NAME

proxy−certificates − Proxy certificates in OpenSSL

DESCRIPTION

Proxy certificates are defined in RFC 3820. They 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 (End Entity) certificate.

The requirements for a valid proxy certificate are:

- They are issued by an End Entity, either a normal EE certificate, or another proxy certificate.
- They must not have the **subjectAltName** or **issuerAltName** extensions.
- They must have the **proxyCertInfo** extension.
- They must have the subject of their issuer, with one **commonName** added.

Enabling proxy certificate verification

OpenSSL expects applications that want to use proxy certificates to be specially aware of them, and make that explicit. This is done by setting an X509 verification flag:

```
X509_STORE_CTX_set_flags(ctx, X509_V_FLAG_ALLOW_PROXY_CERTS);
```

or

```
X509_VERIFY_PARAM_set_flags(param, X509_V_FLAG_ALLOW_PROXY_CERTS);
```

See ‘‘NOTES’’ for a discussion on this requirement.

Creating proxy certificates

Creating proxy certificates can be done using the **openssl−x509** (1) command, with some extra extensions:

```
[ 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 *syntag*s are recognised:

text
indicates that the string is a byte sequence, without any encoding:

```
policy=text:räksmörgås
```

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

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

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

NOTE: The proxy policy value is what determines the rights granted to the process during the proxy certificate. It's up to the application to interpret and combine these policies.

With a proxy extension, creating a proxy certificate is a matter of two commands:

```
openssl req −new −config proxy.cnf \
    −out proxy.req −keyout proxy.key \
    −subj "/DC=org/DC=openssl/DC=users/CN=proxy 1"

openssl x509 −req −CAcreateserial −in proxy.req −out proxy.crt \
    −CA user.crt −CAkey user.key −days 7 \
    −extfile proxy.cnf −extensions v3_proxy1
```

You can also create a proxy certificate using another proxy certificate as issuer (note: using a different configuration section for the proxy extensions):

```
openssl req −new −config proxy.cnf \
    −out proxy2.req −keyout proxy2.key \
    −subj "/DC=org/DC=openssl/DC=users/CN=proxy 1/CN=proxy 2"

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

Using proxy certs in applications

To interpret proxy policies, the application would normally start with some default rights (perhaps none at all), then compute the resulting rights by checking the rights against the chain of proxy certificates, user certificate and CA certificates.

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

The following ingredients are needed for such processing:

- a callback function that will be called for every certificate being validated. The callback is 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.

The following skeleton code can be used as a starting point:

```
#include <string.h>
#include <netdb.h>
#include <openssl/x509.h>
#include <openssl/x509v3.h>

#define total_rights 25

/*
 * 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 + 7) / 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(X509_STORE_CTX *ctx)
{
    static volatile int idx = -1;

    if (idx < 0) {
        X509_STORE_lock(X509_STORE_CTX_get0_store(ctx));
        if (idx < 0) {
            idx = X509_STORE_CTX_get_ex_new_index(0,
                                                  "for verify callback",
                                                  NULL,NULL,NULL);
        }
        X509_STORE_unlock(X509_STORE_CTX_get0_store(ctx));
    }
    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 = X509_STORE_CTX_get_current_cert(ctx);

        if (X509_get_extension_flags(xs) & EXFLAG_PROXY) {
            YOUR_RIGHTS *rights =
                (YOUR_RIGHTS *)X509_STORE_CTX_get_ex_data(ctx,
                    get_proxy_auth_ex_data_idx(ctx));
            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 its
                     * 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 (!(X509_get_extension_flags(xs) & EXFLAG_CA)) {
            /* We have an 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(ctx));

            /*
             * 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 ok;
        int (*save_verify_cb)(int ok,X509_STORE_CTX *ctx) =
            X509_STORE_CTX_get_verify_cb(ctx);
        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(ctx),
                                   &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);
```

NOTES

To this date, it seems that proxy certificates have only been used in environments that are aware of them, and no one seems to have investigated how they can be used or misused outside of such an environment.

For that reason, OpenSSL requires that applications aware of proxy certificates must also make that explicit.

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.

SEE ALSO

X509_STORE_CTX_set_flags (3), **X509_STORE_CTX_set_verify_cb** (3), **X509_VERIFY_PARAM_set_flags** (3), **SSL_CTX_set_cert_verify_callback** (3), **openssl−req** (1), **openssl−x509** (1), RFC 3820 <https://tools.ietf.org/html/rfc3820>

COPYRIGHT