The Random Engineer

…passionately curious.

Adding Custom Data to X.509 SSL Certificates

APRIL 18, 2014 DUSTIN

Signed SSL certificates have a feature known as "extensions". In order for them to be there, they must be in the CSR. Therefore, CSR's support them too. Although X.509 certificates are not meant for a lot of data and were never meant to act as databases (rather, an identity with associated information), **they act as a great solution when you need to store secured information alongside your application at a client site.** Though the data is viewable, you have the following guarantees:

- The data (including the extensions) can not be interfered with, or it'll fault its signatures.
- The certificate will expire at a set time (and can be renewed if need be).
- A certificate-revocation list (CRL) can be implemented (using a CRL distribution point, or "CDP") so that you can invalidate a certificate remotely.

As long as you don't care about keep the data secret, this makes extensions an ideal solution to a problem like on-site client licenses, where your software needs to regularly check whether the client still has permission to operate. You can also use a CRL to disable them if they stop paying their bill.

These extensions accommodate data that goes beyond the distinguished-name (DN) fields (locality, country, organization, common-name, etc..), chain of trust, key fingerprints, the signatures that guarantee the trustworthiness of the certificate (using the signature of the CA), and the integrity of the certificate (the signature of the certificate contents). Extensions seem relatively easy to add to certificates, whether you're creating CSRs from code or from command-line. They're just manageably-sized strings (though it technically seems like there is no official length limit) of human-readable text.

If you own the CA, then you might also create your own extensions. In this case, you'll refer to your extension with a unique dotted identifier called an "OID" (we'll go into this in the ASN.1 explanation below). Libraries might have trouble if you just refer to your own extension without properly registering it with your library prior. For example, OpenSSL has the ability to register and use custom extensions, but the M2Crypto SSL library doesn't expose the registration call, and, therefore, can't use custom extensions.

Unsupported extensions might be skipped or omitted from the signed certificate by a CA that doesn't recognize/support them, so beware that you'll need to stick to the popular extensions if you *can't* use your own CA. Extensions that are mandatory for you requirements can be marked as "critical", so that signing won't precede if any of your extensions aren't recognized.

The extension that we're interested in, here, is "subjectAltName", and it is recognized/supported by

all CAs. This extension can describe the "alternative subjects" (using DNS-type entries) that you might need to specify if your X.509 needs to be used with more than one common-name (more than one hostname). It can also describe email-addresses and other kinds of identity information. However, it can also store custom text.

This is an example of two "subjectAltName" extensions (multiple instances of the same extensions can be present in a certificate):

```
DNS:server1.yourdomain.tld, DNS:server2.yourdomain.tld
otherName:1.3.6.1.4.1.99;UTF8:This is arbitrary data.
```

However, due to details soon to follow, it's very difficult to pull the extension text back out, again. In order to go further, we have to take a quick diversion into certificate structure. This isn't necessarily required, but it is information that is obscure-enough to find that you won't have any coping skills if you encounter issues, otherwise.

Certificate Encoding

All of the standard, human-readable, SSL documents, such as the private-key, public-key, CSR, and X.509, are encoded in a format called *PEM*. This is base64-encoded data with anchors (e.g. "––BEGIN DSA PRIVATE KEY––") on the top and bottom.

In order to have any use, a PEM-encoded document must be converted to a DER-encoded document. This just means that it's stripped of the anchors and newlines, and then base64-decoded. DER is a tighter subset of "BER" encoding.

ASN.1 Encoding

The DER-encoding describes an ASN.1 data structure node. ASN.1 combines a tree of data with a tree of grammar specifications, and reduces down to hierarchical sets of DER-encoded data. All nodes (called "tags") are represented by dot-separated identifiers called OIDs (mentioned above). Usually these are officially-assigned OIDs, but you might have some custom ones if you don't have to pass your certificates to higher authority that might have a problem with them.

In order to decode the structure, you must walk it, applying the correct specs as required. There is nothing self-descriptive within the data. This makes it fast, but useless until you have enough pre-existing knowledge to descend to the information you require.

The specification for the common grammars (like RFC 2459 for X.509) in ASN.1 is so massive that you should expect to avoid getting involved in the mechanics at all costs, and to learn how to survive with the limited number of libraries already available. In all likelihood, a need for anything outside the realm of popular usage will require a non-trivial degree of debugging.

ASN.1 has been around… for a while (about thirty years, as of this year). It's obtuse, impossible, and not understood in great deal by very few individuals. However, it's going to be here for a while.

Extension Decoding

The reason that extensions are tough to decode is because the encoding depends on the text that you put in the extension. Specifically, the "otherName" and "UT8" parts. OpenSSL can't present these values when it dumps the certificate, because it just doesn't have enough information to decode them. M2Crypto, since it uses OpenSSL, has the same problem.

Now that we've introduced a little of the conceptual ASN.1 structure, let's go back to the previous *subjectAltName* "otherName" example:

```
otherName:1.3.6.1.4.1.99;UTF8:This is arbitrary data.
```

The following is the breakdown:

1. "otherName": A classification of the subjectAltName extension that indicates custom-data. This has an OID of its own in the RFC 2459 grammar.
2. 1.3.6.1.4.1.99: The OID of your company. The first eight parts comprise the common-prefix, combined with a "private enterprise number" (PEN). You can register for your own (http://pen.iana.org/pen/PenApplication.page).
3. Custom data, prefixed with a type. The "UTF8" prefix determines the encoding of the data, but is not itself included.

I used the following calls to M2Crypto to add these extensions to the X.509:

```
1  ext = X509.new_extension(
2          'subjectAltName',
3          'otherName:1.3.6.1.4.1.99;UTF8:This is arbitrary data.'
4      )
5
6  ext.set_critical(1)
7  cert.add_ext(ext)
```

Aside from the extension information itself, I also indicate that it's to be considered "critical". Signing will fail if the CA doesn't recognize the extension, and not simply omit it. When this gets encoded, it'll be encoded as three separate "components":

1. The OID for the "otherName" type.
2. The "critical" flag.
3. A DER-encoded sequence of the PEN and the UTF8-encoded string.

It turns out that it's quicker to use a library that specializes in ASN.1, rather than trying to get the information from OpenSSL. After all, it's out-of-scope as it's colocated with cryptographical data while not being cryptographical itself.

I used pyasn1 (http://pyasn1.sourceforge.net/).

Decoding Our Extension

To decode the string from the previous extension:

1. Enumerate the extensions.
2. Decode the third component (mentioned above) using the RFC 2459 "subjectAltName" grammar.
3. Descend to the first component of the "SubjectAltName" node: the "GeneralName" node.
4. Descend to the first component of the "General Name" node: the "AnotherName" nerve.
5. Match the OID against the OID we're looking for.
6. Decode the string using the RFC 2459 UTF8 specification.

This is a dump of the structure using *pyasn1*:

```
SubjectAltName().
   setComponentByPosition(
       0,
       GeneralName().
           setComponentByPosition(
               0,
               AnotherName().
                   setComponentByPosition(
                       0,
                       ObjectIdentifier(1.3.6.1.5.5.7.1.99)
                   ).
                   setComponentByPosition(
                       1,
                       Any(hexValue='0309006465616462656566')
                   )
           )
   )
```

The process might seem easy, but this took some work (and collaboration) to get right, with the primary difficulty coming from obscurity meeting unfamiliarity. However, the process should be somewhat set in stone, every time.

This is the corresponding code. "cert" is an M2Crypto X.509 certificate:

```
cert, rest = decode(cert.as_der(), asn1Spec=rfc2459.Certificate())

extensions = cert['tbsCertificate']['extensions']
for extension in extensions:
    extension_oid = extension.getComponentByPosition(0)
    print("0 [%s]" % (repr(extension_oid)))

    critical_flag = extension.getComponentByPosition(1)
```

Wrap Up

I wanted to provide an end-to-end tutorial in adding and retrieving “otherName”-type “subjectAltName” extensions because none currently exists. It’s a good solution for keeping data safe on someone else’s assets (as long as you don’t overburden the certificate with extensions, as it’ll decrease the efficiency to verify).

Don’t forget to implement the CRL/CDP, or you won’t have the possibility of faulting the certificate (and its extensions) without having to wait for them to expire.

ASN.1 **DER** **OPENSSL** **PYASN1** **PYTHON** **SSL** **X.509**