TLS

Alessandro Tomasi*[1]

[1]Security and Trust, Fondazione Bruno Kessler

November 16, 2018

Abstract

Generating certificates and using TLS

1 Setup

Use the pyca/cryptography library for convenience.

```
pip install cryptography
```

2 Generating a certificate

2.1 Keys and certificates

Asymmetric key pairs may be generated using an openssl command line argument, e.g.

```
ssh-keygen -t rsa -b 2048
```

For a complete list of parameters, check the `man` page (`man ssh-keygen`); the most common options are:

-t algorithm, or key **t**ype

-b key size, in **b**its

-f output **f**ile name

-p **p**assword to unlock the key

*altomasi@fbk.eu

2.2 Encoding, extensions, and naming conventions

For user keys, the naming convention often followed is for private key files to be `id_<type>`, without extension (e.g. `id_rsa`), and public keys to be in a file of the same name with extension `.pub`. Key files are saved in a hidden subdirectory of the user's home, `/.ssh/`.

While a single user may have several keys for the same algorithm to connect with many servers, server keys are unique per algorithm, named `ssh_host_type_key`, and saved in a location that depends on the software, e.g. `/etc/ssh/` for apache.

Certificates are frequently presented using the PEM (Privacy Enhanced Mail) standard container and encoding. Another, more recent encoding is OpenSSH. We will not consider PKCS (Public-Key Cryptography Standards) as they are RSA proprietary. You will encounter the following file types, by extension:

.key Private key of a certificate, usually PEM encoded. On a server (e.g. Apache) this is sometimes kept in a default folder, such as `/etc/ssl/private`. The access control policy on this file is critical.

.csr Certificate Signing Request.

.pem Container format that usually includes just the public certificate but may also contain other fields, or a CSR.

.pub Public key container, usually OpenSSH.

PEM encoding is more widely used but OpenSSH encoding is considered more robust.

3 Links

On X.509, see the proposed standard RFC 5280, and an introduction on wikipedia. On TLS, the v1.3 standard is RFC 8446; bear in mind that the wikipedia article may still be at v1.2.

Refer to the latest documentation for pyca/cryptography, and their X.509 tutorial.

For historical reasons you may be interested in the PEM (Privacy Enhancement for Internet Electronic Mail) documents, RFC 1421 to 1424. I don't recommend you read the whole thing, but be aware of its existence.

4 Exercises

Practice with user keys and command line ssh:

1. Generate an rsa key pair using `ssh-keygen` with a 2048-bit key. Inspect the files with a text editor.

2. Register for an account on github.com. Follow their instructions on how to connect using an ssh key.

Practice with X.509 using the cryptography python library, or otherwise. You may wish to refer to their latest reference X.509 documentation and tutorial, or otherwise.

1. Generate a self-signed PEM-encoded X.509 certificate with 4092-bit RSA keys. Save the private key, the public key, and the certificate as separate files.

2. Write a script to read a certificate from file and decode it so that its contents are human-readable. Does either key appear in the certificate you created? If so, which key, and in which field?

3. Generate a CSR (Certificate Signing Request). Share it with the person to your left, and ask them to sign it. Once they return it, load the certificate, verify the signature, and inspect the issuer information.

4. Accepted a CSR from the person to your right, sign it, and return it to them.

5. In the previous step: did you check the subject information was correct, or even plausible? Did they in fact use their real name? Did you, when you signed their certificate?

Practice with flask.

1. Run your flask app with the certificates you generated, e.g. with the following command:

   ```
   app.run( ssl_context = ( cert_file , key_file) )
   ```

   The curious among you may wish to look up how the werkzeug server called by flask handles the input, here.