobody will be able to use it without knowing the passphrase.

```
You need a Passphrase to protect your secret key.
```

Choose a sequence of random words, or possibly a unique sentence in any language, the longer the better. Don't choose anything that might be feasibly guessable, like proverbs or movie quotes. You will also need to remember how you typed the passphrase exactly; I recommend using all-lowercase and no punctuation. Wikipedia has .

You'll need to type the passphrase twice to confirm it, and it won't echo on your terminal, much as if you were typing a password.

Finally, the system will prompt us to generate some entropy:

```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

```
Not enough random bytes available.  Please do some other work to give
the OS a chance to collect more entropy! (Need 283 more bytes)
```

This step is necessary for the computer to generate sufficient random information to ensure that the private key being generated could not feasibly be reproduced. Moving the mouse around and using the keyboard on a desktop system is ideal, but generating any kind of hardware activity (including spinning disks up) should do the trick. Running expensive `find(1)` operations over a filesystem (with contents that couldn't be reasonably predicted or guessed) helps too.

This step benefits from patience. You might find discussion online about forcing the use of the non-blocking PRNG random device `/dev/urandom` instead, using a tool like `rngd(1)`. This definitely speeds up the process, but if you're going to be using your key for anything serious, I recommend actually interacting with the computer and using hardware noise to seed the randomness adequately, if you can.

When adequate entropy is read and the key generation is done, you'll be presented with some details for your master signing key pair and its encrypting subkey pair, and the private and public keys for each are automatically added to your keyring for use:

```
gpg: /home/tom/.gnupg/trustdb.gpg: trustdb created
gpg: key 040FE79B marked as ultimately trusted
public and secret key created and signed.
gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   4096R/040FE79B 2013-03-23
    Key fingerprint = 7A28 5ADA 7680 6813 48DF  401B 6207 438A 040F E79B
uid                  Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>
sub   4096R/AA159E5B 2013-03-23
```

With this done, we have our own keys added to the private and public keychain:

```
$ gpg --list-secret-keys
/home/tom/.gnupg/secring.gpg
----------------------------
sec   4096R/040FE79B 2013-03-23
uid                  Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>
ssb   4096R/AA159E5B 2013-03-23

$ gpg --list-public-keys
/home/tom/.gnupg/pubring.gpg
----------------------------
pub   4096R/040FE79B 2013-03-23
uid                  Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>
sub   4096R/AA159E5B 2013-03-23
```

The directory `~/.gnupg` contains the managed keys. It's very, very important to keep this directory private and to back it up securely, preferably to removable media that you keep in some physically secure place. Don't lose it!

In most contexts in GnuPG, you can refer to a key by the name of its owner, or by its eight-digit hex ID. I prefer the latter method. Here, the short ID of my main key is `040FE79B`. While you shouldn't use this for any actual verification, it's sufficiently unique that you can use it to identify a specific key on your keyring with which you want to work.

For example, if we want to provide someone with a copy of our public key, a friendly way to do so is to export it in ASCII format with `--armor`, providing the appropriate key's short ID:

```
$ gpg --armor --export 040FE79B > tom-ryder.public.asc
```

While you can export private keys the same way with `--export-secret-key`, you should never, ever provide anyone with your private key, so this shouldn't be necessary.

After generating your keys, you should generate a **revocation certificate**:

```
$ gpg --output revoke.asc --gen-revoke 040FE79B

sec  4096R/040FE79B 2013-03-23 Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>

Create a revocation certificate for this key? (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
(Probably you want to select 1 here)
Your decision? 1
Enter an optional description; end it with an empty line:
>
Reason for revocation: Key has been compromised
(No description given)
Is this okay? (y/N) y

You need a passphrase to unlock the secret key for
user: "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
4096-bit RSA key, ID 040FE79B, created 2013-03-23


ASCII armored output forced.
Revocation certificate created.

Please move it to a medium which you can hide away; if Mallory gets
access to this certificate he can use it to make your key unusable.
It is smart to print this certificate and store it away, just in case
your media become unreadable.  But have some caution:  The print system of
your machine might store the data and make it available to others!
```

You should store the resulting `revoke.asc` file somewhere safe. You can use this certificate to later on if the private key is ever compromised, so that people know the key should no longer be used or trusted. You may even like to print it out and keep a hard copy, as the output of `gpg` suggests.

With the above setup done, we can proceed with some basic usage of GnuPG, as discussed in the next article.

In the output of both commands, you'll note we actually have two private and two public keys. The `sub` line refers to the **encryption subkey** automatically generated for you. The master key is used for cryptographic signing, and the subkey for encryption; this is how GnuPG does things by default with RSA keypairs.

For extra security, it might be appropriate to physically remove the master private key from your computer, and instead use a second generated subkey for signing files as well. This is desirable because it allows you to keep the master key secure on some removable media (preferably with a backup), and not loaded on your main computer in case you get compromised.

This means you can sign and encrypt files as normal with your signing subkey and encryption subkey. If those keys ever get compromised, you can simply revoke them and generate new ones with your uncompromised master key; everyone who has signed your public master key or otherwise indicated they trust it will not have to do that all over again.

For details on how to do this, I suggest reading the . However, it's not necessary for performing basic GPG operations.

Thanks to commenter coldtobi for recommending setting a key expiry.

This entry is part 2 of 10 in the series .

Posted in | Tagged , , , , , , , , , , , ,

GNU/Linux Crypto: GnuPG Usage

Posted on

With our private and public key generated and stored, we can start using a few of GnuPG's features to sign, verify, encrypt, and decrypt files and messages for distribution over untrusted channels like the internet.

We'll start by signing a simple text file, using the `--clearsign` option. This includes the signature in the message, which we can then distribute to people to read. Here's the contents of `message.txt`:

```
This is a public message from Tom Ryder.
```

We'll sign that with our new private key like so:

```
$ gpg --clearsign message.txt
```

We're prompted for our passphrase for the private key:

```
You need a passphrase to unlock the secret key for
user: "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
4096-bit RSA key, ID 040FE79B, created 2013-03-23
```

Having provided that, the file `message.txt.asc` is created, with PGP sections and a plaintext ASCII signature:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

This is a public message from Tom Ryder.
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

iQIcBAEBCAAGBQJRTQcTAAoJEGIHQ4oED+ebtjoP/19PlndkGhR46BA6YZmDVdC1
```

```
Snk9aXe4Eo42kRpW13bjo8xg+pb+U26ylHkH52OBB7fJ3/BR5eZMug/RXJLGIf+U
aiylbGVz4dGjkeTCSxtg1TvcyEtzhm1ETfIWLarboj9PPBJfO1QIh4uPDkD7kb6+
+OnpdxPURbiJ03osu2Mj2fFq5wYT0La+I9BdKUAJmS8zt+CaTirKw+xF6l0sEAVv
lqslWjEwF9JCfumKAj8aMBeZMndoKRqW18ZHoYJdP0x3g1SUKAjZ/NRRUGug6Eg+
JTJ82ETCRKGKmYFLkHJ6iCaucrmhLTd9IYyEQZE/weUuClKqtshollhNHF1dD3SF
fWMPq0+29DInjlXwIkXyzVDln1wULNzbd5zv5Wg5b6lSCZlwH0xCrNjiYO7f413c
Ty4q6SqtRFUommnpMA5XcmX7ebbUpMfqfqqzoLqeTpA15Yuhh3DDR6NoMN82oLyF
FFt7UZh/JlYMc8GOnEqyZfT7d57FbKSLn3vpZbH9QXNFWG6/oZabFFyRm8r7k8F2
FzYTdyp59O0dW4T5OJ6a/xo/OnZutUN1RqW6ZJS19Xb4/5eEohFAFL9cDVLuC6zo
HmlA2m5zwv8aYbJad6Rk6+vpQAAdxHgNq/VYdcOfLOtcJAE6Jnm3alVyeXb1PbMB
WBxaM998Z6R8VmeMB7gQ
=WMzO
-----END PGP SIGNATURE-----
```

Note that the message itself is plainly readable; this message isn't encrypted, it's just verified as having been written by a particular person, and not altered since it was written.

Now anyone who has our public key on their keyring (as we ourselves do) can verify that it was actually us who wrote this message:

```
$ gpg --verify message.txt.asc
gpg: Signature made Sat 23 Mar 2013 14:32:17 NZDT using RSA key ID 040FE79B
gpg: Good signature from "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
```

If anybody tampers with the message, even something like removing a period from the end of a sentence, the verification will fail, suggesting the message was tampered with:

```
$ gpg --verify message.txt.asc
gpg: Signature made Sat 23 Mar 2013 14:32:17 NZDT using RSA key ID 040FE79B
gpg: BAD signature from "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
```

For all other files, we likely need to make the signature file separate with a **detached signature**:

```
$ gpg --armor --detach-sign archive.tar.gz
```

This produces a file `archive.tar.gz.asc` in the same directory, containing the signature. We use `--armor` to make the signature in ASCII, which makes for a longer file but easier distribution online.

In this case, both the file and the signature are required for verification; put the signature file first when you check this:

```
$ gpg --verify archive.tar.gz.asc archive.tar.gz
```

You can use this method to verify software downloads from trusted sources, such as the . First, we would download and import all their public keys at the URL they nominate:

```
$ wget http://www.apache.org/dist/httpd/KEYS
$ gpg --import KEYS
```

We could then download an Apache HTTPD release, along with its key, from an arbitrary mirror:

```
$ wget http://www.example.com/apache/httpd/httpd-2.4.4.tar.gz
$ wget https://www.apache.org/dist/httpd/httpd-2.4.4.tar.gz.asc
```

We can then use the key and signature to verify that it's an uncompromised copy of the original file signed by the developers:

```
$ gpg --verify httpd-2.4.4.tar.gz.asc httpd-2.4.4.tar.gz
gpg: Signature made Tue 19 Feb 2013 09:28:39 NZDT using RSA key ID 791485A8
gpg: Good signature from "Jim Jagielski (Release Signing Key) <jim@apache.org>"
gpg:                 aka "Jim Jagielski <jim@jaguNET.com>"
gpg:                 aka "Jim Jagielski <jim@jimjag.com>"
```

```
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: A93D 62EC C3C8 EA12 DB22  0EC9 34EA 76E6 7914 85A8
```

Note that the `gpg` output cautions that this is still not perfect assurance that the release actually came from Jim Jagielski, because we've never met him and can't absolutely, definitely say that this is his public key. , we can however see a lot of other Apache developers have signed his key, which looks promising, but do we know who *they* are?

Despite the lack of absolute certainty, when downloading from mirrors this is a lot better (and harder to exploit) than simply downloading without validating or checksumming at all, given that the signature and the `KEYS` file were downloaded from Apache's own site.

You will need to decide for yourself whether a person's public key really corresponds to them. This might extend to the point of arranging to meet them with government-issued identification!

We can encrypt a file so that only nominated people can decrypt and read it. In this case, we encrypt it not with our own private key, but with the recipient's public key. This means that they will be able to decrypt it using their own private key.

Here's the contents of `secret-message.txt`:

```
This is a secret message from Tom Ryder.
```

Now we need at least one recipient. Let's say this message was intended for my friend John Public. He's given me his public key in a file called `john-public.asc` on a USB drive in person; he even brought along his birth certificate and driver's license (which is weird, because I've known him since I was four).

To start with, I'll import his key into my keychain:

```
$ gpg --import john-public.asc
gpg: key 695195A5: public key "John Public (Main key) <johnpublic@example.com>" imported
```

```
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Now I can encrypt the message for only John to read. I like to use the 8-digit hex code for the key for `--recipient`, to make sure I've got the right person. You can see it in the output above, or in the output of `gpg --list-keys`.

```
$ gpg --armor --recipient 695195A5 --encrypt secret-message.txt
```

The encrypted message is written to `secret-message.txt.asc`:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.10 (GNU/Linux)

hQEMAxiBb8eWSupuAQgAgOUQvqbTh60N6RQhDtP/bY9l+gjm4Grx5XcuhgQqK6pn
YtyPTKcpHdPK679lhbv0vE0RYe7pL+nBOngU1hCQYuGbRDZDxIXTIZW/rBvXbtHA
jgeSxrquad2totfh2nc7upePVCqXncPrLraJyDJBLLMrBHVvmOZymDabJbemOFuq
A/NbcmT3+osptvaEPFdlbgAW+J3vGxXMuQQYkT8GSnuutfEhZRb7SEL1ktaXwaMc
AA6NAan5ak7nCyDDHhDSDFMS9SQQHd8TDvQPF6OzRXlq26EOFD8HvlbDcgc51lbS
+N5nWaHM/CiuPh9dIOEV0H4Y8WDBdgkxp6kXKQfqb9JzAdwQ047r82SJAA7MSqCS
HRVtCRf5SNM12HqTRzF9XXum4uG+HXT6Bpy+K/lYpLgmHcHoUVKh8c2OcGaCHWQh
UC9B+aaThKdkxUfD/9tVIRmugjutgj7KdtDTGm+qLeCoJqp6HK5z5SX8Ha+P6/P5
hxinyw==
=kqUG
-----END PGP MESSAGE-----
```

Note that even I can't read it, because I didn't list myself as a recipient, and I don't have access to John's private key:

```
tom@tombox:~$ gpg --decrypt secret-message.txt.asc
gpg: encrypted with 2048-bit RSA key, ID 964AEA6E, created 2013-03-10
    "John Public (Main key) <johnpublic@example.com>"
gpg: decryption failed: secret key not available
```

However, on John's computer, using his private key, he can decrypt and read it:

```
john@johnbox:~$ gpg --decrypt secret-message.txt.asc
gpg: encrypted with 2048-bit RSA key, ID 964AEA6E, created 2013-03-10
    "John Public (Main key) <johnpublic@example.com>"
This is a private, secret message from Tom Ryder.
```

If I wanted to make sure I could read the message too, I'd add my own public key to identify myself as a recipient when I encrypt it. Then either of us will be able to read it with our private keys (independently of the other):

```
$ gpg --recipient 695195A5 --recipient 040FE79B \
    --armor --encrypt secret-message.txt
```

Just to be thorough, we can sign the message as well to prove it came from us:

```
$ gpg --recipient 695195A5 --recipient 040FE79B \
    --armor --sign --encrypt secret-message.txt
```

Then when John runs the `--decrypt`, `gpg` will automatically verify the signature for us too, provided he has my public key in his keyring:

```
$ gpg --decrypt secret-message.txt.asc
gpg: encrypted with 2048-bit RSA key, ID 964AEA6E, created 2013-03-10
    "John Public (Main key) <johnpublic@example.com>"
gpg: encrypted with 4096-bit RSA key, ID AA159E5B, created 2013-03-23
    "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
This is a private, secret message from Tom Ryder.
gpg: Signature made Sat 23 Mar 2013 17:23:20 NZDT using RSA key ID 040FE79B
gpg: Good signature from "Tom Ryder (Test Key Only) <tom@sanctum.geek.nz>"
```

These are all the basic functions of GnuPG that will be useful to most people. We haven't considered here , or participating in the ; you should only look into this once you're happy with how your key setup is working, and are ready to publish your key for public use.

This entry is part 3 of 10 in the series .

Posted in | Tagged , , , , , , , , ,
, ,

GNU/Linux Crypto: SSH keys

Posted on

The usual method of authenticating to an OpenSSH server is to type your shell password for the remote machine:

```
tom@local:~$ ssh remote
The authenticity of host 'remote (192.168.0.64)' can't be established.
RSA key fingerprint is d1:35:45:a6:d1:b2:e4:08:f8:67:b1:19:fe:04:ca:1c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'remote,192.168.0.64' (RSA) to the list of known hosts.
tom@remote's password:

tom@remote:~$
```

This is appropriate for first contact with a machine, and the authentication method is supported out of the box for most OpenSSH `sshd(8)` installations.

`sshd(8)` is a very common target for attacks, particularly automated ones; malicious bots attempt to connect to servers listening on the standard SSH destination port of `tcp/22`, as well as some common alternatives like `tcp/2222`. If you enforce a strong password policy on your system, this generally isn't too much of a problem, particularly if only appropriate users have shells, or if you restrict SSH connections only to certain usernames or groups.

There are other measures to defeat automated attacks, such as employing systems like to reject clients who make too many spurious connection attempts, but perhaps the most effective way of short-circuiting automated attacks is to bypass passwords completely and instead use **SSH keys**, allowing this as the only connection method to the relevant machines.

Similar to the GnuPG keys setup in the first two articles in this series, SSH keypairs are comprised of one **private key** and one **public key**, two cryptographically linked files. The basis of keys for authentication is that if someone has your public key, they're able to authenticate you by requesting operations that you would only be able to perform with the corresponding private key; it works similarly to cryptographic **signing**.

The reason this is so effective is because if you require a valid public key to authenticate, with sufficient key length it's effectively impossible for an attacker to guess your authentication details; there's no such thing as a "common" private key to guess, so they would need to run through every possible private key, which is not even remotely practical.

Your system's `sshd(8)` may still be attacked, but if you use only public key authentication, then you can be comfortably certain it's *effectively impossible* to brute-force your credentials. Note that this doesn't necessarily protect you from security problems in `sshd(8)` itself, however, and you will still need to protect your private key from being hijacked or compromised, hence the necessity of a **passphrase**.

All of the below assumes you have OpenSSH installed as both the client and the server on the appropriate systems. On Debian-derived systems, these can be installed with:

```
# apt-get install ssh
# apt-get install openssh-server
```

Both the client and server often come standard with systems (e.g. their native).

Similar to the GnuPG setup process, we start by generating a keypair on the machine from which we'd like to connect, using `ssh-keygen(1)`. I'm using 4096-bit RSA here, as it's widely supported even on very old systems, and should be relatively future-proof, although generating new keys if RSA ever becomes unsafe is not hard. If you'd prefer to use the newer

that's the default in recent versions of OpenSSH, all of this will still work. I'm also applying a **comment** for the key as an unencrypted identifier to distinguish multiple keys if I have them. I find email addresses work well.

```
$ ssh-keygen -t rsa -b 4096 -C tom@sanctum.geek.nz
Generating public/private rsa key pair.
```

First, we're prompted for a location to which the key files should be saved. I recommend accepting the default by pressing `Enter`, as using the default location makes the next few steps easier:

```
Enter file in which to save the key (/home/tom/.ssh/id_rsa):
```

Next, we're prompted for a passphrase, which should definitely be added to keep the key from being used if it's ever compromised. The same guidelines for passphrases apply to SSH here, and you should choose a different passphrase:

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

This done, the key is generated, including a pictorial representation to recognise keys at a glance. I've never found this very useful, but the key fingerprint is helpful:

```
Your identification has been saved in /home/tom/.ssh/id_rsa.
Your public key has been saved in /home/tom/.ssh/id_rsa.pub.
The key fingerprint is:
d5:81:8c:eb:c6:c5:a2:b9:6a:ae:32:cc:20:bf:cf:66 tom@local
The key's randomart image is:
+--[ RSA 4096]----+
|         o ..    |
|        . o. .   |
|         o. .    |
|        o.o      |
|       =So       |
```

```
|o     o +        |
|=.     o         |
|oo..E .          |
| ooO=.           |
+-----------------+
```

The key files should now be available in `~/.ssh`:

```
$ ls -l .ssh
-rw-------  1 tom  tom  3326 Apr  2 22:47 id_rsa
-rw-r--r--  1 tom  tom   754 Apr  2 22:47 id_rsa.pub
```

The `id_rsa` file contains the encrypted **private key**, and should be kept locked down and confidential. The `id_rsa.pub` file, however, contains the public key, which can be safely distributed, in the same way as a PGP public key.

We can now arrange to use the newly generated public key for authentication in lieu of a password. Start by ensuring you can connect to the remote machine with your username and password:

```
$ ssh remote
tom@remote's password:
```

Once connected, ensure that the `~/.ssh` directory exists on the remote machine, and that you don't already have keys listed in `~/.ssh/authorized_keys`, as we're about to overwrite them:

```
$ mkdir -p ~/.ssh
$ chmod 0700 ~/.ssh
```

If this worked, close the connection (`exit` or `Ctrl-D`) to return to your local machine's shell, and copy your public key onto the remote machine with `scp(1)`:

```
$ scp ~/.ssh/id_rsa.pub remote:.ssh/authorized_keys
tom@remote's password:
id_rsa.pub    100%    754    0.7KB/s    00:00
```

Note that there's a tool included in recent versions of OpenSSH that does this for you called `ssh-copy-id(1)`, but it's good to have some idea of what it's doing in the background.

With this done, your next connection attempt to the remote host should prompt you for your passphrase, rather than your password:

```
$ ssh remote
Enter passphrase for key '/home/tom/.ssh/id_rsa':
```

At first, it may not seem like you've done much useful here. After all, you still have to type in something to connect each time. From a security perspective, the first major advantage to this method is that neither your password, nor your passphrase, nor your private key are ever transmitted to the server to which you're connecting; authentication is done purely based on the public-private key pair, decrypted by your passphrase.

This means that if the machine you're connecting to were compromised, or your DNS had been poisoned, or some similar attack tricked you into connecting to a fake SSH daemon designed to collect credentials, your private key and your password remain safe.

The second advantage comes with turning off password authentication entirely on the host machine, once all its users have switched to public key authentication only. This is done with the following settings in `sshd_config(5)`, usually in `/etc/ssh/sshd_config` on the remote server:

```
PubkeyAuthentication yes
ChallengeResponseAuthentication no
PasswordAuthentication no
```

Restart the SSH server after these are applied:

```
$ sudo /etc/init.d/ssh restart
```

You should then no longer be able to connect via passwords at all, only by private keys, which as mentioned above are effectively (though not literally) impossible to brute-force. In order to connect to the server as you, an attacker would not only need to know your passphrase, but also have access to your private key, making things significantly harder.

Using public key authentication also allows `sshd(8)` some finer-grained control over authentication, such as which hosts can connect with which keys, whether they can execute TCP or X11 tunnels, and (to an extent) which commands they can run once connected. See the manual page for `authorized_keys(5)` to take a look at some examples.

Finally, there's a major usability advantage in using SSH keys for authentication with **agents**, which we'll discuss in the next article.

SSH connection should ideally be a two-way authentication process. Just as the server to which you're connecting needs to be sure who you are, you need to be sure that the host you're connecting to is the one you expect. With tunnelling, firewalls, DNS poisoning, NAT, hacked systems, and various other tricks, it's appropriate to be careful that you're connecting to the right systems. This is where OpenSSH's **host key** system comes into play.

The first time you connect to a new server, you should see prompts like this:

```
$ ssh newremote
The authenticity of host 'newremote (192.168.0.65)' can't be established.
RSA key fingerprint is f4:4b:f4:8c:c5:50:f6:c8:d3:b2:e9:14:68:86:b5:7b.
Are you sure you want to continue connecting (yes/no)?
```

A lot of administrators turn these off; don't! They are very important.

The **key fingerprint** is a relatively short hash for the host key used by OpenSSH on that server. It's verified by your SSH client, and can't easily be faked. If you're connecting to a new server, it's appropriate to check the host key fingerprint matches the one you see on first connection attempt, or to ask the system's administrator to do so for you.

The host key's fingerprint can be checked on the SSH server with a call to `ssh-keygen(1)`:

```
$ ssh-keygen -lf /etc/ssh/ssh_host_rsa_key
2048 f4:4b:f4:8c:c5:50:f6:c8:d3:b2:e9:14:68:86:b5:7b /etc/ssh/ssh_host_rsa_key.pub (RSA)
```

If you want, you can check the key without making a connection attempt with a similar call on the client side:

```
$ ssh-keygen -lF newremote
# Host 192.168.0.65 found: line 1 type RSA
2048 f4:4b:f4:8c:c5:50:f6:c8:d3:b2:e9:14:68:86:b5:7b newremote (RSA)
```

If you can't check the host key yourself, have the administrator send it to you over a secure, trusted channel, such as in person or via a PGP signed message. If the colon-delimited SSH fingerprint is not exactly the same, then you might be the victim of someone attempting to spoof your connection!

This is definitely overkill for new virtual machines and probably new machines on a trusted LAN, but for machines accessed over the public internet, it's a very prudent practice.

Similarly, `ssh(1)` by default keeps a record of the host keys for hosts, which is why when a different host key is presented on a connection attempt, it warns you:

```
$ ssh newremote
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
d7:06:51:16:80:f6:32:b4:35:7c:53:8d:5a:49:69:ec
Please contact your system administrator.
Add correct host key in /home/tom/.ssh/known_hosts to get rid of this message.
Offending RSA key in /home/tom/.ssh/known_hosts:22
RSA host key for newremote has changed and you have requested strict checking.
```

Again, this is something `ssh(1)` users often turn off, which is pretty risky, especially if you are using password authentication and hence might send your password to a malicious or compromised server!

This entry is part 4 of 10 in the series .

Posted in | Tagged , , , , , ,

GNU/Linux Crypto: Agents

Posted on

Now that we have both GnuPG and SSH securely set up, we're able to encrypt, decrypt, sign, and verify messages, and securely authenticate to remote servers without any risk of exposing passwords and with effectively zero possibility of a brute-force attack. This is all great, but there's still one more weak link in the chain with which to deal — our passphrases.

If you use these tools often, typing your passphrase for most operations can get annoying. It may be tempting to either include a means of automating the passphrase entry, or not to use a passphrase at all, leaving your private key unencrypted. As security-conscious users, we definitely want to avoid the latter in case our private key file ever gets stolen, which is where the concepts of **agents** for both SSH and GnuPG comes into play.

An agent is a daemon designed to streamline the process of using decrypted private keys by storing the details securely in memory, ideally for a limited period of time. What this allows you do with both SSH and GnuPG is to type your passphrase just once, and subsequent uses that require the unencrypted private key are managed by the agent.

In this article, we'll go through the basics of agent setup for both SSH and GnuPG. Once we know how they work, we'll then introduce a convenient tool to start both of them and manage them for us easily.

The `ssh-agent(1)` program comes as part of the . It can be run in two modes, either as a parent process, or daemonized into the background. We'll discuss the latter method, as it's more commonly used and more flexible.

When we run `ssh-agent(1)` for the first time, its behavior is curious; it appears to do nothing except spit out some cryptic shell script:

```
$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-EYqoH3qwfvbe/agent.28881; export SSH_AUTH_SOCK;
SSH_AGENT_PID=28882; export SSH_AGENT_PID;
echo Agent pid 28882;
```

However, we can see that the daemon is running with the PID it mentions:

```
$ ps 28882
  PID TTY      STAT   TIME COMMAND
28882 ?        Ss     0:00 ssh-agent
```

So if it's running fine, what's with all the shell script it outputs? Why doesn't it just run that for us?

The answer is an interesting workaround to a stricture of the Unix process model; specifically, a process cannot modify its parent environment. The variables `SSH_AUTH_SOCK` and `SSH_AGENT_PID` are designed to allow programs like `ssh(1)` to find the agent so it can communicate with it, so we definitely need them set. However, if `ssh-agent(1)` were to set these variables itself, it would only apply for its own process, not the shell where we called it.

Therefore, not only do we need to run `ssh-agent(1)`, we need to execute the code it outputs so the variables get assigned in our shell. A good method of doing this in Bash is using `eval` and command substitution with `$(...)`:

```
$ eval "$(ssh-agent)"
Agent 3954
```

If we run this, we can see that not only is `ssh-agent(1)` running, we have two new variables in our environment identifying its socket path and process ID:

```
$ pgrep ssh-agent
3954
$ env | grep ^SSH
SSH_AUTH_SOCK=/tmp/ssh-oF1sg154ygSt/agent.3953
SSH_AGENT_PID=3954
```

With this done, the agent is ready, and we can start using it to manage our keys for us.

The next step is to load our keys into the agent with `ssh-add(1)`. Pass this program the full path to the private key you would like to use with the agent. This is likely either `~/.ssh/id_rsa` or `~/.ssh/id_dsa`:

```
$ ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/tom/.ssh/id_rsa:
Identity added: /home/tom/.ssh/id_rsa (/home/tom/.ssh/id_rsa)
```

You can leave out the filename argument if you want `ssh-add` to add any or all of the default key types in `~/.ssh` if they exist (`id_dsa`, `id_rsa`, and `id_ecdsa`):

```
$ ssh-add
```

Either way, you should be prompted for your passphrase; this is expected, and you should go ahead and type it in.

If we then ask `ssh-add(1)` to list the keys it's managing, we see the key we just added:

```
$ ssh-add -l
4096 87:ec:57:8b:ea:24:56:0e:f1:54:2f:6b:ab:c0:e8:56 /home/tom/.ssh/id_rsa (RSA)
```

With this done, if we try to use this key to connect to another server, we no longer need to provide the passphrase; we're just logged straight in:

```
tom@local:~$ ssh remote
Welcome to remote.sanctum.geek.nz, running GNU/Linux!
tom@remote:~$
```

The default is to maintain the keys permanently, until the agent is stopped or the keys are explicitly removed one-by-one with `ssh-add -d <keyfile>` or all at once with `ssh-add -D`. For the cautious, you can set a time limit in seconds with `ssh-add -t`. For example, to have `ssh-add` forget about your keys after two hours, you might use:

```
$ ssh-add -t 7200 ~/.ssh/id_rsa
```

To kill the agent completely, you can use `ssh-agent -k`, again with an `eval $(...)` wrapper:

```
$ eval "$(ssh-agent -k)"
Agent pid 4501 killed
```

You may like to consider adding this to `~/.bash_logout` or a similar script to get rid of the running agent after you're done with your session.

If you like this and find it makes your key management more convenient, it makes sense to put it into a startup script like `~/.bash_profile`. This way, the agent will be started for each login shell, and we will be able to communicate with it from any subshell (`xterm`, `screen`, or an appropriately configured `tmux`):

```
eval "$(ssh-agent)"
ssh-add ~/.ssh/id_rsa
```

On our next TTY login, we should be prompted for a passphrase, and from there be able to connect to any machine using the keys managed by the agent:

```
tom@local:~$ ssh remote
Welcome to remote.sanctum.geek.nz, running GNU/Linux!
```

If you want this to work for a desktop manager like GDM or XDM, you can add a variable pointing to the `ssh-askpass(1)` program:

```
eval $(ssh-agent)
export SSH_ASKPASS=/usr/bin/ssh-askpass
ssh-add ~/.ssh/id_rsa
```

If `SSH_ASKPASS` is set like this and `DISPLAY` refers to a working display, then a simple graphical prompt will appear asking for your passphrase:

This program may need to be installed separately. Under Debian-derived systems, its package name is `ssh-askpass`.

All child processes and subshells of the login shell will inherit the agent's variables, since they were exported with `export`:

```
tom@local:~$ screen
tom@local:~$ tmux bash
tom@local:~$ bash
tom@local:~$ ssh remote
Welcome to remote.sanctum.geek.nz, running GNU/Linux!
tom@remote:~$
```

We thus have to type our passphrase only once per login session, and can connect to all of the servers to which our keys confer access … very convenient!

Just like `ssh-agent(1)`, there exists an agent for managing GnuPG keys too, called `gpg-agent(1)`. Its behavior is very similar. On Debian-derived systems, it can be installed as the `gnupg-agent` . You should also install a `pinentry` program; as we're focussing on learning the nuts and bolts on the command line here, we'll use `pinentry-curses(1)` for a console-based passphrase prompt:

```
# apt-get install gnupg-agent pinentry-curses
```

We'll start the agent using the same `eval $(...)` trick we learned with `ssh-agent`:

```
$ eval "$(gpg-agent --daemon)"
```

We can verify that the agent is running in the background with the given PID, and that we have a new environment variable:

```
$ pgrep gpg-agent
5131
$ env | grep ^GPG
GPG_AGENT_INFO=/tmp/gpg-hbro8r/S.gpg-agent:5131:1
```

We'll also set `GPG_TTY`, which will help the pinentry program know on which terminal to draw its passphrase request screen:

```
$ export GPG_TTY=$(tty)
$ echo $GPG_TTY
/dev/pts/2
```

Finally, to prod `gpg(1)` into actually using the agent, we need to add a line to `~/.gnupg/gpg.conf`. You can create this file if it doesn't exist.

```
use-agent
```

With this done, if we try to do anything requiring our private key, we should be prompted for a passphrase not directly on the command line, but by our PIN entry program:

```
$ gpg --armor --sign message1.txt

┌──────────────────────────────────────────────────────┐
│ You need a passphrase to unlock the secret key for user: │
│ "Thomas Ryder (tyrmored, tejr) <tom@sanctum.geek.nz>"    │
│ 4096-bit RSA key, ID 25926609, created 2013-03-12        │
│ (main key ID 77BB8872)                                   │
│                                                          │
│                                                          │
│ Passphrase ***_____________________________________      │
│                                                          │
│       <OK>                              <Cancel>         │
└──────────────────────────────────────────────────────┘
```

When we enter the passphrase, our operation is performed:

```
$ ls message1*
message1.txt
message1.txt.asc
```

Afterwards, if we perform another option requiring the private key, we see that we are not prompted:

```
$ gpg --armor --sign message2.txt
$ ls message2*
message2.txt
message2.txt.asc
```

The agent has thus cached the private key for us, making it much easier to perform a series of operations with it. The default timeout is 10 minutes, but you can change this with the `default-cache-ttl` and `max-cache-ttl` settings in `~/.gnupg/gpg-agent.conf`. For example, to retain any private key for one hour after its last use and a maximum of two hours from its first use, we could write:

```
default-cache-ttl 3600
max-cache-ttl 7200
```

Changing these values will require prompting the agent to reload:

```
$ gpg-connect-agent <<<RELOADAGENT
OK
```

Just like `ssh-agent(1)`, an ideal place for `gpg-agent(1)`'s startup lines is in a login shell setup script like `~/.bash_profile`:

```
eval "$(gpg-agent --daemon)"
```

The agent will be started, and all of its environment variables will be set and inherited by all subshells, just as with `ssh-agent`.

If you're using the console PIN entry tool, you should also add this to end of your interactive shell startup script. This should be `~/.bashrc` for Bash on Linux; you may need to put it in `~/.bashrc` on Mac OS X.

```
export GPG_TTY=$(tty)
```

To manage both `ssh-agent(1)` and `gpg-agent(1)` effectively, a tool called `keychain(1)` is available. It provides a simple way to start both agents with one command, including loading keys at startup, and also prevents running either agent twice, picking up on agents started elsewhere on the system. Because desktop environments are often configured to start one or both agents for users, it makes sense to re-use them where possible, at which `keychain(1)` excels.

On Debian-derived systems, the program is available in the `keychain` :

```
# apt-get install keychain
```

With `keychain` installed, we can start both agents with just one command in `~/.bash_profile`:

```
eval "$(keychain --eval)"
```

We can optionally include the filenames of SSH keys in `~/.ssh` or the hex IDs of GnuPG keys as arguments to prompt loading the private key (including requesting the passphrase) at startup:

```
eval "$(keychain --eval id_rsa 0x77BB8872)"
```

If this program is available to you, then I highly recommend this; managing agents and environments can be fiddly, and `keychain(1)` does all the hard work for you in this regard so you don't have to worry about whether an agent is available to you in your particular context. Check out for more information about the tool.

This entry is part 5 of 10 in the series .

Posted in | Tagged , , , , , ,