Create Service Certificate

I use the term *"Service Certificate"* here since we are attaching a certificate to a service, be it a web server running Apache or an sftp server using openssh. However, I believe it is commonly called a *"Server Certificate"* as one certificate can generally be used to authenticate multiple services on the same machine.

On our workstations, we installed the Certificate of Authority (CA) at a very low level, so all services could use that to validate remote services. On our servers, we will install Certificates, signed by that CA, to validate that a service is really the one you think it is.

Doing this requires building a certificate, signing it with our CA, then exporting it to the server whose service we want to validate. An example would be an Apache web service, where we might put the certificate into a directory, then modify the Apache configuration to use that certificate for SSL sessions (ie, using https).

Creating the Certificate

Creating the certificate is similar to creating a CA: We create a key file, then turn it into a certificate. The difference is, as an intermediate step, we create a CSR (Certificate Request) which uses the key, then we use that to create and sign the certificate with the CA. The basic functionality is shown in the following four steps.

1. Create an EXT (extension) file containing the names which the service will be called by.
2. Generate a private key
3. Create a signing request
4. Generate the certificate and sign them with the CA

That first step, creating the extension file, is useful for repeated tasks. It is basically an openssl.cnf file, with some additional information (and, ignoring some CA specific information). Since I'm lazy, I simply copy the openssl.cnf file to a new file name, modify it, then I'm ready to go.

We'll take each one in turn, then I'll show you a simple script that can automate the process.

Create EXT file

The EXT file defines how to create the cert and what services it is valid for. You can have one or more names which the certificate is valid for. For example, if a web server can be called as web.example.local, or simply web, you would want both. If you want the same certificate to also handle mail.example.local, you could add that. And, in some cases, it is advantageous to include the IP address(es) of the server it is on.

An EXT file is a lot like a openssl.cnf file, but with some additional stanzas. To simplify things, I tend to copy the openssl.cnf file to a new file name, then add/modify the additional stanzas, then use that new file file for both.

Here is an example of an ext file which has been merged with an openssl.cnf file to allow it to be used for both functions.

www.example.local.ext

```
[ req ]
default_bits       = 2048 # default key size
default_md         = sha256 # default message digest algorithm
distinguished_name = req_distinguished_name # definition used for DN
req_extensions     = v3_req # go look at v3_req section for the
extensions def
prompt             = no # do not ask questions, take defaults

[ req_distinguished_name ]
# Required field, must be different for each certificate
# server and ca
CN = example.com
# not required
C  = US
ST = Texas
O  = Example Corp
L  = Dallas
OU = Headquarters
emailAddress = info@example.com

[ v3_req ]
keyUsage          = critical, digitalSignature, keyEncipherment
extendedKeyUsage  = serverAuth, clientAuth
subjectAltName    = @alt_names # look for section [ alt_names ] for all
the names
basicConstraints  = CA:FALSE

[ alt_names ]
DNS.1 = www.example.local
DNS.2 = example.local
DNS.3 = mail.example.local
IP.1 = 192.168.1.1
```

This is basically the openssl.cnf file, with the addition of the *[v3_req]* and *[alt_names]* stanzas. Actually, the Create Config includes them, but *[alt_names]* has bogus entries.

In the *[v3_req]* stanza, we tell what this key will be used for (critical, digitalSignature, keyEncipherment), tell it the extendedKeyUsage (serverAuth, clientAuth), and add a basicConstraint of CA:FALSE, meaning this is **not** a Certificate of Authority. This is the default usage of this configuration file (remember, we used the -extensions parameter when we created the CA).

Finally, we give it *subjectAltName*, which points to the *[alt_names]* stanza where we will list one or more names the certificate will be responsible for.

In the alt_names, we list on or more names which the service may be called as, using the form

DNS=URL

openssl configuration files can not list the same key more than once, we modify it slightly by adding arbitrary text after a period. openssl will ignore anything between the period and the equals sign except to allow them to coexist in the same configuration file, so we just list them as DNS.1=name DNS.2=alias DNS.3=another alias

Note: For an IP address, use **IP** instead of **DNS** for correctness, though DNS will work for IP addresses. Use the same format (IP.1, IP.2) if you have more than one IP.

This is used when we build a Certificate Request and then integrated as alternate names in the subject for the DN.

I save this file as name (the primary name of the service) with an extension of .ext, so each service (server) will have a different file, which can be reused to recreate the Server Certificate.

Generate Private Key

Private key generation is the same as it was for the CA, except we do not want a password in most cases. If we have a password, it would require you to enter the password every time a service was restarted.

Here, we're creating a private key named www.example.internal.key. This allows us to know which key this is for. Also note we did not include the -des3. Leaving off the encryption algorithm tells genpkey that we don't want to encrypt the key.

```
openssl
   genpkey \
   -algorithm RSA \
   -out servername.key \
   -pkeyopt rsa_keygen_bits:2048
```

here, replace *servername* with the actual name of the server or service as an identifier (same name you used for the .ext file).

Here is a breakdown of the parameters:

- *genpkey* - openssl has multiple functions. This says you want to generate a private key. Notice there is no dash before the command.
- *-algorithm* - Generate using RSA. You should use this for all private keys unless you know a reason not to.
- *-out* - followed by the file name to put the private key in. If not specified, will send output to STDOUT. I use .key as the suffix
- *-pkeyopt* - options specific to the key you are generating. In this case, we are saying the rsa key will be 2048 bits in length

Note: unlike when we created the CA, we did not use the -des3 in this. If you put a passphrase on the key, you will need to manually start each server as the passphrase will be required to open the key (necessary for the certificate, in this case).

Create CSR (Request)

Creating a Certificate Signing Request is simpler since we have the configuration file created earlier. Basically, we call openssl with the req flag and tell it what to do.

A server signing request is simply that. We are asking openssl to create a request from the service to create a certificate, and sign it with the CA. Some systems, like routers and network switches, can generate csr's for you, though I have not tested this part. For now, I'm assuming we will create the csr through openssl

```
openssl \
   req \
   -new \
   -key servername.key \
   -out servername.csr \
   -config servername.ext
```

You can almost read this in english. Create a new (-new) signing request (req) using the key *servername.key*, sending the output to *servername.csr*, using the parameters found in the config file *servername.ext*.

Generate Certificate and sign

The certificate file is what all of this is about. We generate it using the Signing Request (csr), signing with the key.

```
openssl \
   x509 \
   -req \
   -in servername.csr \
   -CA CAName.crt \
   -CAkey CAName.key \
   -out servername.crt \
   -days 365 \
   -extensions req_ext \
   -extfile servername.ext
```

The parameters here are: *x509* - tells openssl we are creating a certificate from a CSR *-req* - normally, this command expects the certificate passed in on input. This option says “use the csr instead”. *-in* - give it the name of the CSR we are using *CA* - The CA Certificate we are signing this with *-CAkey* - the name of the CA's key file. Note that we have to unlock the key during processing, so the passpharse for the key will be requested. *-out* - name of the certificate to be created. We just use servername.crt, with servername being the same as the key, csr, and ext files. *-days* - number of days the certifcate is valid for. The expire date is calculated from the date you create it (when you type the command) plus this value. Note that some programs will not accept certificates with this value set too high, so 365 days is about the max (letsencrypt is set for 90 days by default, I believe). *-extfile* - reads the ext file to get a lot of default parameters.

Note: you can use the parameters *-CAserial* and *-CAcreateserial* to use a sequential serial number on your certificates. It will use a file to store the last serial number and increment that by 1 everytime you ask for a new one. If you do not do this (as we have not), a random serial number is created.

Deploy Certificate and Key

You have created the certificate, and now need to deploy it. This is generally a matter of copying they key and certificate to the target server, and configuring one or more services to use that key. This is covered in the final document, Deploy Server Certificate.