HOWTO proxy certificates

0. WARNING

NONE OF THE CODE PRESENTED HERE HAS BEEN CHECKED! The code is just examples to show you how things could be done. There might be typos or type conflicts, and you will have to resolve them.

1. Introduction

Proxy certificates are defined in RFC 3820. They are really usual certificates with the mandatory extension proxyCertInfo.

Proxy certificates are issued by an End Entity (typically a user), either directly with the EE certificate as issuing certificate, or by extension through an already issued proxy certificate. Proxy certificates are used to extend rights to some other entity (a computer process, typically, or sometimes to the user itself). This allows the entity to perform operations on behalf of the owner of the EE certificate.

See http://www.ietf.org/rfc/rfc3820.txt for more information.

2. A warning about proxy certificates

No one seems to have tested proxy certificates with security in mind. To this date, it seems that proxy certificates have only been used in a context highly aware of them.

Existing applications might misbehave when trying to validate a chain of certificates which use a proxy certificate. They might incorrectly consider the leaf to be the certificate to check for authorisation data, which is controlled by the EE certificate owner.

subjectAltName and issuerAltName are forbidden in proxy certificates, and this is enforced in OpenSSL. The subject must be the same as the issuer, with one commonName added on.

Possible threats we can think of at this time include:

 - impersonation through commonName (think server certificates).
 - use of additional extensions, possibly non-standard ones used in certain environments, that would grant extra or different authorisation rights.

For these reasons, OpenSSL requires that the use of proxy certificates be explicitly allowed. Currently, this can be done using the following methods:

 - if the application directly calls X509_verify_cert(), it can first call:

   X509_STORE_CTX_set_flags(ctx, X509_V_FLAG_ALLOW_PROXY_CERTS);

   Where ctx is the pointer which then gets passed to X509_verify_cert().

 - proxy certificate validation can be enabled before starting the application by setting the environment variable OPENSSL_ALLOW_PROXY_CERTS.

In the future, it might be possible to enable proxy certificates by editing openssl.cnf.

3. How to create proxy certificates

Creating proxy certificates is quite easy, by taking advantage of a lack of
checks in the 'openssl x509' application (*ahem*). You must first create a
configuration section that contains a definition of the proxyCertInfo extension,
for example:

```
  [ v3_proxy ]
  # A proxy certificate MUST NEVER be a CA certificate.
  basicConstraints=CA:FALSE

  # Usual authority key ID
  authorityKeyIdentifier=keyid,issuer:always

  # The extension which marks this certificate as a proxy
  proxyCertInfo=critical,language:id-ppl-anyLanguage,pathlen:1,policy:text:AB
```

It's also possible to specify the proxy extension in a separate section:

```
  proxyCertInfo=critical,@proxy_ext

  [ proxy_ext ]
  language=id-ppl-anyLanguage
  pathlen=0
  policy=text:BC
```

The policy value has a specific syntax, {syntag}:{string}, where the syntag
determines what will be done with the string. The following syntags are
recognised:

  text indicates that the string is simply bytes, without any encoding:

```
          policy=text:räksmörgås
```

        Previous versions of this design had a specific tag for UTF-8 text.
        However, since the bytes are copied as-is anyway, there is no need for
        such a specific tag.

  hex   indicates the string is encoded in hex, with colons between each byte
        (every second hex digit):

```
          policy=hex:72:E4:6B:73:6D:F6:72:67:E5:73
```

        Previous versions of this design had a tag to insert a complete DER
        blob. However, the only legal use for this would be to surround the
        bytes that would go with the hex: tag with whatever is needed to
        construct a correct OCTET STRING. The DER tag therefore felt
        superfluous, and was removed.

  file indicates that the text of the policy should really be taken from a
        file. The string is then really a file name. This is useful for
        policies that are large (more than a few lines, e.g. XML documents).

The 'policy' setting can be split up in multiple lines like this:

```
  0.policy=This is
  1.policy= a multi-
  2.policy=line policy.
```

NOTE: the proxy policy value is the part which determines the rights granted to
the process using the proxy certificate. The value is completely dependent on

the application reading and interpreting it!

Now that you have created an extension section for your proxy certificate, you
can easily create a proxy certificate by doing:

```
  openssl req -new -config openssl.cnf -out proxy.req -keyout proxy.key
  openssl x509 -req -CAcreateserial -in proxy.req -days 7 -out proxy.crt \
    -CA user.crt -CAkey user.key -extfile openssl.cnf -extensions v3_proxy
```

You can also create a proxy certificate using another proxy certificate as
issuer (note: I'm using a different configuration section for it):

```
  openssl req -new -config openssl.cnf -out proxy2.req -keyout proxy2.key
  openssl x509 -req -CAcreateserial -in proxy2.req -days 7 -out proxy2.crt \
    -CA proxy.crt -CAkey proxy.key -extfile openssl.cnf -extensions v3_proxy2
```

4. How to have your application interpret the policy?

The basic way to interpret proxy policies is to start with some default rights,
then compute the resulting rights by checking the proxy certificate against
the chain of proxy certificates, user certificate and CA certificates. You then
use the final computed rights. Sounds easy, huh? It almost is.

The slightly complicated part is figuring out how to pass data between your
application and the certificate validation procedure.

You need the following ingredients:

 - a callback function that will be called for every certificate being
   validated. The callback be called several times for each certificate,
   so you must be careful to do the proxy policy interpretation at the right
   time. You also need to fill in the defaults when the EE certificate is
   checked.

 - a data structure that is shared between your application code and the
   callback.

 - a wrapper function that sets it all up.

 - an ex_data index function that creates an index into the generic ex_data
   store that is attached to an X509 validation context.

Here is some skeleton code you can fill in:

```
  /* In this example, I will use a view of granted rights as a bit
     array, one bit for each possible right.  */
  typedef struct your_rights {
    unsigned char rights[total_rights / 8];
  } YOUR_RIGHTS;

  /* The following procedure will create an index for the ex_data
     store in the X509 validation context the first time it's called.
     Subsequent calls will return the same index.  */
  static int get_proxy_auth_ex_data_idx(void)
  {
    static volatile int idx = -1;
    if (idx < 0)
      {
        CRYPTO_w_lock(CRYPTO_LOCK_X509_STORE);
        if (idx < 0)
```

```
        {
          idx = X509_STORE_CTX_get_ex_new_index(0,
                                                "for verify callback",
                                                NULL,NULL,NULL);
        }
      CRYPTO_w_unlock(CRYPTO_LOCK_X509_STORE);
    }
  return idx;
}

/* Callback to be given to the X509 validation procedure.  */
static int verify_callback(int ok, X509_STORE_CTX *ctx)
{
  if (ok == 1) /* It's REALLY important you keep the proxy policy
                  check within this section.  It's important to know
                  that when ok is 1, the certificates are checked
                  from top to bottom.  You get the CA root first,
                  followed by the possible chain of intermediate
                  CAs, followed by the EE certificate, followed by
                  the possible proxy certificates.  */
    {
      X509 *xs = ctx->current_cert;

      if (xs->ex_flags & EXFLAG_PROXY)
        {
          YOUR_RIGHTS *rights =
            (YOUR_RIGHTS *)X509_STORE_CTX_get_ex_data(ctx,
              get_proxy_auth_ex_data_idx());
          PROXY_CERT_INFO_EXTENSION *pci =
            X509_get_ext_d2i(xs, NID_proxyCertInfo, NULL, NULL);

          switch (OBJ_obj2nid(pci->proxyPolicy->policyLanguage))
            {
            case NID_Independent:
              /* Do whatever you need to grant explicit rights to
                 this particular proxy certificate, usually by
                 pulling them from some database.  If there are none
                 to be found, clear all rights (making this and any
                 subsequent proxy certificate void of any rights).
              */
              memset(rights->rights, 0, sizeof(rights->rights));
              break;
            case NID_id_ppl_inheritAll:
              /* This is basically a NOP, we simply let the current
                 rights stand as they are. */
              break;
            default:
              /* This is usually the most complex section of code.
                 You really do whatever you want as long as you
                 follow RFC 3820.  In the example we use here, the
                 simplest thing to do is to build another, temporary
                 bit array and fill it with the rights granted by
                 the current proxy certificate, then use it as a
                 mask on the accumulated rights bit array, and
                 voilà, you now have a new accumulated rights bit
                 array.  */
              {
                int i;
                YOUR_RIGHTS tmp_rights;
                memset(tmp_rights.rights, 0, sizeof(tmp_rights.rights));
```

```
                /* process_rights() is supposed to be a procedure
                   that takes a string and it's length, interprets
                   it and sets the bits in the YOUR_RIGHTS pointed
                   at by the third argument.  */
                process_rights((char *) pci->proxyPolicy->policy->data,
                               pci->proxyPolicy->policy->length,
                               &tmp_rights);

                for(i = 0; i < total_rights / 8; i++)
                  rights->rights[i] &= tmp_rights.rights[i];
              }
              break;
            }
          PROXY_CERT_INFO_EXTENSION_free(pci);
        }
      else if (!(xs->ex_flags & EXFLAG_CA))
        {
          /* We have a EE certificate, let's use it to set default!
          */
          YOUR_RIGHTS *rights =
            (YOUR_RIGHTS *)X509_STORE_CTX_get_ex_data(ctx,
              get_proxy_auth_ex_data_idx());

          /* The following procedure finds out what rights the owner
             of the current certificate has, and sets them in the
             YOUR_RIGHTS structure pointed at by the second
             argument.  */
          set_default_rights(xs, rights);
        }
    }
  return ok;
}

static int my_X509_verify_cert(X509_STORE_CTX *ctx,
                               YOUR_RIGHTS *needed_rights)
{
  int i;
  int (*save_verify_cb)(int ok,X509_STORE_CTX *ctx) = ctx->verify_cb;
  YOUR_RIGHTS rights;

  X509_STORE_CTX_set_verify_cb(ctx, verify_callback);
  X509_STORE_CTX_set_ex_data(ctx, get_proxy_auth_ex_data_idx(), &rights);
  X509_STORE_CTX_set_flags(ctx, X509_V_FLAG_ALLOW_PROXY_CERTS);
  ok = X509_verify_cert(ctx);

  if (ok == 1)
    {
      ok = check_needed_rights(rights, needed_rights);
    }

  X509_STORE_CTX_set_verify_cb(ctx, save_verify_cb);

  return ok;
}
```

If you use SSL or TLS, you can easily set up a callback to have the certificates checked properly, using the code above:

```
SSL_CTX_set_cert_verify_callback(s_ctx, my_X509_verify_cert, &needed_rights);
```