

# **X.509 Style Guide**

**Peter Gutmann, [pgut001@cs.auckland.ac.nz](mailto:pgut001@cs.auckland.ac.nz)  
October 2000**



## Table of Contents

<b>TABLE OF CONTENTS .....</b>	<b>3</b>
<b>INTRODUCTION .....</b>	<b>5</b>
<b>CERTIFICATE.....</b>	<b>7</b>
TBSCERTIFICATE .....	7
VERSION .....	8
SERIALNUMBER .....	8
SIGNATURE.....	9
NAME .....	9
VALIDITY .....	13
SUBJECTPUBLICKEYINFO.....	14
UNIQUEIDENTIFIER.....	14
<b>EXTENSIONS.....</b>	<b>15</b>
KEY USAGE, EXTENDED KEY USAGE, AND NETSCAPE CERT-TYPE .....	18
BASIC CONSTRAINTS .....	19
ALTERNATIVE NAMES .....	19
CERTIFICATE POLICIES AND POLICY MAPPINGS AND CONSTRAINTS .....	20
NAME CONSTRAINTS .....	21
SUBJECT AND AUTHORITY KEY IDENTIFIERS.....	21
OTHER EXTENSIONS .....	22
<b>CHARACTER SETS.....</b>	<b>24</b>
<b>COMPARING DNS.....</b>	<b>28</b>
<b>PKCS #10 .....</b>	<b>30</b>
<b>ASN.1 DESIGN GUIDELINES .....</b>	<b>32</b>
<b>BASE64 ENCODING .....</b>	<b>37</b>
<b>KNOWN BUGS/PECULIARITIES .....</b>	<b>38</b>
AN POST .....	38
BANKGATE .....	38
BELSIGN .....	38
BTG .....	38
CDSA.....	38
COMPUSOURCE .....	38
COST .....	38
CRLs.....	39
DAP .....	39
DEUTSCHES FORSCHUNGSNETZ (DFN) .....	39
DIGICERT.....	39
DIGINOTAR.....	39
DIGITAL SIGNATURE TRUST .....	39
DIGITRUST HELLAS .....	39
DNS WITH UNIQUEIDS .....	40
ENTRUST .....	40
ESTONIAN CLO CA.....	40
ESTONIAN NATIONAL PCA.....	40

## X.509 Style Guide

ESTONIAN IOC CA.....	40
ESTONIAN PIIRIVALVEAMET CA .....	40
ESTONIAN POLITSEI CA .....	40
FIRST DATA .....	40
GENERALNAME.....	41
GIP-CPS .....	41
GTE .....	41
HBCI.....	41
IBM.....	41
ICE-TEL .....	41
IPSEC .....	41
JDK/JAVA .....	41
KEYWITNESS .....	42
LDAP V2/QUIPU.....	42
MICROSOFT .....	42
MISSI.....	44
NASA .....	45
NETLOCK.....	45
NETSCAPE.....	45
NIST .....	45
NORTEL .....	46
PKIX.....	46
SAFELAYER .....	46
SECUDE.....	46
SECURENET.....	47
SECURITY DOMAIN/ZERGO/CERTIFICATES AUSTRALIA.....	47
SEIS .....	47
SET .....	47
SHTTP SPECIFICATION.....	47
SI-CA.....	48
SIGNET .....	48
SOUTH AFRICAN CA.....	48
SSLEAY.....	48
SWEDEN POST/ID2 .....	48
SWIFT .....	49
SYSCALL GBR .....	49
TC TRUSTCENTER .....	49
TELESEC/DEUTSCHE TELEKOM TRUSTCENTER.....	49
THAWTE .....	50
TIMESTEP/NEWBRIDGE/ALCATEL .....	50
UNINETT .....	50
UNIQUEIDENTIFIER .....	50
VERISIGN.....	50
Y2K/2038 ISSUES .....	51
<b>ANNEX A - THE STANDARDS DESIGNER JOKE.....</b>	<b>52</b>
<b>ACKNOWLEDGEMENTS .....</b>	<b>53</b>

## Introduction

There seems to be a lot of confusion about how to implement and work with X.509 certificates, either because of ASN.1 encoding issues, or because vagueness in the relevant standards means people end up taking guesses at what some of the fields are supposed to look like. For this reason I've put together these guidelines to help in creating software to work with X.509 certificates, PKCS #10 certification requests, CRLs, and other ASN.1-encoded data types.

I knew a guy who set up his own digital ID heirarchy, could issue his own certificates, sign his own controls, ran SSL on his servers, etc. I don't need to pay Verisign a million bucks a year for keys that expire and expire. I just need to turn off the friggen [browser warning] messages.

-- Mark Bondurant, "Creating My Own Digital ID", in alt. computer. security.

In addition, anyone who has had to work with X.509 has probably experienced what can best be described as ISO water torture, which involves ploughing through all sorts of ISO, ANSI, ITU, and IETF standards, amendments, meeting notes, draft standards, committee drafts, working drafts, and other work-in-progress documents, some of which are best understood when held upside-down in front of a mirror (this has lead to people trading hard-to-find object identifiers and ASN.1 definitions like baseball cards - "I'll swap you the OID for triple DES in exchange for the latest CRL extensions"). This document is an attempt at providing a cookbook for certificates which should give you everything that you can't easily find anywhere else, as well as comments on what you'd typically expect to find in certificates.

Given humanity's track record with languages, you wonder why we bother with standards committies.

-- Marcus Leech

Since the original X.509 spec is somewhat vague and open-ended, every non-trivial group which has any reason to work with certificates has to produce an X.509 profile which nails down many features which are left undefined in X.509.

You can't be a real country unless you have a beer and an airline. It helps if you have some kind of a football team, or some nuclear weapons, but at the very least you need a beer.

-- Frank Zappa

And an X.509 profile.

-- Me

The difference between a specification (X.509) and a profile is that a specification doesn't generally set any limitations on combinations what can and can't appear in various certificate types, while a profile sets various limitations, for example by requiring that signing and confidentiality keys be different (the Swedish profile requires this, and the German profile specifies exclusive use of certificates for digital signatures). The major profiles in use today are:

PKIX	Internet PKI profile
FPKI	(US) Federal PKI profile
MISSI	US DoD profile
ISO 15782	Banking - Certificate Management Part 1: Public Key Certificates
TeleTrust/MailTrusT	German MailTrusT profile for TeleTrusT (it really is capitalised that way)
German SigG Profile	Profile to implement the German digital signature law (the certificate profile Sigl is particularly good, providing not just the usual specification but also examples of each certificate field and extension including the encoded forms)
ISIS Profile	Another German profile
Australian Profile	Profile for the Australian PKAF (this may be the same as DR 98410, which I haven't seen yet)
SS 61 43 31 Electronic ID Certificate	Swedish profile

FINEID S3	Finnish profile
ANX Profile	Automotive Network Exchange profile
Microsoft Profile	This isn't a real profile, but the software is widespread enough and nonstandard enough that it constitutes a significant de facto profile

No standard or clause in a standard has a divine right of existence.

-- A Microsoft PKI architect explaining Microsoft's position on standards compliance.

Unfortunately the official profiles tend to work like various monotheistic religions where you either do what we say or burn in hell (that is, conforming to one profile generally excludes you from claiming conformance with any others unless they happen to match exactly). This means that you need to either create a chameleon-like implementation which can change its behaviour at a whim, or restrict yourself to a single profile which may not be accepted in some locales. There is (currently) no way to mark a certificate to indicate that it should be processed in a manner conformant to a particular profile, which makes it difficult for a relying party to know how their certificate will be processed by a particular implementation.

Interoperability Testing. Conclusion: It doesn't work.

-- Richard Lampart, CESG, talking about UK government - PKI experiences

Although I've tried to take into account the various "Use of this feature will result in the immediate demise of all small furry animals in an eight-block radius"-type warnings contained in various standards documents to find a lowest common denominator set of rules which should result in the least pain for all concerned if they're adhered to, the existence of conflicting profiles makes this a bit difficult. The idea behind the guide is to at least try to present a "If you do this, you should be OK" set of guidelines, rather than a "You're theoretically allowed to do this if you can find an implementation which supports it" feature list.

Finally, the guide contains a (rather lengthy) list of implementation errors, bugs, and problems to look out for with various certificates and the related software in order to allow implementors to create workarounds.

The conventions used in the text are:

- All encodings follow the DER unless otherwise noted.
- Most of the formats are ASN.1, or close enough to it to be understandable (the goal was to make it easily understandable, not perfectly grammatically correct). Occasionally 15 levels of indirection are cut out to make things easier to understand.

The resulting type and value of an instance of use of the new value notation is determined by the value (and the type of the value) finally assigned to the distinguished local reference identified by the keyword VALUE, according to the processing of the macrodefinition for the new type notation followed by that for the new value notation.

-- ISO 8824:1988, Annex A

## Certificate

```
Certificate ::= SEQUENCE
{
    tbsCertificate          TBSCertificate,
    signatureAlgorithm      AlgorithmIdentifier,
    signature               BIT STRING
}
```

The goal of a cert is to identify the holder of the corresponding private key, in a fashion meaningful to relying parties.

-- Stephen Kent

By the power vested in me, I now declare this text string and this bit string 'name' and 'key'. What RSA has joined, let no man put asunder.

-- Bob Blakley

The encoding of the Certificate may follow the BER rather than the DER. At least one implementation uses the indefinite-length encoding form for the SEQUENCE.

### ***TBSCertificate***

The default tagging for certificates varies depending on which standard you're using. The original X.509v1 definition used the ASN.1 default of explicit tags, with X.509v3 extensions in a separate module with implicit tags. The PKIX definition is quite confusing because the ASN.1 definitions in the appendices use TAGS IMPLICIT but mix in X.509v3 definitions which use explicit tags. Appendix A has such a mixture of implied implicit and implied explicit tags that it's not really possible to tell what tagging you're supposed to use. Appendix B (which first appeared in draft 7, March 1998) is slightly better, but still confusing in that it starts with TAGS IMPLICIT, but tries to partially switch to TAGS EXPLICIT for some sections (for example the TBSCertificate has an 'EXPLICIT' keyword in the definition which is probably intended to signify that everything within it has explicit tagging, except that it's not valid ASN.1). The definitions given in the body of the document use implicit tags, and the definitions of TBSCertificate and TBSCertList have both EXPLICIT and IMPLICIT tags present. To resolve this, you can either rely entirely on Appendix B with the X.509v1 sections moved into a separate section declared without 'IMPLICIT TAGS', or use the X.509v3 definitions. The SET definitions consistently use implicit tags.

Zaphod felt he was teetering on the edge of madness and wondered whether he shouldn't just jump over and have done with it.

-- Douglas Adams, "The Restaurant at the End of the Universe"

```
TBSCertificate ::= SEQUENCE
{
    version          [ 0 ] Version DEFAULT v1(0),
    serialNumber     CertificateSerialNumber,
    signature        AlgorithmIdentifier,
    issuer           Name,
    validity         Validity,
    subject          Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID   [ 1 ] IMPLICIT UniqueIdentifier OPTIONAL,
    subjectUniqueID [ 2 ] IMPLICIT UniqueIdentifier OPTIONAL,
    extensions       [ 3 ] Extensions OPTIONAL
}
```

## Version

```
Version ::= INTEGER { v1(0), v2(1), v3(2) }
```

This field is used mainly for marketing purposes to claim that software is X.509v3 compliant (even when it isn't). The default version is v1(0), if the issuerUniqueID or subjectUniqueID are present than the version must be v2(1) or v3(2). If extensions are present than the version must be v3(2). An implementation should target v3 certificates, which is what everyone is moving towards.

I was to learn later in life that we tend to meet any new situation by reorganizing: and a wonderful method it can be for creating the illusion of progress, while producing confusion, inefficiency and demoralization

-- Petronius Arbiter, ~60 A. D

Note that the version numbers are one less than the actual X.509 version because in the ASN.1 world you start counting from 0, not 1 (although it's not necessary to use sequences of integers for version numbers. X.420, for example, is under the impression that 2 is followed by 22 rather than the more generally accepted 3).

If your software generates v1 certificates, it's a good idea to actually mark them as such and not just mark everything as v3 whether it is or not. Although no standard actually forbids marking a v1 certificate as v3, backwards-compatibility (as well as truth-in-advertising) considerations would indicate that a v1 certificate should be marked as such.

## SerialNumber

```
CertificateSerialNumber ::= INTEGER
```

This should be unique for each certificate issued by a CA (typically a CA will keep a counter in persistent store somewhere, perhaps a config file under Unix and in the registry under Windows). A better way is to take the current time in seconds and subtract some base time like the first time you ran the software, to keep the numbers manageable. This has the further advantage over a simple sequential numbering scheme that it doesn't allow tracking of the number of certificates which have been signed by a CA, which can have nasty consequences both if various braindamaged government regulation attempts ever come to fruition, and because by using sequential numbers a CA ends up revealing just how few certs it's actually signing (at the cost of a cert per week, the competition can find out exactly how many certs are being issued each week).

Although this is never mentioned in any standards document, using negative serial numbers is probably a bit silly (note the caveat about encoding INTEGER values in the section on SubjectPublicKeyInfo).

Serial numbers aren't necessarily restricted to 32-bit quantities. For example the RSADSI Commercial Certification Authority serial number is 0x0241000016, which is larger than 32 bits, and Verisign seem to like using 128 or 160-bit hashes as serial numbers. If you're writing certificate-handling code, just treat the serial number as a blob which happens to be an encoded integer (this is particularly important for the case of the vendors who have forgotten that the high bit of an integer is the sign bit, and generate negative serial numbers for their certificates).

## Signature

This rather misnamed field contains the algorithm identifier for the signature algorithm used by the CA to sign the certificate. There doesn't seem to be much use for this field, although you should check that the algorithm identifier matches the one of the signature on the cert (if someone can forge the signature on the cert then they can also change the inner algorithm identifier, it's possible that this was included because of some obscure attack where someone who could convince (broken) signature algorithm A to produce the same signature value as (secure) algorithm B could change the outer, unprotected algorithm identifier from B to A, but couldn't change the inner identifier without invalidating the signature. What this would achieve is unclear).

Be very careful with your use of Object Identifiers. In many cases there are a great many OIDs available for the same algorithm, but the exact OID you're supposed to use varies somewhat.

You see, the conditional modifiers depend on certain variables like the day of the week, the number of players, chair positions, things like that. [...] There can't be more than a dozen or two that are pertinent.  
-- Robert Asprin, "Little Myth Marker"

Your best bet is to copy the OIDs everyone else uses and/or use the RSADSI or X9 OIDs (rather than the OSI or OIW or any other type of OID). OTOH if you want to be proprietary while still pretending to follow a standard, use OSI OIDs which are often underspecified, so you can do pretty much whatever you want with things like block formatting and padding.

Another pitfall to be aware of is that algorithms which have no parameters have this specified as a NULL value rather than omitting the parameters field entirely. The reason for this is that when the 1988 syntax for AlgorithmIdentifier was translated into the 1997 syntax, the OPTIONAL associated with the AlgorithmIdentifier parameters got lost. Later it was recovered via a defect report, but by then everyone thought that algorithm parameters were mandatory. Because of this the algorithm parameters should be specified as NULL, regardless of what you read elsewhere.

The trouble is that things \*never\* get better, they just stay the same, only more so  
-- Terry Pratchett, "Eric"

## Name

```
Name ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

AttributeValueAssertion ::= SEQUENCE
{
    attributeType          OBJECT IDENTIFIER,
    attributeValue         ANY
}
```

This is used to encode that wonderful ISO creation, the Distinguished Name (DN), a path through an X.500 directory information tree (DIT) which uniquely identifies everything on earth. Although the RelativeDistinguishedName (RDN) is given as a SET OF AttributeValueAssertion (AVA) each set should only contain one element. However you may encounter other people's certs which could contain more than one AVA per set, there has been a reported sighting of a certificate which contained more than one element in the SET.

When the X.500 revolution comes, your name will be lined up against the wall and shot  
-- John Gilmore

They can't be read, written, assigned, or routed. Other than that, they're perfect  
-- Marshall Rose

When encoding sets with cardinality > 1, you need to take care to follow the DER rules which say that they should be ordered by their encoded values (although ASN.1 says a SET is unordered, the DER adds ordering

rules to ensure it can be encoded in an unambiguous manner). What you need to do is encode each value in the set, then sort them by the encoded values, and output them wrapped up in the SET OF encoding,

First things first, but not necessarily in that order.  
-- Dr. Who

however your software really shouldn't be producing these sorts of RDN entries.

In theory you don't have to use a Name for the subject name if you don't want to; there is a subjectAltName extension which allows use of email addresses or URL's. In theory if you want to do this you can make the Name an empty sequence and include a subjectAltName extension and mark it critical, but this will break a lot of implementations. Because it is possible to do this, you should be prepared to accept a zero-length sequence for the subject name in version 3 certificates. Since the DN is supposed to encode the location of the certificate in a DIT, having a null issuer name would mean you couldn't actually locate the certificate, so CAs will need to use proper DNs. The S/MIME certificate spec codifies this by requiring that all issuer DNs be non- null (so only an end-user certificate can have a null DN, and even then it's not really recommended), and this requirement was back-ported to the PKIX profile shortly before it was finalised. The reason for requiring issuer DNs is that S/MIME v2 and several related standards identify certificates by issuer and serial number, so all CA certificates must contain an issuer DN (S/MIME v3 allows subjectKeyIdentifiers, but they're almost never used).

SET provides an eminently sensible definition for DNs:

```
Name ::= SEQUENCE SIZE(1..5) OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET SIZE(1) OF AttributeTypeAndValue

AttributeTypeAndValue ::= { OID, C | O | OU | CN }
```

This means that when you see a SET DN it'll be in a fixed, consistent, and easy-to-process format (note in particular the fixed maximum size, the requirement for a single element per AVA, and the restriction to sensible element types).

Note that the (issuer name, serialNumber (with a possible side order of issuerUniqueID, issuerAltName, and keyUsage extension)) tuple uniquely identifies a certificate and can be used as a key to retrieve certificates from an information store. The subject name alone does not uniquely identify a certificate because a subject can own multiple certificates.

You would normally expect to find the following types of AVAs in an X.509 certificate, starting from the top:

```
countryName           ::= SEQUENCE { { 2 5 4 6 }, StringType(SIZE( 2 )) }
organization          ::= SEQUENCE { { 2 5 4 10 }, StringType(SIZE( 1..64 )) }
organizationalUnitName ::= SEQUENCE { { 2 5 4 11 }, StringType(SIZE( 1..64 )) }
commonName            ::= SEQUENCE { { 2 5 4 3 }, StringType(SIZE( 1..64 )) }
```

You might also find:

```
localityName          ::= SEQUENCE { { 2 5 4 7 }, StringType(SIZE( 1..64 )) }
stateOrProvinceName  ::= SEQUENCE { { 2 5 4 8 }, StringType(SIZE( 1..64 )) }
```

Some profiles require at least some of these AVAs to be present, for example the German profile requires at least a countryName and commonName, and in some cases also an organization name. This is a reasonable requirement, as a minimum you should always include the country and common name.

Finally, you'll frequently also run into:

```
emailAddress          ::= SEQUENCE { { 1 2 840 113549 1 9 1 }, IA5String }
```

from PKCS #9, although this shouldn't be there.

## X.509 Style Guide

I can't afford to make exceptions. Once word leaks out that a pirate has gone soft, people begin to disobey you and it's nothing but work, work, work all the time  
-- The Dread Pirate Roberts, "The Princess Bride"

The reason why oddball components like the emailAddress have no place in a DN created as per the original X.500 vision is because the whole DN is intended to be a strictly hierarchical construction specifying a path through a DIT. Unfortunately the practice adopted by many CAs of tacking on an emailAddress, an element which has no subordinate relationship to the other components of the DN, creates a meaningless mishmash which doesn't follow this hierarchical model. For this reason the ITU defined the GeneralName, which specifically allows for components such as email addresses, URL's, and other non-DN items. GeneralNames are discussed in "Extensions" below.

Since the GeneralName provides a proper means of specifying information like email addresses, your software shouldn't populate DNs with these components, however for compatibility with legacy implementations you need to be able to accept existing certificates which contain odd things in the DN. Currently all mailers appear to be able to handle an rfc822Name in an altName, so storing it in the correct location shouldn't present any interoperability problems. One problem with email address handling is that many mailers will accept not only 'J. Random Luser <jrandom@aol.com>' as a valid emailAddress/rfc822Name but will be equally happy with 'President William Jefferson Clinton <jrandom@aol.com>'. The former is simply invalid, but the latter can be downright dangerous because it completely bypasses the stated purpose of email certificates, which is to identify the other party in an email exchange. Both PKIX and S/MIME explicitly require that an rfc822Name only contain an RFC 822 addr-spec which is defined as local-part@domain, so the mailbox form 'Personal Name <local-part@domain>' isn't allowed (many S/MIME implementations don't enforce this though). Unfortunately X.509v3 just requires "an Internet electronic mail address defined in accordance with Internet RFC 822" without tying it down any further, so it could be either an addr-spec or a mailbox.

Okay, I'm going home to drink moderately and then pass out.  
-- Steve Rhoades, "Married with Children"

The countryName is the ISO 3166 code for the country. Noone seems to know how to specify non-country-aligned organisations, it's possible that 'EU' will be used at some point but there isn't any way to encode a non-country code although some organisations have tried using 'INT'. Actually noone really even knows what a countryName is supposed to refer to (let alone something as ambiguous as "locality"), for example it could be your place of birth, country of citizenship, country of current residence, country of incorporation, country where corporate HQ is located, country of choice for tax and/or jurisdictional issues, or a number of other possibilities (moving from countryName to stateOrProvinceName, people in the US military can choose a state as their official "residence" for tax purposes even if they don't own any property in that state, and politicians are allowed to run for office in one state while their wives claim residence and run for office in another state).

The details of the StringType are discussed further down. It's a good idea to actually limit the string lengths to 64 characters as required by X.520 because, although many implementations will accept longer encoded strings in certs, some can't manipulate them once they've been decoded by the software, and you'll run into problems with LDAP as well. This means residents of places like

TaumatawhakatangiHangakoauotamateaturipukakapikimaungahoronukupokai-whenuakitanataha

are out of luck when it comes to getting X.509 certs.

Comparing two DNs has its own special problems, and is dealt with in the rather lengthy "Comparing DNs" section below.

There appears to be some confusion about what format a Name in a certificate should take.

Insufficient facts always invite danger  
-- Spock, "Space Seed"

## X.509 Style Guide

In theory it should be a full, proper DN, which traces a path through the X.500 DIT, eg:

```
C=US, L=Area 51, O=Hanger 18, OU=X.500 Standards Designers, CN=John Doe
```

but since the DIT's usually don't exist, exactly what format the DN should take seems open to debate. A good guideline to follow is to organize the namespace around the C, O, OU, and CN attribute types, but this is directed primarily at corporate structures. You may also need to use ST(ate) and L(ocality) RDNs. Some implementations seem to let you stuff anything with an OID into a DN, which is not good.

There is nothing in any of these standards that would prevent me from including a 1 gigabit MPEG movie of me playing with my cat as one of the RDN components of the DN in my certificate.  
-- Bob Jueneman on IETF-PKIX

Note: There is a certificate of this form available from [http://www.cs.auckland.ac.nz/~pgut001/pubs/{dave\\_ca|dave}.der](http://www.cs.auckland.ac.nz/~pgut001/pubs/{dave_ca|dave}.der), although the MPEG is limited to just over 1MB.

With a number of organisations moving towards the use of LDAP-based directory services, it may be that we'll actually see X.500 directories in our lifetime,

Well, it just so happens that your friend here is only mostly dead. There's a big difference between mostly dead and all dead. Now, mostly dead is slightly alive.  
-- Miracle Max, "The Princess Bride"

which means you should make an attempt to have a valid DN in the certificate. LDAP uses the RFC 1779 form of DN, which is the opposite endianness to the ISO 9594 form used above:

```
CN=John Doe, OU=X.500 Standards Designers, O=Hanger 18, L=Area 51, C=US
```

There are always alternatives  
-- Spock, "The Galileo Seven"

In order to work with LDAP implementations, you should ensure you only have a single AVA per RDN (which also avoids the abovementioned DER-encoding hassle).

As the above text has probably indicated, DNs don't really work - there's no clear idea of what they should look like, most users don't know about (and don't want to know about) X.500 and its naming conventions, and as a consequence of this the DN can end up containing just about anything. At the moment they seem to be heading in two main directions:

- Public CAs typically set C=CA country, O=CA name, OU=certificate type, CN=user name
- A small subset of CAs in Europe which issue certs in accordance with various signature laws and profiles with their own peculiar requirements can have all sorts of oddities in the DN. You won't run into many of these in the wild.
- A small subsets of CAs will modify the DN by adding a unique ID value to the CN to make it a truly Distinguished Name. See the Bugs and Peculiarities sections for more information on this.
- Private CAs (mostly people or organisations signing their own certs) typically set any DN fields supported by their software to whatever makes sense for them (some software requires all fields in the set {C,O,OU,SP,L,CN} to be filled in, leading to strange or meaningless entries as people try and guess what a Locality is supposed to be).

Generally you'll only run into certs from public CAs, for which the general rule is that the cert is identified by the CN and/or email address. Some CAs issue certs with identical CN's and use the email address to disambiguate them, others modify the CN to make it unique. The accepted user interface seems to be to let users search on the CN and/or email address (and sometimes also the serial number, which doesn't seem terribly useful), display a list of matches, and let the user pick the cert they want.

Probably the best strategy for a user interface which handles certs is:

```

if( email address known )
    get a cert which matches the email address (any one should do);
elseif( name known )
    search for all certs with CN=name;
    if( multiple matches )
        display email addresses for matched certs to user, let them choose;
else
    error;

```

If you need something unique to use as an identifier (for example for a database key) and you know your own software (or more generally software which can do something useful with the identifier) will be used, use an X.500 serialNumber in a subjectAltName directoryName or use a subjectAltName otherName (which was explicitly created to allow user-defined identifiers). For internal cert lookups, encode the cert issuer and serial number as a PKCS #7 issuerAndSerialNumber, hash it down to a fixed size with SHA-1 (you can either use the full 20 bytes or some convenient truncated form like 64 bits), and use that to identify the cert. This works because the internal structure of the DN is irrelevant anyway, and having a fixed-size unique value makes it very easy to perform a lookup in various data structures (for example the random hash value generated leads to probabilistically balanced search trees without requiring any extra effort).

## Validity

```

Validity ::= SEQUENCE
{
    notBefore          UTCTIME,
    notAfter           UTCTIME
}

```

This field denotes the time at which you have to pay your CA a renewal fee to get the certificate reissued. The IETF originally recommended that all times be expressed in GMT and seconds not be encoded, giving:

```
YYMMDDHHMMZ
```

as the time encoding. This provided an unambiguous encoding because a value of 00 seconds was never encoded, which meant that if you read a UTCTime value generated by an implementation which didn't use seconds and wrote it out again with an implementation which did, it would have the same encoding because the 00 wouldn't be encoded.

However newer standards (starting with the Defence Messaging System (DMS), SDN.706), require the format to be:

```
YYMMDDHHMSSZ
```

even if the seconds are 00. The ASN.1 encoding rules were in late 1996 amended so that seconds are always encoded, with a special note that midnight is encoded as ...000000Z and not ...240000Z. You should therefore be prepared to encounter UTCTimes with and without the final 00 seconds field, however all newer certificates encode 00 seconds. If you read and then write out an existing object you may need to remember whether the seconds were encoded or not in the original because adding the 00 will invalidate the signature (this problem is slowly disappearing as pre-00 certificates expire).

A good workaround for this problem when generating certificates is to ensure that you never generate a certificate with the seconds set to 00, which means that even if other software re-encodes your certificate, it can't get the encoding wrong.

At least one widely-used product generated incorrect non-GMT encodings so you may want to consider handling the "+/-xxxx" time offset format, but you should flag it as a decoding error nonetheless.

In coming up with the worlds least efficient machine-readable time encoding format, the ISO nevertheless decided to forgo the encoding of centuries, a problem which has been kludged around by redefining the time as UTCTime if the date is 2049 or ealier, and GeneralizedTime if the date is 2050 or later (the original plan was to cut over in 2015, but it was felt that moving it back to 2050 would ensure that the designers were either retired or dead by the time the issue became a serious problem, leaving someone else to take the blame). To decode a date, if it's UTCTime and the year is less than or equal to 49 it's 20xx, if it's UTCTime and the year is equal to or greater than 50 it's 19xx, and if it's GeneralizedTime it's encoded properly (but shouldn't really be used for dates before 2050 because you could run into interoperability problems with existing software). Yuck.

To make this even more fun, another spec at one time gave the cutover date as 2050/2051 rather than 2049/2050, and allowed GeneralizedTime to be used before 2050 if you felt you could get away with it. It's likely that a lot of conforming systems will briefly become nonconforming systems in about half a centuries time, in a kind of security-standards equivalent of the age-old paradox in which Christians and Moslems will end up in the other side's version of hell.

Confusion now hath made his masterpiece.  
-- Macduff, "Macbeth", II. i.

Another issue to be aware of is the problem of issuer certificates which have a different validity time than the subject certificates they are used to sign. Although this isn't specified in any standard, some software requires validity period nesting, in which the subject validity period lies inside the issuer validity period. Most software however performs simple pointwise checking in which it checks whether a cert chain is valid at a certain point in time (typically the current time). Maintaining the validity nesting requires that a certain amount of care be used in designing overlapping validity periods between successive generations of certificates in a hierarchy. Further complications arise when an existing CA is re-rooted or re-parented (for example a divisional CA is subordinated to a corporate CA). Australian and New Zealand readers will appreciate the significance of using the term "re-rooted" to describe this operation.

Finally, CAs are handling the problem of expiring certificates by reissuing current ones with the same name and key but different validity periods. In some cases even CA roots have been reissued with the only different being extended validity periods. This can result in multiple identical-seeming certificates being valid at one time (in one case three certificates with the same DN and key were valid at once). The semantics of these certificates/keys are unknown. Perhaps Validity could simply be renamed to RenewalFeeDueDate to reflect its actual usage.

An alternative way to avoid expiry problems is to give the certificate an expiry date several decades in the future. This is popular for CA certs which don't require an annual renewal fee.

### ***SubjectPublicKeyInfo***

This contains the public key, either a SEQUENCE of values or a single INTEGER. Keep in mind that ASN.1 integers are signed, so if any integers you want to encode have the high bit set you need to add a single zero octet to the start of the encoded value to ensure that the high bit isn't mistaken for a sign bit. In addition you are allowed at most a single 0 byte at the start of an encoded value (and that only when the high bit is set), if the internal representation you use contains zero bytes at the start you have to remove them on encoding. This is a bit of a nuisance when encoding signatures which have INTEGER values, since you can't tell how big the encoded signature will be without actually generating it.

### ***UniqueIdentifier***

```
UniqueIdentifier ::= BITSTRING
```

These were added in X.509v2 to handle the possible reuse of subject and/or issuer names over time. Their use is deprecated by the IETF, so you shouldn't generate these in your certificates. If you're writing certificate-handling code, just treat them as a blob which happens to be an encoded bitstring.

## Extensions

```
Extensions ::= SEQUENCE OF Extension

Extension ::= SEQUENCE
{
    extnid                OBJECT IDENTIFIER,
    critical              BOOLEAN DEFAULT FALSE,
    extnValue             OCTETSTRING
}
```

X.509 certificate extensions are like a LISP property list: an ISO-standardised place to store cruffties. Extensions can consist of key and policy information, certificate subject and issuer attributes, certificate path constraints, CRL distribution points, and private extensions.

X.509v3 and the X.509v4 draft contains the ASN.1 formats for the standard V3 Certificate, V2 CRL and V2 CRLentry extensions. In theory you should be able to handle all of these, but there are large numbers of them and some may not be in active use, or may be meaningless in some contexts.

'It's called a shovel,' said the Senior Wrangler. 'I've seen the gardeners use them. You stick the sharp end in the ground. Then it gets a bit technical'  
-- Terry Pratchett, "Reaper Man"

The extensions are encoded with IMPLICIT tags, it's traditional to specify this in some other part of the standard which is at least 20 pages away from the section where the extension is actually defined (but see the comments above about the mixture of explicit and implicit tags in ASN.1 definitions).

There are a whole series of superseded and deprecated OIDs for extensions, often going back through several generations. Older software and certificates (and buggy newer software) will still use obsolete OIDs, any new software should try and emit attributes tagged with the OID du jour rather than using deprecated OIDs.

We can break extensions into two types, constraint extensions and informational extensions. Constraint extensions limit the way in which the key in a certificate, or the certificate itself, can be used. For example they may limit the key usage to digital signatures only, or limit the DNs for which a CA may issue certificates. The most common constraint extensions are basic constraints, key usage and extended key usage, certificate policies (modified by policy mappings and policy constraints), and name constraints. In contrast, informational extensions contain information which may or may not be useful for certificate users, but which doesn't limit the certificate use in any way. For example an informational extension may contain additional information which identifies the CA which issued it. The most common informational extensions are key identifiers and alternative names.

The processing of these extensions is mostly specified in three different standards, which means that there are three often subtly incompatible ways to handle them. In theory, constraint extensions should be enforced religiously, however the three standards which cover certificates sometimes differ both in how they specify the interpretation of the critical flag, and how they require constraint extensions to be enforced.

We could not get it out of our minds that some subtle but profoundly alien element had been added to the aesthetic feeling behind the technique.  
-- H. P. Lovecraft, "At the Mountains of Madness"

The general idea behind the critical flag is that it is used to protect the issuing CA against any assumptions made by software which doesn't implement support for a particular extension (none of the X.509-related standards provide much of a definition for what a minimally, average, and fully compliant implementation needs to support, so it's something of a hit and miss proposition for an implementation to rely on the correct handling of a particular extension). One commentator has compared the various certificate constraints as serving as the equivalent of a Miranda warning ("You have the right to remain silent, you have the right to an attorney, ...") to

anyone using the certificate. Without the critical flag, an issuer who believes that the information contained in an extension is particularly important has no real defence if the end users software chooses to ignore the extension.

The original X.509v3 specification requires that a certificate be regarded as invalid if an unrecognised critical extension is encountered. As for the extension itself, if it's non-critical you can use whatever interpretation you choose (that is, the extension is specified as being of an advisory nature only). This means that if you encounter constraints which require that a key be used only for digital signatures, you're free to use it for encryption anyway. If you encounter a key which is marked as being a non-CA key, you can use it as a CA key anyway. The X.509v3 interpretation of extensions is a bit like the recommended 130 km/h speed limit on autobahns, the theoretical limit is 130, you're sitting there doing 180, and you're getting overtaken by Porsches doing about 250. The problem with the original X.509v3 definitions is that although they specify the action to take when you don't recognise an extension, they don't really define the action when you do recognise it. Using this interpretation, it's mostly pointless including non-critical extensions because everyone is free to ignore them (for example the text for the keyUsage extension says that "it is an advisory field and does not imply that usage of the key is restricted to the purpose indicated", which means that the main message it conveys is "I want to bloat up the certificate unnecessarily").

The second interpretation of extensions comes from the IETF PKIX profile. Like X.509v3, this also requires that a certificate be regarded as invalid if an unrecognised critical extension is encountered. However it seems to imply that a critical extension must be processed, and probably considers non-critical extensions to be advisory only. Unfortunately the wording is ambiguous enough that a number of interpretations exist. Section 4.2 says that "CAs are required to support <constraint extensions>", but the degree of support is left open, and what non-CAs are supposed to do isn't specified. The paragraph which follows this says that implementations "shall recognise extensions", which doesn't imply any requirement to actually act on what you recognise. Even the term "process" is somewhat vague, since processing an extension can consist of popping up a warning dialog with a message which may or may not make sense to the user, with an optional "Don't display this warning again" checkbox. In this case the application certainly recognised the extension and arguably even processed it, but it didn't force compliance with the intent of the extension, which was probably what was intended by the terms "recognise" and "process".

The third interpretation comes from S/MIME, which requires that implementations correctly handle a subset of the constraint and informational extensions. However, as with PKIX, "correctly handle" isn't specified, so it's possible to "correctly handle" an extension as per X.509v3, as per PKIX (choose the interpretation you prefer), or as per S/MIME, which leaves the issue open (it specifies that implementations may include various bits and pieces in their extensions, but not how they should be enforced). S/MIME seems to place a slightly different interpretation on the critical flag, limiting its use to the small subset of extensions which are mentioned in the S/MIME spec, so it's not possible to add other critical extensions to an S/MIME certificate.

"But it izz written!" bellowed Beelzebub.

"But it might be written differently somewhere else" said Crowley. "Where you can't read it".

"In bigger letters" said Aziraphale.

"Underlined" Crowley added.

"Twice" suggested Aziraphale.

-- Neil Gaiman and Terry Pratchett, "Good Omens"

Finally, the waters are further muddied by CA policies, which can add their own spin to the above interpretations. For example the Verisign CPS, section 2.4.3, says that "all persons shall process the extension [...] or else ignore the extension", which would seem to cover all the bases. Other policies are somewhat more specific, for example Netscapes certificate extension specification says that the keyUsage extension can be ignored if it's not marked critical, but Netscape Navigator does appear to enforce the basicConstraints extension in most cases.

The whole issue is complicated by the fact that implementations from a large vendor will reject a certificate which contains critical constraint extensions, so that even if you interpret the critical flag to mean "this extension must be enforced" (rather than just "reject this certificate if you don't recognise the extension"), you can't use it because it will render the certificate unusable. These implementations provide yet another interpretation of the critical flag, "reject this certificate if you encounter a critical extension". The same vendor also has software which ignores the critical flag entirely, making the software essentially useless to relying parties who can't rely on

it to perform as required (the exact behaviour depends on the software and version, so one version might reject a certificate with a critical extension while another would ignore a critical extension).

Zaphod stared at him as if expecting a cuckoo to leap out of his forehead on a small spring.  
-- Douglas Adams, "The Restaurant at the End of the Universe"

Because of this confusion, it's probably impossible to include a set of constraint extensions in a certificate which will be handled properly by different implementations. Because of problems like this, the digital signature laws of some countries are requiring certification of the software being used as part of compliance with the law, so that you can't just claim that your software "supports X.509v3 certificates" (everyone claims this whether they actually do or not), you actually have to prove that it supports what's required by the particular countries' laws. If you're in a country which has digital signature legislation, make sure the software you're using has been certified to conform to the legal requirements.

The best interpretation of constraint extensions is that if a certificate is marked as an X.509v3 certificate, constraints should always be enforced. This includes enforcing implied settings if the extension is missing, so that a certificate being used in a CA role which has no basicConstraints extension present should be regarded as being invalid (note however the problem with PKIX-compliant certificates described later on). However even if one of the standards is reworded to precisely define extension handling, there are still plenty of other standards and interpretations which can be used. The only solution to this would be to include a critical policy extension which requires that all constraint extensions up and down the cert chain be enforced. Going back to the autobahn analogy, this mirrors the situation at the Austrian border, where everyone slows down to the strictly enforced speed limit as soon as they cross the border.

Currently the only way to include a constraint enforcement extension is to make it a critical policy extension. This is somewhat unfortunate since including some other random policy may make the extension unrecognisable, causing it, and the entire certificate, to be rejected (as usual, what constitutes an unrecognisable extension is open to debate: if you can process all the fields in an extension but don't recognise the contents of one of the fields, it's up to you whether you count this as being unrecognisable or not).

A better alternative would be to define a new extension, enforceConstraints:

```
enforceConstraints EXTENSION ::=
{
  SYNTAX EnforceConstraintsSyntax
  IDENTIFIED BY id-ce-enforceConstraints
}

EnforceConstraintsSyntax ::= BOOLEAN DEFAULT FALSE
```

This makes the default setting compatible with the current "do whatever you feel like" enforcement of extensions. Enforcing constraints is defined as enforcing all constraints contained in constraint extensions, including implied settings if the extension is missing, as part of the certificate chain validation process (which means that they should be enforced up and down the cert chain). Recognising/supporting/handling/<whatever other wording is used in standards> an extension is defined as processing and acting on all components of all fields of an extension in a manner which is compliant with the semantic intent of the extension.

'Where was I?' said Zaphod Beeblebrox the Fourth.  
'Pontificating' said Zaphod Beeblebrox.  
'Oh yes'.  
-- Douglas Adams, "The Restaurant at the End of the Universe"

Just to mention a further complication with critical extensions, there are instances in which it's possible to create certificates which are always regarded as being invalid due to conflicts with extensions. For example a generation n-1 critical extension might be replaced by a generation n critical extension, resulting in a mixture of certs with generation n-1 extensions, generation n-1 and generation n extensions (for compatibility) and (eventually) generation n extensions only. However until every piece of software is upgraded, generation n-1

software will be forced to reject all certs with generation n extensions, even the (supposedly) backwards-compatible certs with both generations of extension in them.

'Mr. Beeblebrox, sir', said the insect in awed wonder, 'you're so weird you should be in movies'.  
-- Douglas Adams, "The Restaurant at the End of the Universe"

### **Key Usage, Extended Key Usage, and Netscape cert-type**

X.509 and PKIX use `keyUsage` and `extKeyUsage` to select the key to use from a selection of keys unless the extension is marked critical, in which case it's treated as a usage restriction. Microsoft claims to support key usage enforcement, although experimentation with implementations has shown that it's mostly ignored (see the entry on Microsoft bugs further on). In addition if an `extKeyUsage` extension is present, all certificates in the chain up to the CA root must also support the same `extKeyUsage` (so that, for example, a general-purpose CA can't sign a server gated crypto certificate - the reasoning behind this is obvious). As it turns out though, `extKeyUsage` seems to be mostly ignored just like `keyUsage`.

Netscape uses `keyUsage` as a key selection mechanism, and uses the Netscape cert-type extension in a complex manner described in the Netscape certificate extension specification. Since the cert-type extension includes the equivalent of the basicConstraints CA flag, it's possible to specify some types of CA with the cert-type extension. If you do this, you should be careful to synchronise the basicConstraints CA flag with the setting of the cert-type extension because some implementations (you can probably guess which one) will allow a Netscape CA-like usage to override a non-CA `keyUsage` value, treating the certificate as if it were a CA certificate. In addition Netscape also enforces the same `extKeyUsage` chaining as Microsoft.

Unfortunately the `extKeyUsage` chaining interpretation is wrong according to PKIX, since the settings apply to the key in the certificate (ie the CA's key) rather than the keys in the certificates it issues. In other words an `extKeyUsage` of `emailProtection` would indicate that the CA's certificate is intended for S/MIME encryption, not that the CA can issue S/MIME certificates. Both of the major implementors of certificate-handling software use the chaining interpretation, but there also exist implementations which use the PKIX interpretation, so the two main camps will fail to verify the other side's cert chains unless they're in the (smaller) third camp which just ignores `extKeyUsage`.

For `keyUsage` there is much disagreement over the use of the `digitalSignature` and `nonRepudiation` bits since there was no clear definition in X.509 of when the `nonRepudiation` flag should be used alongside or in place of the `digitalSignature` flag. One school of thought holds that `digitalSignature` should be used for ephemeral authentication (something which occurs automatically and frequently) and `nonRepudiation` for legally binding long-term signatures (something which is performed consciously and less frequently). Another school of thought holds that `nonRepudiation` should act as an additional function for the `digitalSignature` mechanism, with `digitalSignature` being a mechanism bit and `nonRepudiation` being a service bit. The different profiles are split roughly 50:50 on this, with some complicating things by specifying that both bits should be set but the certificate not be used for one or the other purpose. Probably the best usage is to use `digitalSignature` for "digital signature for authentication purposes" and `nonRepudiation` for "digital signature for nonrepudiation purposes".

"I think" said the Metatron, "that I shall need to seek further instructions".  
"I alzzo" said Beelzebub.  
-- Neil Gaiman and Terry Pratchett, "Good Omens"

In terms of profiles, MISSI and FPKI follow the above recommendation, PKIX uses `nonRepudiation` strictly for nonrepudiation and `digitalSignature` for everything else, ISO uses `digitalSignature` for entity authentication and `nonRepudiation` strictly for nonrepudiation (leaving digital signatures for data authentication without nonrepudiation hanging), and others use something in between. When this issue was debated on PKI lists in mid-1998, over 100 messages were exchanged without anyone really being able to uncontestably define what `digitalSignature` and `nonRepudiation` really signified. The issue is further confused by the fact that noone can agree on what the term "nonRepudiation" actually means, exemplified by a ~200-message debate in mid-1999 which couldn't reach any useful conclusion.

He had attached the correct colour-coded wires to the correct pins; he'd checked that it was the right amperage fuse; he'd screwed it all back together. So far, no problems. He plugged it into the socket. Then he switched the socket on. Every light in the house went out.  
-- Neil Gaiman and Terry Pratchett, "Good Omens"

Although everyone has their own interpretation, a good practical definition is "Nonrepudiation is anything which fails to go away when you stop believing in it". Put another way, if you can convince a user that it isn't worth trying to repudiate a signature then you have nonrepudiation. This can take the form of having them sign a legal agreement saying they won't try to repudiate any of their signatures, giving them a smart card and convincing them that it's so secure that any attempt to repudiate a signature generated with it would be futile, threatening to kill their kids, or any other method which has the desired effect. One advantage (for vendors) is that you can advertise just about anything as providing nonrepudiation, since there's sure to be some definition which matches whatever it is you're doing (there are "nonrepudiation" schemes in use today which employ a MAC using a secret shared between the signer and the verifier, which must be relying on a particularly creative definition of nonrepudiation).

Bei ihnen auf dem Server muesste irgendwie ein Key rumliegen, den ich mit Netscape vermutlich erzeugt hab. Wenn da mein Name drin steht, dann wird er das schon sein. Koennten sie mir den zertifizieren?  
-- endergone Zwiebeltuete

One might as well add a "crimeFree" (CF) bit with usage specified as 'The crimeFree bit is asserted when subject public key is used to verify digital signatures for transactions that are not a perpetration of fraud or other illegal activities'  
-- Tony Bartoletti on ietf-pkix

I did have the idea that we mandate that CAs MUST set this bit randomly whenever a keyUsage extension is present, just to stop people who argue that its absence has a meaning.  
-- Stephen Farrell on ietf-pkix

## ***Basic Constraints***

This is used to specify whether a certificate is a CA certificate or not. You should always mark this critical, because otherwise some implementations will ignore it and allow a non-CA certificate to act as a CA.

## ***Alternative Names***

The subject and issuer alternative names are used to specify all the things which aren't suitable for a DN, which for most purposes means practically everything of any use on the Internet (X.509v3 defines the alternative names (or, more specifically, the GeneralName type) for use in specifying identifying information which isn't suited for, or part of, a DN). This includes email addresses, URL's for web pages, server addresses, and other odds and ends like X.400 and EDI addresses. There's also a facility to include your postal address, physical address, phone, fax and pager numbers, and of course the essential MPEG of your cat.

The alternative names can be used for certificate identification in place of the DNs, however the exact usage is somewhat unclear. In particular if an altName is used for certificate chaining purposes, there's a roughly 50/50 split of opinion as to whether all the items in the altName must match or any one of the items must match. For example if an altName contains three URL's in the issuer and one in the client (which matches one of the issuer URL's), noone really knows whether this counts as a valid altName match or not. Eventually someone will make a decision one way or the other, probably the S/MIME standards group who are rather reliant on altNames for some things (the S/MIME group have requested that the PKIX group make DNs mandatory in order to allow proper certificate chaining, and the S/MIME specs themselves require DNs for CAs). Until this is sorted out, it's not a good idea to rely on altNames for chaining.

This confusion is caused by the fact that an altName is serving two conflicting purposes. The first of these is to provide extra information on the certificate owner which can't be specified in the DN, including things like their phone number, email address, and physical address. The second is to provide an alternative to the ITU-managed (or, strictly speaking, non-managed) DN space. For example a DNS name or IP address, which falls outside the range of ITU (non-)management, is controlled by the IETF, which has jurisdiction over the name space of the Internet-related altName components. However since an altName can only specify a collection of names with a single critical attribute to cover all of them, there's no way to differentiate between something which really is critical (for example an rfc822Name being used in place of DN) and something which is merely present to provide extra details on the certificate owner (an rfc822Name being provided as a contact address). One IETF draft overloaded this even further by forcing authorityInfoAccess semantics onto some of the altName components.

This ambiguity is complicated by the presence of other attributes like path processing constraints. For example does an included or excluded subtree constraint containing a DN cover the subjectName DN, the altName directoryName, or both?. More seriously, since a subjectName and altName directoryName have the same form, it's possible to specify an identical DN in two different ways across an issuer and subject cert, leading to the same problem described below in the name constraints section in which it's possible to evade constraint checks by using alternative encodings for the same item.

The solution to this problem would be to split the altName into two new extensions, a true altName which provides a single alternative to the subjectName (for example a single dNSName or rfc822Name) and which is used only when the subject DN is empty, and a collection of other information about the subject which follows the current altName syntax but which is used strictly for informational purposes. The true altName provides a single alternative for the subjectName, and the informational altName provides any extra identification information which the subject may want to include with their certificate.

A different (and much uglier) solution is to try and stuff everything imaginable into a DN. The problem with this approach is that it completely destroys any hope of interoperability with directories, especially X.500 directories which rely on search arguments being predefined as a type of filter. Unless you have this predefined filter, you can't easily search the directory for a match, so it's necessary to have some limits placed on the types of names (or schemas in X.500-speak) which are acceptable. Unfortunately the "stuff everything into a DN" approach violates this principle, making the result un-searchable within a directory, which voids the reason for having the DN in the first place.

### ***Certificate Policies and Policy Mappings and Constraints***

The general idea behind the certificate policies extension is simple enough, it provides a means for a CA to identify which policy a certificate was issued under. This means that when you check a certificate, you can ensure that each certificate in the chain was issued under a policy you feel comfortable with (certain security precautions taken, vetting of employees, physical security of the premises, no loud ties, that sort of thing). The certificatePolicies extension (in its minimal form) provides a means of identifying the policy a certificate was issued under, and the policyMappings extension provides a means of mapping one policy to another (that is, for a CA to indicate that policy A, under which it is issuing a certificate, is equivalent to policy B, which is required by the certificate user).

Unfortunately on top of this there are qualifiers for the certificatePolicies and the policyConstraints extension to muddy the waters. As a result, a certificate policy often consists of a sequence of things identified by unknown object identifiers, each with another sequence of things identified by partially known, partially unknown object identifiers, with optional extra attachments containing references to other things which software is expected to know about by magic or possibly some form of quantum tunnelling.

Marx Brothers fans will possibly recall the scene in "A Day at the Races" in which Groucho, intending to put his money on Sun-up, is induced instead to buy a coded tip from Chico and is able to establish the identity of the horse only, at some cost in terms of time and money, by successive purchases from Chico of the code book, the master code book, the breeders' guide and various other works of reference, by the end of which the race is over, Sun-up having won.

-- Unknown, forwarded by Ed Gerck to cert-talk

This makes it rather difficult to make validity decisions for a certificate which have anything more complex than a basic policyIdentifier present.

Because of this, you should only use a single policyIdentifier in a certificate, and forgo the use of policy qualifiers and other odds and ends. Currently noone but Verisign appears to use these, the presence of these qualifiers in the PKIX profile may be related to the presence of Verisign in the PKIX profiling process.

## **Name Constraints**

The name constraints are used to constrain the certificates' DN to lie inside or outside a particular subtree, with the excludedSubtrees field taking precedence over the permittedSubtrees field. The principal use for this extension is to allow balkanization of the certificate namespace, so that a CA can restrict the ability of any CAs it certifies to issue certificates outside a very restricted domain.

Unfortunately if the X.500 string encoding rules are followed, it's always possible to avoid the excludedSubtrees by choosing unusual (but perfectly valid) string encodings which don't appear to match the excludedSubtrees (see the section on string encodings for more details on this problem). Although PKIX constrains the string encodings to allow the nameConstraints to function, it's a good idea to rely on permittedSubtrees rather than excludedSubtrees for constraint enforcement (actually since virtually nothing supports this extension, it's probably not a good idea to rely too much on either type of constraint, but when it is supported, use permitted rather than excluded subtrees).

## **Subject and Authority Key Identifiers**

These are used to provide additional identification information for a certificate. Unfortunately it's specified in a somewhat complex manner which requires additional ASN.1 constraints or text to explain it, you should treat it as if it were specified with the almost-ASN.1 of:

```
AuthorityKeyIdentifier ::= CHOICE
{
    keyIdentifier [ 0 ] OCTET STRING,
    authorityCert [ 1 ] GeneralNames, authoritySerialNumber [ 2 ] INTEGER
}
```

X.509v3 allows you to use both types of identifier, but other standards and profiles either recommend against this or explicitly disallow it, allowing only the keyIdentifier. Various profiles have at various times required the use of the SHA-1 hash of the public key (whatever that constitutes), the SHA-1 hash of the subjectPublicKeyInfo data (for some reason this has to be done *\*without\** the tag and length at the start), the SHA-1 hash of the subjectPublicKey (again without the tag, length, and unused bits portion of the BIT STRING, which leaves just the raw public key data but omits the algorithm identifier and parameters so that two keys for different algorithms with different parameters which happen to share the same public key field end up with the same hash), a 64-bit hash of the subjectPublicKeyInfo (presumably with the tag and length), a 60-bit hash of the subjectPublicKey (again with tag and length) with the first four bits set to various values to indicate MISSI key types, and some sort of unique value such as a monotonically increasing sequence. Several newer profiles have pretty much given up on this and simply specify "a unique value". RFC 2459 also allows "a monotonically increasing sequence of integers", which is a rather bad move since the overall supply of unique small integers is somewhat limited and this scheme will break as soon as a second CA decides to issue a cert with a "unique" subjectKeyIdentifier of the same value.

To balance the problems caused by this mess of conflicting and incompatible standards, it appears that most implementations either ignore the keyIdentifier entirely or don't bother decoding it, because in 1997 and 1998 a widely-used CA accidentally issued certificates with an incorrect encoding of the keyIdentifier (it wasn't even

valid ASN.1 data, let alone X.509-conformant ASN.1) without anyone noticing. Although a few standards require that a keyIdentifier be used, its absence doesn't seem to cause any problems for current implementations.

### **Recommendation**

Don't even try and decode the authorityKeyIdentifier field, just treat everything inside the OCTET STRING hole as an opaque blob. Given that most current implementations seem to ignore this extension, don't create certificate chains which require it to be processed in order for the chaining to work.

The claimed reason for using the keyIdentifier rather than the issuerAndSerialNumber is because it allows a certificate chain to be re-rooted when an intermediate CA changes in some manner (for example when its responsibilities are handed off from one department in an organisation to another). If the certificate is identified through the keyIdentifier, no nameConstraints are present, the certificate policies are identical or mapped from one to the other, the altNames chain correctly, and no policyConstraints are present, then this type of re-rooting is possible (in case anyone missed the sarcasm in there, the gist is that it's highly unlikely to work).

### **Other Extensions**

There are a wide variety of other extensions defined in various profiles. These are in effect proprietary extensions because unless you can track down something which recognises them (typically a single-vendor or small-group-of-vendors product), you won't be able to do much with them - most software will either ignore them completely or reject the certificate if the extension is marked critical and the software behaves as required. Unless you can mandate the use of a given piece of certificate-processing software which you've carefully tested to ensure it processes the extension correctly, and you can block the use of all other software, you shouldn't rely on these extensions. Obviously if you're in a closed, carefully controlled environment (for example a closed shop EDI environment which requires the use of certain extensions such as reliance limits) the use of specialised extensions isn't a problem, but otherwise you're on your own.

In addition to the use of other people's extensions, you may feel the need to define your own. In short if you're someone other than Microsoft (who can enforce the acceptance of whatever extensions they feel like), don't do it. Not only is it going to be practically impossible to find anything to support them (unless you write it yourself), but it's also very easy to stuff all sorts of irrelevant, unnecessary, and often downright dangerous information into certificates without thinking about it. The canonical example of something which has no place in a certificate is Microsoft's cKeyCertIndexPair extension, which records the state of the CA software running on a Windows 2000 machine at the time the certificate was generated (in other words it offloads the CA backup task from the machine's administrator to anyone using one of the certificates).

Only wimps use tape backup: `_real_` men just upload their important stuff on ftp, and let the rest of the world mirror it.  
-- Linus Torvalds

The canonical example of a dangerous certificate extension is one which indicates whether the owner is of legal age for some purpose (buying alcohol/driving/entry into nightclubs/whatever). Using something like a drivers license for this creates a booming demand for forged licenses which, by their very nature, are difficult to create and tied to an individual through a photo ID. Doing the same thing with a certificate creates a demand for those over the age limit to make their keys (trivially copied en masse and not tied to an individual) available to those under the age limit, or for those under the age limit to avail themselves of the keys in a surreptitious manner. The fact that the borrowed key which is being used to buy beer or rent a porn movie can also be used to sign a legally binding contract or empty a bank account probably won't be of concern until it's too late. This is a good example of the law of unintended consequences in action.

If scientists can be counted on for anything, it's for creating unintended consequences.  
-- Michael Dougan

A related concern about age indicators in certificates, which was one of the many nails in X.500's coffin, is the fact that giving a third party the information needed to query a certificate directory in order to locate, for example, all teenage girls in your localityName, probably wouldn't be seen as a feature by most certificate holders. Similar

## X.509 Style Guide

objections were made to the use of titles in DNs, for example a search on a title of "Ms" would have allowed someone to locate all single women in their localityName, and full-blown X.500 would have provided their home addresses and probably phone numbers to boot. Until early 1999 this type of extension only existed as a hypothetical case, but it's now present as a mandatory requirement in at least one digital signature law, which also has as a requirement that all CAs publish their certificates in some form of openly-accessible directory.

I saw, and I heard an eagle, flying in mid heaven, saying with a loud voice, "Woe! Woe! Woe for those who dwell on the earth"

-- Revelations 8:15

## Character Sets

Character strings are used in various places (most notably in DNS), and are encumbered by the fact that ASN.1 defines a whole series of odd subsets of ASCII/ISO 646 as character string types, but only provides a few peculiar and strange oddball character encodings for anything outside this limited character range.

The protruding upper halves of the letters now appear to read, in the local language, "Go stick your head in a pig", and are no longer illuminated, except at times of special celebration.  
-- Douglas Adams, "The Restaurant at the End of the Universe"

To use the example of DNS, the allowed string types are:

```
DirectoryString ::= CHOICE
{
    teletexString          TeletexString (SIZE (1..maxSize)),
    printableString       PrintableString (SIZE (1..maxSize)),
    bmpString              BMPString (SIZE (1..maxSize)),
    universalString        UniversalString (SIZE (1..maxSize))
}
```

The easiest one to use, if you can get away with it, is IA5String, which is basically 7-bit ASCII (including all the control codes), but with the dollar sign potentially replaced with a "currency symbol". A more sensible alternative is VisibleString (aka ISO646String), which is IA5String without the control codes (this has the advantage that you can't use it to construct macro viruses using ANSI control sequences). In the DirectoryString case, you have to make do with PrintableString, which is one of the odd ASCII/ISO 646 subsets (for example you can't encode an '@', which makes it rather challenging to encode email addresses).

Beyond that there is the T.61/TeletexString, which can select different character sets using escape codes (this is one of the aforementioned "peculiar and strange oddball encodings"). The character sets are Japanese Kanji (JIS C 6226-1983, set No.87), Chinese (GB 2312-80, set No.58), and Greek, using shifting codes specified in T.61 or the international version, ISO 6937 (strictly speaking T61String isn't defined in T.61 but in X.680, which defines it by profiling ISO 2022 character set switching). Some of the characters have a variable-length encoding (so it takes 2 bytes to encode a character, with the interpretation being done in a context-specific manner). The problem isn't helped by the fact that the T.61 specification has changed over the years as new character sets were added, and since the T.61 spec has now been withdrawn by the ITU there's no real way to find out exactly what is supposed to be in there (but see the previous comment on T.61 vs T61String - a T61String isn't really a T.61 string). Even using straight 8859-1 in a T61String doesn't always work, for example the 8859-1 character code for the Norwegian OE (slashed O) is defined using a T.61 escape sequence which, if present in a certificate, may cause a directory to reject the certificate.

And then there came the crowning horror of all - the unbelievable, unthinkable, almost unmentionable thing.  
-- H. P. Lovecraft, "The Statement of Randolph Carter"

For those who haven't reached for a sick bag yet, one definition of T61String is given in ISO 1990 X.208 which indicates that it contains registered character sets 87, 102 (a minimalist version of ASCII), 103 (a character set with the infamous "floating diacritics" which means things like accented characters are encoded as "<add an accent to the next character> + <character>" rather than with a single character code), 106 and 107 (two useless sets containing control characters which noone would put in a name), SPACE + DELETE. The newer ITU-T 1997 and ISO 1998 X.680 adds the character sets 6, 126, 144, 150, 153, 156, 164, 165, and 168 (the reason for some of these additions is because once a character set is registered it can never change except by "clarifying" it, which produces a completely new character set with a new number (as with sex, once you make a mistake you end up having to support it for the rest of your life)). In fact there are even more definitions of T61String than that: The original CCITT 1984 ASN.1 spec defined T61String by reference to a real T.61 recommendation (from which finding the actual permitted characters is challenging, to put it mildly), then the ISO 1986 spec defined them by reference to the international register, then the CCITT 1988 spec changed them again (the ISO 1990

spec described above may be identical to the CCITT 1988 one), and finally they were changed again for ISO/ITU-T 1994 (this 1994 spec may again be the same as ITU-T 1997 and ISO 1998). I'm not making this up!

The disciples came to him privately, saying, "Tell us, what is the sign of your coming, and of the end of the world?" [...] "You will hear of wars and rumors of wars; there will be famines, plagues, and earthquakes in various places; the sun will be darkened, the moon will not give her light, the stars will fall from the sky, the powers of the heavens will be shaken; certificates will be issued with floating diacritics in their DNs; and then the end will come".

-- Matthew 24 (mostly)

The encoding for this mess is specified in X.209 which indicates that the default character sets at the start of a string are 102, 106 and 107, although in theory you can't really make this assumption without the appropriate escape sequences to invoke the correct character set. The general consensus among the X.500/ISODE directory crowd is that you assume that set 103 is used by default, although Microsoft and Netscape had other ideas for their LDAPv2 products. In certificates, the common practice seems to be to use straight latin-1, which is set numbers 6 and 100, the latter not even being an allowed T61String set.

There are those who will say Danforth and I were utterly mad not to flee for our lives after that; since our conclusions were now completely fixed, and of a nature I need not even mention to those who have read my account so far.

-- H. P. Lovecraft, "At the Mountains of Madness"

Next are the BMPString and UniversalString, with BMPString having 16-bit characters (UCS-2) and UniversalString having 32-bit characters (UCS-4), both encoded in big-endian format. BMPString is a subset of UniversalString, being the 16-bit character range in the 0/0 plane (ie the UniversalString characters in which the 16 high bits are 0), corresponding to straight ISO 10646/Unicode characters. The ASN.1 standard says that UniversalString should only be used if the encoding possibilities are constrained, it's better to avoid it entirely and only use BMPString/ISO 10646/Unicode.

However, there is a problem with this: at the moment few implementors know how to handle or encode BMPStrings, and people have made all sorts of guesses as to how Unicode strings should be encoded: with or without Unicode byte order marks (BOMs), possibly with a fixed endianness, and with or without the terminating null character.

I might as well be frank in stating what we saw; though at the time we felt that it was not to be admitted even to each other. The words reaching the reader can never even suggest the awfulness of the sight itself.

-- H. P. Lovecraft, "At the Mountains of Madness"

The correct format for BMPStrings is: big-endian 16-bit characters, no Unicode byte order marks (BOMs), and no terminating null character (ISO 8825-1 section 8.20).

An exception to this is PFX/PKCS #12, where the passwords are converted to a Unicode BMPString before being hashed. However both Netscape and Microsoft's early implementations treated the terminating null characters as being part of the string, so the PKCS #12 standard was retroengineered to specify that the null characters be included in the string.

A final string type which is presently only in the PKIX profile but which should eventually appear elsewhere is UTF-8, which provides a means of encoding 7, 8, 16, and 32-bit characters into a single character string. Since ASN.1 already provides character string types which cover everything except some of the really weird 32-bit characters which noone ever uses,

It was covered in symbols that only eight other people in the world would have been able to comprehend; two of them had won Nobel prizes, and one of the other six dribbled a lot and wasn't allowed anything sharp because of what he might do with it.

-- Neil Gaiman and Terry Pratchett, "Good Omens"

the least general encoding rule means that UTF-8 strings will practically never be used. The original reason they were present in the PKIX profile is because of an IETF rule which required that all new IETF standards support UTF-8, but a much more compelling argument which recently emerged is that, since most of the other ASN.1 character sets are completely unusable, UTF-8 would finally breathe a bit of sanity into the ASN.1 character set nightmare. Unfortunately, because it's quite a task to find ASN.1 compilers (let alone certificate handling software) which supports UTF-8, you should avoid this string type for now. PKIX realised the problems which would arise and specified a cutover date of 1 January 2004 for UTF-8 use. Some drafts have appeared which recommend the use of RFC 2482 language tags, but these should be avoided since they have little value (they're only needed for machine processing, if they appear in a text string intended to be read by a human they'll either understand it or they won't and a language tag won't help). In addition UTF-8 language tags are huge (about 30 bytes) due to the fact that they're located out in plane 14 in the character set (although I don't have the appropriate reference to hand, plane 14 is probably either Gehenna or Acheron), so the tag would be much larger than the string being tagged in most cases.

One final problem with UTF-8 is that it shares some of the T.61 string problems in which it's possible for a malicious encoder to evade checks on strings either by using different code points which produce identical-looking characters when displayed or by using suboptimal encodings (in ASN.1 terms, non-distinguished encodings) of a code point. They are aided in this by the standard, which says (page 47, section 3.8 of the Unicode 3.0 standard) that "when converting from UTF-8 to a Unicode scalar value, implementations do not need to check that the shortest encoding is being used. This simplifies the conversion algorithm". What this means is that it's possible to encode a particular character in a dozen different ways in order to evade a check which uses a straight byte-by-byte comparison as specified in RFC 2459. Although some libraries such as glibc 2.2 use "safe" UTF-8 decoders which will reject non-distinguished encodings, it's not a good idea to assume that everyone does this.

Because of these problems, the SET designers produced their own alternative, SETString, for places where DNs weren't required for compatibility purposes. The design goals for the SETString were to both provide the best coverage of ASCII and national-language character sets, and also to minimise implementation pain. The SETString type is defined as:

```
SETString ::= CHOICE
{
    visibleString      VisibleString (SIZE (1. . maxSize)),
    bmpString          BMPString (SIZE (1. . maxSize))
}
```

This provides complete ASCII/ISO 646 support using single byte characters, and national language support through Unicode, which is in common use by industry.

In addition the SET designers decided to create their own version of the DirectoryString which is a proper subset of the X.500 version. The initial version was just an X.500 DirectoryString with a number of constraints applied to it, but just before publication this was changed to:

```
DirectoryString ::= CHOICE
{
    printableString    PrintableString (SIZE(1. . maxSize)),
    bmpString          BMPString (SIZE(1. . maxSize))
}
```

You must unlearn what you have learned.  
-- Yoda

It was felt that this improved readability and interoperability (and sanity). T61String was never seriously considered in the design, and UniversalString with its four-byte characters had no identifiable industry support and required too much overhead. If you want to produce certs which work for both generic X.509 and SET, then using the SET version of the DirectoryString is a good idea. It's trivial to convert an ISO 8859-1 T61String to a BMPString and back (just add/subtract a 0 byte every other byte).

MISSI also subsets the string types, allowing only PrintableString and T61String in DNs.

When dealing with these character sets you should use the "least inclusive" set when trying to determine which encoding to use. This means trying to encode as PrintableString first, then T61String, and finally BMPString/UniversalString. SET requires that either PrintableStrings or BMPStrings be used, with TeletexStrings and UniversalStrings being forbidden.

From this we can build the following set of recommendations:

- Use PrintableString if possible (or VisibleString or IA5String if this is allowed, because it's rather more useful than PrintableString).
- If you use a T61String (and assuming you don't require SET compliance), avoid the use of anything involving shifting and escape codes at any cost and just treat it as a pure ISO 8859-1 string. If you need anything other than 8859-1, use a BMPString.
- If it won't go into one of the above, try for a BMPString.
- Avoid UniversalStrings.

Version 7 of the PKIX draft dropped the use of T61String altogether (probably in response to this writeup :-), but this may be a bit extreme since the extremely limited character range allowed by PrintableString will result in many simple strings blowing out to BMPStrings, which causes problems on a number of systems which have little Unicode support.

In 2004, you can switch to UTF-8 strings and forget about this entire section of the guide.

I saw coming out of the mouth of the dragon, and out of the mouth of the beast, and out of the mouth of the false prophet, three unclean spirits, something like frogs; for they are spirits of demons, performing signs

-- Biblical explanation of the origins of character set problems,  
Revelations 16:13-14, recommended rendition: Diamanda Galas, The Plague Mass.

## Comparing DNs

This is an issue which is so problematic that it requires a section of its own to cover it fully. According to X.500, to compare a DN:

- The number of RDNs must match.
- RDNs must have the same number of AVAs.
- Corresponding AVAs must match for equality:
  - Leading and trailing spaces are ignored.
  - Multiple internal spaces are treated as a single internal space.
  - Characters (not code points, which are a particular encoding of a character) are matched in a case-insensitive manner.

As it turns out, this matching is virtually impossible to perform (more specifically, it is virtually impossible to accurately compare two DNs for equivalence).

This, many claim, is not merely impossible but clearly insane, which is why the advertising executives of the star system of Bastablon came up with this quote: 'If you've done six impossible things this morning, why not round it off with breakfast at Milliways, the Restaurant at the End of the Universe?'

-- Douglas Adams, "The Restaurant at the End of the Universe"

The reason for this is that, with the vast number of character sets, encodings, and alternative encodings (code points) for the same character, and the often very limited support for non-ASCII character sets available on many systems, it isn't possible to accurately and portably compare any RDNs other than those containing one of the basic ASCII string types. The best you can probably do is to use the strategy outlined below.

First, check whether the number of RDNs is equal. If they match, break up the DNs into RDNs and check that the RDN types match. If they also match, you need to compare the text in each RDN in turn. This is where it gets tricky.

He walked across to the window and suddenly stumbled because at that moment his Joo-Janta 200 Super-Chromatic Peril Sensitive sunglasses had turned utterly black.

-- Douglas Adams, "The Restaurant at the End of the Universe"

First, take both strings and convert them to either ASCII (ISO646String) or Unicode (BMPString) using the "least inclusive" rule. This is quite a task in itself, because several implementations aren't too picky about what they'll put into a string, and will stuff T61String characters into a PrintableString, or (more commonly) Unicode characters into a T61String or anything into a BMPString. Finding a T61String in a PrintableString or an 8-bit character set string in a BMPString is relatively easy, but the best guess you can take at detecting a Unicode string in a T61String is to check whether either the string length is odd or all the characters are ASCII or ASCII with the high bit set. If neither are true, it might be a Unicode string disguised as a T61String.

Once this is done, you need to canonicalise the strings into a format in which a comparison can be done, either to compare strings of different types (eg 8-bit character set or DBCS string to BMPString) or strings with the same type but different encodings (eg two T61Strings using alternative encodings). To convert ASCII-as-Unicode to ASCII, just drop the odd-numbered bytes. Converting a T61String to Unicode is a bit more tricky. Under Windows 95 and NT, you can use MultiByteToWideChar(), although the conversion will depend on the current code page in use. On systems with widechar support, you can use mbstowcs() directly if the widechar size is the same as the BMPString char size (which it generally isn't), otherwise you need to first convert the string to a widechar string with mbstowcs() and then back down again to a BMPString, taking the machine endianness into account. Again, the behaviour of mbstowcs() will depend on the current locale in use. If the system doesn't have widechar support, the best you can do is a brute-force conversion to Unicode by hoping it's ISO 8859-1 and adding a zero byte every other byte.

Now that you might have the strings in a format where they can be compared, you can actually try and compare them. Again, this often ends up as pure guesswork if the system doesn't support the necessary character sets,

or if the characters use weird encodings which result in identical characters being located at different code points.

First, check the converted character sets: If one is Unicode and the other isn't, then the strings probably differ (depending on how well the canonicalisation step went). If the types are the same, strip leading, trailing, and repeated internal spaces from the string, which isn't as easy as it sounds since there are several possible code points allocated to a space.

Once you've had a go at stripping spaces, you can try to compare the strings. If the string is a BMPString then under Windows NT (but not Windows 95) you can use CompareString(), with the usual caveat that the comparison depends on the current locale. On systems which support towlower() you can extract the characters from the string into widechars (taking machine endianness into account) and compare the towlower() forms, with the usual caveat about locale and the added problem that towlower() implementations vary from bare-bones (8859-1 only under Solaris, HPUX, AIX) to vague (Unicode under Win95, OSF/1). If there's no support for towlower(), the best you can do is use the normal tolower() if the characters have a high byte of zero (some systems will support 8859-1 for tolower(), the worst which can happen is that the characters will be returned unchanged), and compare the code points directly if it isn't an 8-bit value.

Zaphods skin was crawling all over his body as if it was trying to get off.  
-- Douglas Adams, "The Restaurant at the End of the Universe"

Finally, if it's an ASCII string, you can just use a standard case-insensitive string comparison function.

As you can see, there's almost no way to reliably compare two RDN elements. In particular, no matter what you do:

- Some malicious users will deliberately pass DN checks with weird encodings.
- Some normal users will accidentally fail DN checks with weird encodings.

This becomes an issue when certain security checks depend on a comparison of DN's (for example with excluded subtrees in the Name Constraints extension) because it's possible to create multiple strings which are displayed identically to the user (especially if you know which platform and/or software to target) assuming they get displayed at all, but which compare as different strings. If you want to be absolutely certain about DN comparisons, you might need to set a certificate policy of only allowing PrintableStrings in DN's, because they're the only ones which can be reliably compared.

## PKCS #10

According to the PKCS #10 spec, the attributes field is mandatory, so if it's empty it's encoded as a zero-length field. The example however assumes that if there are no attributes, the field shouldn't be present, treating it like an OPTIONAL field. A number of vendors conform to the example rather than the specification, but just to confuse the issue RSADSI, who produced PKCS #10, regard things the other way around, with the spec being right and the example being wrong. The most obvious effect of this is that TIPEM (which was the only available toolkit for PKCS#10 for a long time) follows the spec and does it "wrong (sense #2)", whereas more recent independant implementations follow the example and do it "wrong (sense #1)".

Unfortunately it's difficult to handle certificate requests correctly and be lenient on decoding. Because a request could be reencoded into DER before checking the signature, funny things can happen to your request at the remote end if the two interpretations of PKCS #10 differ. Because of this confusion, you should be prepared to accept either version when decoding, but at the moment it's not possible to provide any recommendation for encoding. When encountering a particularly fascist parser which isn't aware of the PKCS #10 duality, it may be necessary to submit two versions of the request to determine which one works.

No, no. Yes. No, I tried that. Yes, both ways. No, I don't know. No again. Are there any more questions?  
 -- Xena, "Been There, Done That"

PKCS #10 also dates from the days of X.509v1 and includes references to obsolete and deprecated objects and data formats. PKCS #6 extended certificates are a workaround for the absence of certificate extensions in X.509v1 and shouldn't be used at all, and it's probably a good idea to avoid the use of PKCS #9 extended attributes as well (some certification request protocols bypass the use of PKCS #9 by wrapping extra protocol layers containing PKCS #9 elements around the outside of PKCS #10). Instead, you should use of the CMMF draft, which defines a new attribute identified by the OID {pkcs-9 14}, with a value of SEQUENCE OF Extension which allows X.509v3 attributes to be encoded into a PKCS #10 certification request. The complete encoding used to encode X.509v3 extensions into a PKCS #10 certification request is therefore:

```
[0] IMPLICIT SET OF
{
  SEQUENCE                                -- attributes from PKCS #10
  {
    OBJECT IDENTIFIER,                    -- Attribute from X.501
    SET OF                                -- type, {pkcs-9 14}
    {
      SEQUENCE OF                          -- values
      {
        <X.509v3 extensions>              -- ExtensionReq from CMMF draft
      }
    }
  }
}
```

As of late 1998, virtually all CAs ignore this information and at best add a few predefined extensions based on the options selected on the web page which was used to trigger the certification process. There are one or two implementations which do support it, and these provide a convenient means of specifying attributes for a certificate which don't involve kludges via HTML requests. Microsoft started supporting something like it in mid-1999, although they used their own incompatible OID in place of the PKCS #9 one to ensure non-compatibility with any other implementation (the extensions are encoded in the standard format, they're just identified in a way which means nothing else can read them).

Unfortunately since PKCS #10 doesn't mention X.509v3 at all, there's no clear indication of what is and isn't valid as an attribute for X.509v3, but common sense should indicate what you can and can't use. For example a subjectAltName should be treated as a valid attribute, a basicConstraint may or may not be treated as valid depending on the CA's certification policy, and an authorityKeyIdentifier would definitely be an invalid attribute.

## X.509 Style Guide

SET provides its own version of PKCS #10 which uses a slightly different encoding to the above and handles the X.509v3 extensions `keyUsage`, `privateKeyUsagePeriod` (whose use is deprecated by PKIX for some reason), `subjectAltName`, and the SET extensions `certificateType`, `tunneling`, and `additionalPolicy`. Correctly handling the SET extensions while at the same time avoiding Microsoft's broken extensions which look very similar (see the "Known Bugs/Peculiarities" section) provides a particularly entertaining exercise for implementors.

## ASN.1 Design Guidelines

This section contains some guidelines on what I consider good ASN.1 style. This was motivated both by the apparent lack of such guidelines in existing documents covering ASN.1, and by my seeing the ASN.1 definition of the X.400 ORAddress (Originator/Recipient Address). Although there are a number of documents which explain how to use ASN.1, there doesn't seem to be much around on ASN.1 style, or at least nothing which is easily accessible. Because of this I've noted down a few guidelines on good ASN.1 style, tuned towards the kind of ASN.1 elements which are used in certificate-related work. In most cases I'll use the X.400 ORAddress as an example of bad style (I usually use PFX for this since it's such a target-rich environment, but in this case I'll make an exception). The whole ORAddress definition is far too long to include here (it requires pages and pages of definitions just to allow the encoding of the equivalent of an email address), but I'll include excerpts where required.

If you can't be a good example, then you'll just have to be a horrible warning.  
-- Catherine Aird

To start off, keep your structure as simple as possible. Everyone always says this, but when working with ASN.1 it's particularly important because the notation gives you the freedom to design incredibly complicated structures which are almost impossible to work with.

Bud, if dynamite was dangerous do you think they'd sell it to an idiot like me?  
-- Al Bundy, "Married with Children"

Look at the whole ORAddress for an example.

What we did see was something altogether different, and immeasurably more hideous and detestable. It was the utter, objective embodiment of the fantastic novelists 'thing that should not be'.  
-- H. P. Lovecraft, "At the Mountains of Madness"

This includes provisions for every imaginable type of field and element which anyone could conceivably want to include in an address. Now although it's easy enough to feed the whole thing into an ASN.1 compiler and produce an enormous chunk of code which encodes and decodes the whole mess, it's still necessary to manually generate the code to interpret the semantic intent of the content. This is a complex and error-prone process which isn't helped by the fact that the structure contains dozens of little-understood and rarely-used fields, all of which need to be handled correctly by a compliant implementation. Given the difficulty of even agreeing on the usage of common fields in certificate extensions, the problems which will be raised by obscure fields buried fifteen levels deep in some definition aren't hard to imagine.

ASN.1 \*WHAM\* is not \*WHAM\* COBOL \*WHAM\* \*WHAM\* \*WHAM\*. The whole point of an abstract syntax notation is that it's not tied to any particular representation of the underlying data types. An extreme example of reverse-engineering data type dependency back into ASN.1 is X9.42's:

```
OCTET STRING SIZE(4)  -- (Big-endian) Integer
```

Artificially restricting an ASN.1 element to fall within the particular limitations of the hardware you're using creates all sorts of problems for other users, and for the future when people decide that 640K isn't all the memory anyone will ever need. The entire point of ASN.1 is that it not be tied to a particular implementation, and that users not have to worry about the underlying data types. It can also create ambiguous encodings which void the DER guarantee of identical encodings for identical values: Although the ANSI/SET provision for handling currencies which may be present in amounts greater than 10e300 (requiring the amtExp10 field to extend the range of the ASN.1 INTEGER in the amount field) is laudable, it leads to nondeterministic encodings in which a single value can be represented in a multitude of ways, making it impossible to produce a guaranteed, repeatable encoding. Careful with that tagging Eugene! In recent ASN.1 work it seems to have become fashionable to madly tag everything which isn't nailed down, sometimes two or three times recursively for good measure (see the next point).

The entire set of PDU's are defined using an incredible amount of gratuitous and unnecessary tagging. Were the authors being paid by the tag for this?  
 -- Peter Gutmann on ietf-pkix

For example consider the following ORAddress ExtensionAttribute:

```
ExtensionAttribute ::= SEQUENCE
{
  extension-attribute-type [0] INTEGER,
  extension-attribute-value [1] ANY DEFINED BY extension-attribute-type
}
```

(this uses the 1988 ASN.1 syntax, more recent versions change this somewhat). Both of the tags are completely unnecessary, and do nothing apart from complicating the encoding and decoding process. Another example of this problem are extensions like authorityKeyIdentifier, cRLDistributionPoints, and issuingDistributionPoint which, after several levels of nesting, have every element in a sequence individually tagged even though, since they're all distinct, there's no need for any of the tags.

Another type of tagging is used for the ORAddress BuiltInStandardAttributes:

```
BuiltInStandardAttributes ::= SEQUENCE
{
  countryName [APPLICATION 1] CHOICE { ... } OPTIONAL,
  ...
}
```

Note the strange nonstandard tagging - even if there's a need to tag this element (there isn't), the tag should be a context-specific tag and not an application-specific one (this particular definition mixes context-specific and application-specific tags apparently at random). For tagging fields in sequences or sets, you should always use context-specific tags.

Speaking of sequences and sets, if you want to specify a collection of items in data which will be signed or otherwise authenticated, use a SEQUENCE rather than a SET, since the encoding of sets causes serious problems under the DER. You can see the effect of this in newer PKCS #7 revisions, which substitute SEQUENCE almost everywhere where the older versions used a SET because it's far easier to work with the former even though what's actually being represented is really a SET and not a SEQUENCE.

If you have optional elements in a sequence, it's always possible to eliminate the tag on the first element (provided it's not itself tagged), since it can be uniquely decoded without the tag. For example consider privateKeyUsagePeriod:

```
PrivateKeyUsagePeriod ::= SEQUENCE
{
  notBefore [ 0 ] GeneralizedTime OPTIONAL,
  notAfter [ 1 ] GeneralizedTime OPTIONAL
}
```

The first tag is unnecessary since it isn't required for the decoding, so it could be rewritten:

```
PrivateKeyUsagePeriod ::= SEQUENCE
{
  notBefore GeneralizedTime OPTIONAL,
  notAfter [ 0 ] GeneralizedTime OPTIONAL
}
```

saving an unneeded tag.

Because of the ability to specify arbitrarily nested and redefined elements in ASN.1, some of the redundancy built into a definition may not be immediately obvious. For example consider the use of a DN in an IssuingDistributionPoint extension, which begins:

## X.509 Style Guide

```
IssuingDistributionPoint ::= SEQUENCE
{
    distributionPoint [0] DistributionPointName OPTIONAL,
    ...
}

DistributionPointName ::= CHOICE
{
    fullName [0] GeneralNames,
    ...
}

GeneralNames ::= SEQUENCE OF GeneralName

GeneralName ::= CHOICE
{
    ...
    directoryName [4] Name,
    ...
}

Name ::= CHOICE
{
    rdnSequence RDNSequence
}

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeTypeAndValue
```

[It] was of a baroque monstrosity not often seen outside the Maximegalon Museum of Diseased Imaginings.

-- Douglas Adams, "The Restaurant at the End of the Universe"

Once we reach AttributeTypeAndValue we finally get to something which contains actual data - everything before that point is just wrapping.

Now consider a typical use of this extension, in which you encode the URL at which CA information is to be found. This is encoded as:

```
SEQUENCE { [0] { [0] { SEQUENCE { [6] "http://. . ." } } } }
```

All this just to specify a URL!

It looks like they were trying to stress-test their ASN.1 compilers.

-- Roger Schlafly on stds-p1363

It smelled like slow death in there, malaria, nightmares.

This was the end of the river alright.

-- Captain Willard, "Apocalypse Now"

Unfortunately because of the extremely broad definition used (a SEQUENCE OF GeneralName can encode arbitrary quantities of almost anything imaginable, for example you could include the contents of an entire X.500 directory in this extension), producing the code to correctly process every type of field and item which could occur is virtually impossible, and indeed the semantics for many types of usage are undefined (consider how you would use a physical delivery address or a fax number to access a web server).

Because of the potential for creating over-general definitions, once you've written down the definition in its full form, also write it out in the compressed form I've used above, and alongside this write down the encoded form

of some typical data. This will very quickly show up areas in which there's unnecessary tagging, nesting, and generality, as the above example does.

An extreme example of the misuse of nesting, tagging, and generality is the ORName, which when fully un-nested is:

```
ORName ::= [APPLICATION 0] SEQUENCE
{
  [0] { SEQUENCE OF SET OF AttributeTypeAndValue OPTIONAL }
}
```

(it's not even possible to write all of this on a single line). This uses unnecessary tagging, nonstandard tagging, and unnecessary nesting all in a single definition.

It will founder upon the rocks of iniquity and sink headfirst to vanish without trace into the seas of oblivion.

-- Neil Gaiman and Terry Pratchett, "Good Omens"

The actual effect of the above is pretty close to:

```
ORName = Anything
```

Another warning sign that you've constructed something which is too complex to be practical is the manner in which implementations handle its encoding. If you (or others) are treating portions of an object as a blob (without bothering to encode or decode the individual fields in it) then that's a sign that it's too complex to work with. An example of this is the policyQualifiers portion of the CertificatePolicies extension which, in the two implementations which have so far come to light which actually produce qualifiers, treat them as a fixed, opaque blob with none of the fields within it actually being encoded or decoded. In this case the entire collection of qualifiers could just as easily be replaced by a BOOLEAN DEFAULT FALSE to indicate whether they were there or not.

Another warning sign that something is too complex is when your definition requires dozens of paragraphs of accompanying text and/or extra constraint specifications to explain how the whole thing works or to constrain the usage to a subset of what's specified. If it requires four pages of explanatory text to indicate how something is meant to work, it's probably too complex for practical use.

No matter how grandiose, how well-planned, how apparently foolproof an evil plan, the inherent sinfulness will by definition rebound upon its instigators.

-- Neil Gaiman and Terry Pratchett, "Good Omens"

Finally, stick to standard elements and don't reinvent your own way of doing things. Taking the ORAddress again, it provides no less than three different incompatible ways of encoding a type-and-value combination for use in different parts of the ORAddress. The standard way of encoding this (again using the simpler 1988 syntax) is:

```
Attribute ::= SEQUENCE
{
  type OBJECT IDENTIFIER,
  value ANY DEFINED BY type
}
```

The standard syntax for field names is to use biCapitalised words, with the first letter in lowercase, for example:

```
md5WithRSAEncryption
certificateHold
permittedSubtrees
```

Let's take an example. Say you wanted to design an extension for yet another online certificate validation protocol which specifies a means of submitting a certificate validity check request. This is used so a certificate

user can query the certificate issuer about the status of the certificate they're using. A first attempt at this might be:

```
StatusCheck ::= SEQUENCE
{
    statusCheckLocations [0] GeneralNames
}
```

Eliminating the unnecessary nesting and tagging we get:

```
StatusCheck ::= GeneralNames
```

However taking a typical encoding (a URL) we see that it comes out as:

```
StatusCheck ::= SEQUENCE { [6] "http://..." }
```

In addition the use of a SEQUENCE OF GeneralName makes the whole thing far too vague to be useful (someone would be perfectly within their rights to specify a pigeon post address using this definition, and I don't even want to get into what it would require for an implementation to claim it could "process" this extension). Since it's an online check it only really makes sense to do it via HTTP (or at least something which can be specified through a URL), so we simplify it down again to:

```
StatusCheck ::= SEQUENCE OF IA5String -- Contains a URL
```

We've now reached an optimal way of specifying the status check which is easily understandable by anyone reading the definition, and doesn't require enormous amounts of additional explanatory text (what to do with the URL and how to handle the presence of multiple URL's is presumably specified as part of the status-check protocol - all we're interested in is how to specify the location(s) at which the status check is performed).

## base64 Encoding

Many programs allow certificate objects to be encoded using the base64 format popularised in PEM and MIME for transmission of binary data over text-only channels. The format for this is:

```
-----BEGIN <object name>-----
<base64-encoded data>
-----END <object name>-----
```

Unfortunately there is some disagreement over what <object name> should be for objects other than certificates (there's no standard for implemetations to be non-compliant with). Everyone seems to agree that for certificates it's 'CERTIFICATE' (SSLeay can also accept 'X.509 CERTIFICATE'). For certificate requests, it's generally 'NEW CERTIFICATE REQUEST', although SSLeay can also generate 'CERTIFICATE REQUEST' and Microsoft creates an undocumented blob which is nothing like a certificate request while still giving it the certificate request header. CRLs are so rare that I haven't been able to collect a large enough sample to get a consensus, but 'CRL' would seem to be the logical choice (SSLeay uses 'X.509 CRL', matching 'X.509 CERTIFICATE'). Finally, if you see 'PGP ...' then you've got the wrong kind of object.

The number of dashes around the text must be exactly five.

...then shalt thou count to three, no more, no less.

Three shalt be the number thou shalt count, and the number of the counting shalt be three. Four shalt thou not count, neither count thou two, excepting that thou then proceed to three. Five is right out.

-- Monty Python and the Holy Grail

There are three further object types which aren't covered yet, attribute certificates (which are too new to be used), and Netscape cert sequences and PKCS #7 cert chains (which are degenerate signed data objects). The logical choice for these would be 'ATTRIBUTE CERTIFICATE', 'NETSCAPE CERTIFICATE SEQUENCE' and 'PKCS7 CERTIFICATE CHAIN'.

### Recommendation

When encoding objects, for certificates use 'BEGIN CERTIFICATE', for attribute certificates use 'BEGIN ATTRIBUTE CERTIFICATE', for cert requests use 'BEGIN NEW CERTIFICATE REQUEST', for CRLs use 'BEGIN CRL', for Netscape certificate sequences use 'BEGIN NETSCAPE CERTIFICATE SEQUENCE', and for PKCS #7 certificate chains use 'BEGIN PKCS7 CERTIFICATE CHAIN'. When decoding objects, don't make any assumptions about what you might find for <object name> - it's easiest to just look for 'BEGIN' and then work out what's in there from the decoded object.

## Known Bugs/Peculiarities

The following list of issues cover problems and areas to be aware of in X.509 implementations and related data objects from different vendors. The coverage extends to objects related to X.509 such as private keys and encrypted/signed data. This section is not intended as a criticism of different vendors, it is merely an list of issues which people should be aware of when attempting to write interoperable software. If vendors or users are aware of fixes for these problems, could they please notify me of what was fixed, and when or in which version it occurred.

One general comment about certificates is that, although you are allowed to deconstruct them and then re-encode them, the fact that there are so many incorrectly encoded certificates around means that the re-encoded certificates will fail their signature check. For this reason it is strongly advised that you always keep a copy of the original on hand rather than trying to recreate it from its components as they are stored internally by your software.

### ***An Post***

An Post certificates include an enormously long (nearly four times the maximum allowed size) legal disclaimer in the certificate policy extension (the certificate contains as much legal disclaimer as all other data combined).

### ***Bankgate***

Bankgate certificates specify the country as "INT", which isn't a valid country name (presumably it's meant to be either "International" or "Internet", but as it's used now it means "Limbo").

### ***Belsign***

The Belsign CA incorrectly encodes PKCS #7 certificate chains using a zero-length set of certificates or CRLs if there are none present instead of omitting the field which should contain them. Since the field is declared as OPTIONAL, it should be omitted if it is empty.

### ***BTG***

BTG certificates contain incorrectly encoded bit strings in key/certificate usage extensions.

### ***CDSA***

CDSA uses a peculiar deprecated DSA OID which appeared in an early, incomplete version of SDN.702 but vanished in later versions (this OID also appears in the German PKI profile). CDSA also doesn't recognise the recommended X9.57 dsaWithSHA1 OID, causing it to reject DSA certificates which use it.

### ***CompuSource***

This CA has a root cert which has the UTCTime encoded without the seconds (see the section "Validity" above), newer versions encode the time with the seconds. This isn't an error since the accepted encoding was changed in mid-1996, merely something to be aware of.

### ***COST***

The lengths of some of the fields in CRLs are broken. Specifically, the lengths of some sequences are calculated incorrectly, so if your code merely nods at things like SET and SEQUENCE tags and lengths as they whiz past and then works with the individual fields it'll work, but if it tries to work with the length values given (for example making sure the lengths of the components of a sequence add up correctly) then it'll break. The sequence lengths are longer than the amount of data in the sequence, the COST code may be adding the lengths of the

elements of the sequence incorrectly (it's a bit hard to tell what's going wrong. Basically the CRLs are broken). This was supposed to have been fixed, but there still appear to be problems with CRLs (?).

## ***CRLs***

Some CRLs which contain extensions (which are only valid in v2 CRLs) are marked as v1 CRLs because they don't have a version number field present. These CRLs are (in theory) invalid, but your software should be prepared to encounter them.

## ***DAP***

X.500 directories using DAP return BER-encoded certificates since the DAP PDU's are BER encoded. This leads to many certificates failing their signature check when they are re-encoded using the DER because they use incorrect or unexpected encodings. This problem generally requires hacking the directory software to kludge the encoding, since many certificates can't survive the re-encoding process.

## ***Deutsches Forschungsnetz (DFN)***

The DFN CA used the SecuDE software until late-1998, see the SecuDE entry for the quirks present in certificates generated with this software. These certificates expired at the end of 1998, current certificates are generated using OpenSSL.

## ***Digicert***

(Based on the certificate peculiarities this CA uses the same software as Sweden Post, but some of the quirks of the Sweden Post certificates aren't present so it has its own entry)

The subjectKeyIdentifier is encoded in a variety of ways ranging in length from 1 to 8 bytes. In CA certs the subjectKeyIdentifier is the same as the authorityKeyIdentifier (which is valid, but odd) and consists of a text string identifying the CA key (this isn't explicitly disallowed because of the confusion over what's really meant to be in there but it isn't really what's supposed to be in there either).

CA certs include policy mappings which specify the same issuer and subject domain policy (this is known as a tautology mapping).

## ***Diginotar***

End entity certs tend to be issued with a keyUsage specifying every possible type of key usage, including ones which the algorithm being used is incapable of.

## ***Digital Signature Trust***

The certificate policy contains a longer-than-allowed legal disclaimer, although not quite as excessive as the An Post one. Just to make sure you don't miss it, the certificate includes the whole thing a second time as a Netscape comment extension.

End entity certs tend to be issued with a keyUsage specifying every possible type of key usage, including ones which the algorithm being used is incapable of (since this CA operates under the strict Utah law it's possible that this takes precedence over the inability of the RSA algorithm to perform Diffie- Hellman key agreement).

## ***Digitrust Hellas***

The Digitrst Hellas CA incorrectly encodes bit strings in key/certificate usage extensions.

## ***DNs with UniqueIDs***

Given that, in practice, Distinguished Names aren't, some organisations which really require unique names have tried to use unique identifiers as part of whatever's used as the DN in order to make it Distinguished. Unfortunately many applications can't handle multi-AVA RDNs, and those which can generally won't display them to the user, making the use of DN components like dnQualifiers impossible since all the user sees is a collection of certs which all appear to have the same DN. As a result, some organisations modify the DN by appending a unique identifier value to it. Examples of these organisations are the US DoD (a very large and highly distributed organisation which needs unique ID's) and AlphaTrust (which specialises in certificates used in transactions which are legally binding regardless of the state of digital signature legislation).

## ***Entrust***

This was formerly a Nortel division, for notes on earlier versions of the software see the entry for Nortel. Because of their origins, Entrust- specific extensions and attributes are still identified with NSN (Nortel Secure Network) object identifiers.

The Entrust demo web CA encodes liability statements in the issuer DN, making them unusable with X.500/LDAP directories. It also issues certificates with a zero-duration validity (start time == end time), limiting their usefulness somewhat.

Something identified as 'V3.0c' encodes the outer certificate using the BER instead of the DER (which is unusual but legal), however it also omits the final end-of-contents octets (which isn't). Some of the inner components are also encoded using the BER (which isn't allowed). This has been fixed in the 4.0 release.

The same software populates the certificate with multiple generations of extensions (for example it contains multiple copies of authorityKeyIdentifier, keyUsage, and cRLDistributionPoints extensions of different levels of deprecation). Luckily it doesn't mark any of its extensions as critical, avoiding the mutual-exclusion problem documented in the section on extensions.

The extension which identifies V3.0c contains a GeneralString, which is perfectly legal but may come as a considerable surprise to some decoding software (GeneralStrings get even weirder than the other ASN.1 types, and aren't used in anything certificate-related).

### ***Estonian CLO CA***

### ***Estonian National PCA***

### ***Estonian IOC CA***

### ***Estonian Piirivalveamet CA***

### ***Estonian Politsei CA***

These CAs identify RSA public keys in certificates by a peculiar deprecated X.500 OID for RSA (2 5 8 1 1).

The Estonian National CA encodes some DN components as PrintableString's containing illegal characters. I guess the Estonian ASN.1 Politsei will have to look at this.

These CAs appear to be using the same software, possibly SecuDE, so they may exhibit the same DN encoding bug as the Estonian National CA (I only have one cert from each so it's hard to check this).

## ***First Data***

First Data's CA root certificate is (according to the extKeyUsage extension) used for their SSL server, in SSL clients, and for email encryption.

## ***GeneralName***

Some implementations incorrectly encode the `iPAddress` (yes, it really is capitalised like that; ASN.1 is bigger than the Internet) as a dotted address rather than a 4-byte (soon to become 16 byte) OCTET STRING, so you might encounter "123.124.125.126" instead of the correct 0x7B7C7D7E.

## ***GIP-CPS***

The software used in the French healthcare card project incorrectly encodes `cRLDistributionPoints`, encoding the `fullName` as if it were specified with `[0] EXPLICIT GeneralNames`, so that the final encoding is `[0] + SEQUENCE + GeneralName` rather than `[0] + GeneralName`.

## ***GTE***

Older versions of GTE's software encoded `UTCTimes` without the seconds (see the section "Validity" above), newer versions encode the time with the seconds. This isn't an error since the accepted encoding was changed in mid-1996, merely something to be aware of.

## ***HBCI***

The German Home Banking Computer Interface specification contains some unusual requirements for certificates. Signatures are created by signing the raw, unpadded RIPEMD-160 hash of the data. Similarly, RSA encryption is performed by encrypting raw, unpadded content-encryption keys. This provides no semantic security (that is, it's trivial to determine whether a given plaintext corresponds to the ciphertext), and has other problems as well. The IEEE P1363 standard provides further thoughts on this issue.

## ***IBM***

IBM's web CA uses the same peculiar deprecated DSA OID as CDSA and JDK. Since it's based on IBM's Java CryptoFramework it probably ended up in the CA via its use in the JDK.

## ***ICE-TEL***

Early ICE-TEL CRLs are broken, consisting of various portions of full CRLs. See the entry for SECUDE for the explanation, this has been fixed in the ICE-TEL successor project ICE-CAR.

## ***IPSEC***

IPSEC more or less assumes the use of X.509 certificates, however the companies implementing it are usually in the VPN business rather than the PKI business and tend to see certificates as a means to an end rather than an end in itself. As a result, the state of certificate handling in IPSEC in mid-1999 is something of a free-for-all, with certificates containing whatever seems to work. For example some IPSEC implementations may place identification information in the `subjectName`, some ignore the `subjectName` and use the `altName`, and some use (even require) both. In general, you shouldn't make any assumptions about what you'll encounter in certificates designed for or created by IPSEC implementations.

## ***JDK/Java***

JDK 1.1 issues DSA certificates with a signature OID of `dsaWithSHA`, 1.3.14.3.2.13, but the hash algorithm used is SHA-1. The OID should be `dsaWithSHA1` 1.3.14.3.2.27. Since no one ever seems to use SHA, a workaround is to always assume SHA-1 even if the OID says SHA (which is also what the JDK folks are banking on). JDK also uses the peculiar deprecated DSA OID 1.3.14.3.2.12 which appeared in an early, incomplete version of SDN.702 but vanished in later versions (CDSA does this as well, God knows where they're getting these from). These strange OIDs are duplicated in other Java software as well. Apparently these OIDs arise from RSADS's BSAFE 3.0, which is the crypto toolkit which underlies many of these implementations.

## **Keywitness**

Keywitness encodes an email address as a PrintableString as part of a DN component with an unknown OID registered to Keywitness. This encoding is invalid since a PrintableString can't contain an '@' character.

Boolean values are encoded using a non-DER encoding.

## **LDAP V2/QUIPU**

Some implementations will do strange things when storing signed items. Either the client or the server may modify the objects (for example by uppercasing parts of DN's, or changing time fields based on their interpretation of UTC, or dropping seconds in time fields), which changes the resulting DER encoding and causes the signature check to fail.

## **Microsoft**

[Microsoft splits development along product lines rather than functionality, so it's not uncommon to find a bug repeated over multiple products from different teams, or to find a bug which has been fixed in one product reappear in another. Because of this the following descriptions don't usually identify a particular product because it's often a nontrivial exercise identifying in which locations the problems occur]

Earlier versions of MSIE encoded the emailAddress of a PKCS #10 request incorrectly. If the string doesn't fit into the range acceptable for a PrintableString it produces a UniversalString (with 32-bit wide characters). Since a PrintableString doesn't include an '@', you always end up with UniversalStrings. The correct type should be IA5String.

MS software will often generate components with UniversalStrings in places where they shouldn't really occur. According to MS, this was necessary because BMPStrings weren't allowed in DirectoryStrings until October 1997, which if true would require another entry in this list, "Some MS software erroneously produced BMPStrings before it was permitted in October 1997". It also seems to randomly use T61Strings where a PrintableString should be used (there's no discernable pattern to this). This was fixed in MSIE 4.01 (but not in other MS software as far as I can tell), where it uses either PrintableString or BMPString (skipping T61String completely).

The same software will dump multiple AVAs into a single RDN, this is most probably an encoding bug since the AVAs consist of a random jumble of components with nothing in common.

Some Microsoft software will generate negative values about 50% of the time whenever it encodes anything as an INTEGER because it ignores the fact that the top bit of an integer is the sign bit (this is still occurring in programs released as recently as early 1998).

When MSIE stores certificates, it recodes some components (specifically the various time stamps) which don't include seconds so that they now include seconds, which means that the signatures in the certificates are no longer valid. The altered encoding is in fact the correct one, but it's probably not worth altering the certificate to the correct form if it means breaking the signature on it. A workaround for this problem (mentioned in the "Validity" section of this document) is to ensure you never generate a certificate with the seconds field set to 0.

MS software enforces validity period nesting for certificates, which can cause certificates which are valid everywhere else to appear invalid when loaded into a MS product.

Although various MS programs give the impression of handling certificate policies, they only have a single hardcoded policy which is the Verisign CPS. To see an example of this, create a certificate with a policy of (for example) "This policy isn't worth the paper it's not written on" and view the cert using Outlook Express. What's displayed will be the Verisign CPS.

The entire AuthentiCode certification framework collapsed on 1 July 1997 when the AuthentiCode CA certificates expired (most people never noticed this due to stealth upgrades of security components when other products were installed). Microsoft issued an update (AuthentiCode 2.0) which includes a partially-documented

timestamping mechanism which is supposed to allow signatures to be updated in some manner. Creating certificates with a lifetime of over four decades (see below) is probably also intended to stop this problem from recurring.

The MakeCert certificate-generation program gives certificates a default validity of over 40 years (!!!). This creates three problems: firstly, implementations which use the ANSI/ISO C `time_t` type for dates (which most implementations do) will, for certificates generated after late 1997, be unable to check the validity period. Secondly, kids with the next-millennium equivalent of pocket calculators will be breaking the keys for these certificates by the time half the validity period is up. Finally, because of validity nesting of CA certs which typically expire after a year or two, these certificates will either be treated as invalid as soon as they're issued, or will become invalid a long time before their actual expiry date, depending on how the software enforces validity nesting.

MakeCert generates certificates with peculiar collections of deprecated and obsolete extensions. Incredibly, it can be persuaded to generate different incompatible versions of the same extension depending on what options it's given.

When asked to add a Netscape-style extension to a code-signing certificate, MakeCert adds an extension which marks it as an SSL client certificate, presumably because whoever wrote it got the endianness of the bit strings reversed. Given that some implementations will allow Netscape extensions to override other extensions which are present (both MSIE and Netscape Navigator seem to treat a Verisign cert with a Netscape extension of "is a CA" and an X.509v3 extension of "isn't a CA" as being a CA certificate), it'll be interesting to see what other implementations make of these certificates.

In code signing certificates, the `displayName` (aka `agencyInfo`) is encoded as an extension identified by the X.509 `commonName` OID, with the data being an OCTET STRING containing a mostly Unicode representation of an ASCII URL string, winning it the prize for "Most Mangled Extension".

An ever-changing variety of Microsoft products incorrectly encode bit strings in certificate extensions.

Outlook Express generates certificates which not only use the `GeneralizedTime` encoding more than 50 years before they're allowed to, but give the resulting certificate an expiry date in the early 17th century. A Microsoft spokesman has confirmed that this is deliberate, and not a bug.

How many Microsoft programmers does it take to change a lightbulb?  
None. They define darkness to be the new industry standard.  
-- Unknown

The same certificate type is marked as an X.509 v1 certificate even though it contains extensions which aren't allowed in X.509 v1. To be valid, the certificate should be marked as X.509 v3.

Some MS software will reject a certificate with certain basic extensions marked critical (this provides one of the nonstandard definitions of the critical flag mentioned earlier, "reject the certificate if you find a critical extension").

Other MS software, contradicting the previous bug, will ignore the critical flag in extensions, making the software useless to relying parties since they can't rely on it to correctly process critical certificate components.

Microsoft software seems to mostly ignore the `keyUsage` bits and `extKeyUsage` values and use the first certificate it finds for whatever purpose it wants (for example if a subject has a signature and an encryption cert, it will take the first one it finds and use it for both purposes, which will result in the decryption and/or signature check failing).

Microsoft certificates can include arbitrarily mangled interpretations of what comprises a DN, examples ranging from DNs which consist of a single `CommonName` through to DNs with the country missing.

Microsoft's key-handling software assumes that public keys come in a certain fixed format (for example that keys have a certain, set number of bits, that (for RSA) `p` and `q` are both the same size, and in some cases that `e` falls

within a certain limited range). If all these conditions aren't met, encryption and signatures quietly fail. To avoid this, you need to make the keys a standard, common length (eg 512 bits for exportable crypto), make sure p and q are of the same size, and use a small value for e.

In extensions which contain URL's, Microsoft software will sometimes produce something which is neither an HTTP URL nor a UNC file URL, but some weird mixture between the two.

Microsoft certificates contain a peculiar deprecated form of the authorityKeyIdentifier extension. In this extension, CA certificates sometimes identify themselves, rather than the CA which issued the issuing certificate, which would lead to either endless loops or verification failures if software took this identification literally.

Extensions in certificate requests are identified using the nonstandard {microsoft 2 1 14} OID instead of the {pkcs-9 14} one, which means that although they're standard extensions, no non-MS software can recognise them.

After MSIE 5.0, Microsoft partially switched to using the more standard {pkcs-9 14} identifier, but also invented their own format for further extensions alongside the existing one which nothing else can process. The extensions contain either ASCII or (lengthy) Unicode strings identifying the product which created the cert request.

Some DNs produced by MS software are encoded with zero-length strings.

Country codes in DNs created by MS software can contain values other than the allowed two-character ISO code (for example three-character country name abbreviations).

Code dating from about the MSIE 5.0 and newer stage will chain certificates using the authorityKeyIdentifier even if the issuer name doesn't match, in violation of X.509 and PKIX. This means that a certificate could claim an issuer of "Verisign Class 1 Public Primary Certification Authority" even though it was actually issued by "Honest Joe's Used Cars and Certificates", and MSIE will accept it as valid.

Date handling for certificates appears to be completely broken, although it's difficult to determine the real extent and nature of the problems. For example a certificate which is valid from December 1951 to December 2050 is reported by Windows as being valid from December 1950 to December 1950. Despite this claim, it doesn't recognise the certificate as having expired, indicating multiple levels of date-processing bugs in both the decoding and handling of dates.

Certificates don't have to contain valid keys or signatures in order to be accepted by Windows, for example a test certificate with an exponent of 1 (which means the key has no effect) is accepted as valid. This is probably required to support an MSDN knowledge base article which tells users how to extract keys from CryptoAPI by encrypting them with a public key with an exponent of 1.

Some Microsoft products have been spotted which generate things which are claimed to be certificate requests but which actually contain PKCS #7 signedData with a payload which is tagged as raw data but which is really a certificate request with Microsoft's gratuitously-incompatible CRMF extensions which aren't CRMF extensions, one of which is a certificate which is used to sign the PKCS #7 signedData. Needless to say, nothing in existence except for a small subset of other Microsoft products know what to make of this mess.

### **MISSI**

MISSI certificates use shared DSA parameters, with p, q, and g only being specified in the root CA certificate. Apart from the risk this introduces (it allows signatures on certificates to be forged under some circumstances), it also complicates certificate processing because the parameters needed to verify a signature are generally held in a certificate held God knows where, so even if a certificate is valid and trusted, it's not possible to use it without having the entire cert chain up to the root on hand.

## **NASA**

NASA identifies RSA public keys in certificates by a peculiar deprecated X.500 OID for RSA (2 5 8 1 1).

## **Netlock**

The Netlock CA incorrectly encodes bit strings in key/certificate usage extensions.

## **Netscape**

Invalid encoding of some (but only some) occurrences of the integer value 0 in Commerce Server private keys. The problem occurs when the integer value 0 in the RSAPrivateKey is encoded as integer, length = 0 rather than integer, length = 1, value = 0. This was fixed on 20 March 1996 (between the Commerce Server 1.13 and Enterprise 2.0 releases).

Some unidentified early Netscape CA software would encode an email address as a PrintableString CommonName in a DN. This encoding is invalid since a PrintableString can't contain an '@' character. The same CA issued a root certificate with a validity period of zero by encoding the start and end time as the same value.

Earlier Netscape software encoded the critical flag in extensions incorrectly. The flag is defined as BOOLEAN DEFAULT FALSE, but it was always encoded even if set to its default value. This bug was fixed in early 1997.

Handling of non-PrintableString components of DNs is somewhat ad hoc, at one point the code would take any string which wasn't a PrintableString and encode it as a T61String, whether it was or not. This may have been fixed now, it results in improper encodings but most products don't care because they don't know what to do with strings that should really be BMPStrings either. Also, the Cert Server enforces an upper limit on the DN length of 255 characters when the DN is encoded as per RFC 1779 (so the actual limit is slightly less than 255 characters).

The Netscape certificate extensions specification states that the keyUsage extension can be ignored if it's present and non-critical, and lists mappings from keyUsage to Netscape's own cert-type extension. Some implementations follow these mappings and include both types of extension (notably Thawte), others ignore them and include only the Netscape-specific extension (notably Verisign).

Navigator can ignore the basicConstraints extension in some instances when it conflicts with other extensions (see the entry for Verisign for details).

One way to get it to ignore all extensions is to add the cert as type application/x-X.509-ca-cert, in which case it'll accept anything including end entity certificates and certificates with invalid signatures as CA certificates.

The Netscape CA software incorrectly encodes bit strings in key/certificate usage extensions.

Adding a new certificate with the same DN as an existing certificate (for example a CA and site certificate with the same DN) can corrupt the Netscape certificate database.

Encountering a T61String in a certificate will cause versions of Netscape dating from before 1998 to crash with a null pointer dereference. Current versions will still crash if they encounter a BMPString or UTF8String.

## **NIST**

NIST has a root CA certificate which the basicConstraints extension identifies as being a non-CA certificate, making it invalid for its intended purpose. One of these broken certificates was used for several versions of the PKIX X.509 certificate and CRL profile as an example of a CA certificate, which could result in the same problem as the PKCS #10 example vs specification contradiction.

## ***Nortel***

One of Nortel's CA products encodes UTCTime dates using the incorrect non-GMT format. The software is used by (at least) IBM, the Canadian government (GTIS/PWGCS), and Integrion, and can be identified by the "2.1d" version string in a Nortel-specific attribute attached to the cert. Nortel's WebCA 1.0 doesn't have this problem, the fact that the 2.1d software uses a number of old, deprecated OIDs and the WebCA software doesn't would indicate that this is more recent than whatever produced the "2.1d" certs (the "2.1d" refers to a particular release of the infrastructure (Entrust/Manager, Entrust/Officer, and Entrust/Admin) and the corresponding client-side components (toolkits and Entrust/Client) rather than a single product). This problem was fixed in the Entrust 3.0 infrastructure release.

Nortel spun off their crypto/security products group as Entrust Technologies in January 1997, for further notes see the entry for Entrust.

## ***PKIX***

PKIX requires that end entity certificates not have a basicConstraints extension, which leaves the handling of the CA status of the certificate to chance. Several popular applications treat these certificates as CA certificates for backwards-compatibility with X.509v1 CA certificates which didn't include basicConstraints, but in practice it's not really possible to determine how these certificates will be treated. Because of this, it's not possible to issue a PKIX-compliant end entity certificate and know how it'll be treated once it's in circulation.

The theory behind this usage is that applications should know that a v3 certificate without basicConstraints defaults to being a non-CA certificate, however (even assuming that applications implemented this), if basicConstraints would have been the only extension in the certificate then defaulting to leaving it out would make it a v1 certificate as required by PKIX, so the v1 rules would apply. To get the required processing, the certificate would have to be marked as v3 even though it's v1 (and the application processing it would have to know about the expected behaviour). In any case it's a somewhat peculiar use of the certificate version number field to convey certificate constraint information.

One use for this feature is that it may be used as a cryptographically strong random number generator. For each crypto key an application would issue 128 basicConstraint-less certificates, hand them out to different implementations/users, and use their CA/non-CA interpretation as one bit of a session key. This should yield close to 128 bits of pure entropy in each key.

In between the draft versions of the standard (which were around for several years) and the RFC, the policy qualifiers display text type was quietly changed to exclude IA5String, which had been present in all the drafts. As a result, certificates complying with the drafts didn't comply with the RFC. Since no one but Verisign seems to use these fields (see comments elsewhere in this document), it's noticeable by the fact that Verisign certs issued during the lifetime of the drafts appear to contain a string type which is invalid according to the RFC. This isn't a Verisign problem though, since they complied with the spec at the time the certificates were issued.

## ***Safelayer***

Safelayer have solved the T61String problem by unilaterally extending PrintableString to include latin-1 characters (apparently this was a conscious decision, not an encoding bug). Since they're in Spain, this results in most of their certs having invalid string encodings.

## ***SECUDE***

The SecuDE software produces certificates with the public key components identified by a peculiar deprecated X.500 OID for RSA (2 5 8 1 1).

Certificates are hashed with MD2, despite the fact that this algorithm has been deprecated for some time.

Older versions of SECUDE produced broken CRLs consisting of various portions of full CRLs (the software stored the CRLs in a nonstandard format, this was fixed after 4. x but since this was the last free version it's still in use in some places).

### ***SecureNet***

This CA uses Microsoft software for its certificates, which means they display all the bugs typical of MS-created certificates (see the extensive entry under the Microsoft heading for more details).

### ***Security Domain/Zergo/Certificates Australia***

The authorityKeyIdentifier contains both a keyIdentifier which isn't the required SHA-1 hash (the subjectKeyIdentifier is a hash, it's only the authorityKeyIdentifier which isn't), as well as an authorityCertIssuer represented as a registeredID object identifier. Other certificates contain an authorityCertIssuer consisting of a single zero byte. Another cert contains an authorityCertSerialNumber consisting of a single zero byte.

Bit strings in key/certificate usage extensions are encoded incorrectly.

The certificatePolicies extension uses incorrect OIDs for the various components such as the CPS and unnotice, the CPS URL isn't a valid URL, and the unnotice is given as an IA5String rather than a SEQUENCE of assorted components. A different certificatePolicies contains what looks like another invalid OID which could be an attempt at the one in the previously mentioned certificatePolicies.

In some cases the issuerName is encoded as 127 bytes of zero-padded registeredID OID.

(Ugh, this stuff just gets worse and worse - the later attempts at things like PKCS #7 cert chains are so far removed from the real thing that they don't even remotely approach the actual standard. I'll stop now).

These issues were resolved in 1999 in a merger with Baltimore by switching to Baltimore's UniCERT product.

### ***SEIS***

The Swedish SEIS project software formerly created UniqueIdentifiers which contained embedded character strings (this is a peculiarity). In some versions of the software, these were flagged as constructed rather than primitive (this is a bug). The encoding bug was rectified in later versions. The character strings encode 16-digit numbers, which are apparently used as some form of extra identification which doesn't fit easily into a DN.

In the EID v2 certificate draft of February 1998, the use of UniqueIdentifiers was deprecated in favour of a DN entry under a SEIS OID which contained the information formerly in the UniqueIdentifiers.

### ***SET***

There is a minor problem which arises from the fact that SET uses the ':' character as a delimiter in commonName components of DNs. However BMPStrings have more than one character which looks like a ':'. The correct one to use is the ':' which is common to both PrintableString and BMPString, ASCII character 0x3A.

### ***SHTTP specification***

There is at least one invalid PCKS#7 example in earlier versions of the spec. More recent drafts starting with <draft-ietf-wts-shhttp-03.txt>, July 1996, fix this. Implementors should ensure they are working with corrected versions of the draft.

## ***SI-CA***

The SI-CA incorrectly encodes some bit strings in key/certificate usage extensions. Unlike other implementations and CAs which have this problem, it doesn't do it consistently, correctly encoding some bitstrings and incorrectly encoding others.

## ***Signet***

[Some of these problems were fixed in late 1998]

Default fields are encoded when they have the default value rather than being omitted.

Some basicConstraints extensions are marked as being critical, others in the same chain are marked noncritical (using the incorrect default field encoding mentioned above).

Bit strings in key/certificate usage extensions are encoded incorrectly.

Leaf certs are identified by giving the issuing cert a path length constraint of 0 in the basicConstraints extension rather than by marking the cert itself as being a non-CA cert. This isn't invalid, but is somewhat peculiar, and doesn't work if the leaf cert is implicitly trusted (without the signing cert being present), since there's no indication in the leaf cert itself as to whether it's a CA cert or not.

BOOLEAN values have non-DER encodings.

The name constraints extension contains a permittedSubtree with what appears to be an otherName consisting of a single zero byte (no OID or anything else).

## ***South African CA***

This CA has a root cert which has the UTCTime encoded without the seconds (see the section "Validity" above), newer versions encode the time with the seconds. This isn't an error since the accepted encoding was changed in mid-1996, merely something to be aware of.

## ***SSLeay***

SSLeay incorrectly encoded bit strings in key/certificate usage extensions, this was fixed in late 1998 in version 0.9.1.

## ***Sweden Post/ID2***

Sweden Post certificates incorrectly encode the certificate validity time, omitting the seconds field in the UTCTime field.

The subjectKeyIdentifier is encoded in a variety of ways ranging in length from 1 to 8 bytes. In CA certs the subjectKeyIdentifier is the same as the authorityKeyIdentifier (which is valid, but odd) and consists of a text string identifying the CA key (this isn't explicitly disallowed because of the confusion over what's really meant to be in there but it isn't really what's supposed to be in there either).

Instead of using a common name, the data is encoded as a surname + given name + unique ID, producing DN fields with multiple AVAs per RDN (this isn't a bug, but is definitely a peculiarity, and causes problems for software which expects to use a common name as the identity of the certificate owner).

CA certs include policy mappings which specify the same issuer and subject domain policy (this is known as a tautology mapping).

End-entity certs include (deprecated) subjectUniqueIdentifier fields (this is a peculiarity). The fields contain embedded PrintableString's consisting of variable-length numbers.

## **SWIFT**

SWIFT certificates have incorrect field lengths for several of the certificate fields, so that the SWIFT CA doesn't even have a valid root CA certificate.

## **Syscall GbR**

The Syscall GbR CA incorrectly encodes bit strings in key/certificate usage extensions.

## **TC Trustcenter**

Some certs contain zero-length strings in the DN, this was fixed in early 1999.

## **Telesec/Deutsche Telekom Trustcenter**

Interoperability considerations merely create uncertainty and don't serve any useful purpose. The market for digital signatures is at hand and it's possible to sell products without any interoperability  
-- Telesec project leader discussing the Telesec approach to interoperability (translated),  
"Digitale Identitaet" workshop, Berlin, May 1999.

Telesec certificates come in two flavours, general-purpose certificates (for example for S/MIME and SSL use) and PKS (Public Key Service) certificates which are intended for use under the German digital signature law. The two aren't compatible, and it's not possible to tell which one a given certificate follows because the certificates don't include any policy identification extensions. An example of the kind of problem this causes is that the Telesec CPS claims certificates will be signed with RSA/MD5, however published end-entity certs have appeared which are signed with RSA/RIPEMD-160. These aren't invalid, they just follow the PKS profile rather than the PKIX profile or CPS. Another example of this is the fact that PKS certificates use GeneralizedTime, which is allowed under the PKS profile but not the PKIX/SSL/SMIME/etc ones.

Some strings are encoded as T61Strings where PrintableStrings should be used (there's no pattern to this). The strings which really are T61Strings use floating diacritics, which isn't, strictly speaking, illegal, but anyone who does use them should be hung upside down in a bucket of hungry earthworms.

Common names are encoded in an RDN component with two AVAs, one identified by an unknown Telekom OID and the second identified with the CN OID, however the common name in it is modified by appending extra letters and digits which are intended to distinguish non-unique names in the same manner as the (deprecated) X.509v2 uniqueIdentifiers. Since even imaginary (guaranteed unique) names are modified in this way, it appears that this alteration is always performed.

The certificates encode INTEGER values incorrectly by setting the high bit, which makes them negative values. This is particularly problematic with RSA keys since they use a hardcoded exponent of 3,221,225,473 (!!!) which always has the high bit set (0xC0000001), so all the RSA certificates have invalid encodings. This was corrected in late 1999.

CA certificates are encoded with no basicConstraints present, which under PKIX rules (which aren't terribly sensible, see an earlier section) explicitly makes them non-CA certificates and under normal conditions makes them highly ambiguous at best.

[This stuff just gets worse and worse, but I couldn't be bothered going through and figuring out all the broken things they do. Telesec also hardcode things like certificate parameters into their software (so that, for example, half the info about a user might be stored on a smart card (needed for SigG compliance) and the other half is hardcoded into the driver DLL for the card), guaranteeing that nothing else in existence can work with it. Ugh].

## ***Thawte***

For a brief while, Thawte encoded email addresses as a PrintableString in a DN's CommonName. This encoding is invalid since a PrintableString can't contain an '@' character. This problem has been fixed.

## ***TimeStep/Newbridge/Alcatel***

TimeStep incorrectly encode the DirectoryName in a GeneralName, using an implicit rather than an explicit tag. The ASN.1 encoding rules require that a tagged CHOICE always have an explicit tag in order to make the underlying CHOICE tag visible. Timestep were bought by Newbridge who were in turn bought by Alcatel, thus the naming confusion.

## ***UNINETT***

Some certs from this CA identify RSA public keys in certificates by a peculiar deprecated X.500 OID for RSA (2 5 8 1 1). However in one case a second cert for the same person produced on the same day used rsaEncryption instead.

## ***uniqueIdentifier***

There are at least two incompatible objects called uniqueIdentifier, the first is an attribute defined in 1991 in RFC 1274 with string syntax and an illegal OID (rendering it, in theory, invalid), the second is an attribute defined in 1993 in X.520v2 with BIT STRING syntax. LDAPv2 used the RFC 1274 interpretation, RFC 2256 changed the name for the X.520 version to x500uniqueIdentifier for use with LDAPv3. There is also a uid attribute defined in RFC 1274, this is different again.

## ***Verisign***

Verisign incorrectly encodes the lengths of fields in the (deprecated) keyUsageRestriction extension, which is used for the Microsoft code signing certificates they issue. Some software like MSIE will quite happily accept the broken extension, but other software which does proper checking will reject it (actually there are so many weird, unknown critical extensions in the code signing certs that it's unlikely that anything other than MSIE can process them anyway).

Verisign were, as of March 1998, still issuing certificates with an MD2 hash, despite the fact that this algorithm has been deprecated for some time. This may be because they have hardware (BBN SafeKeyper) which can only generate the older type of hash.

Verisign Webpass certificates contain a basicConstraints extension which designate the certificate as a non-CA certificate, and a Netscape cert-type extension which designate the certificate as a CA certificate. Despite this contradiction, MSIE doesn't seem have any problems with using the certificate, indicating that it's ignoring the basicConstraints entirely. Navigator will load the certificate, but gets an internal error when it tries to use it. This was fixed in late May 1998.

Some Verisign certificates mix DER and BER inside the signed portion of the certificate. Only DER-encoded data is allowed in this part of the certificate.

For a brief period of time in mid-1998 Verisign issued certificates signed with an MD2 signature block wrapped inside an MD5 signature block. This was fixed fairly quickly.

Verisign doesn't mark extensions as critical, even fundamental ones like basicConstraints. This is because of Microsoft software which rejects certificates with critical extensions.

### ***Y2K/2038 Issues***

Many implementations base their internal time encoding on the Unix/ANSI/ISO C seconds-since-1970 format, and some use 1970 as the rollover date for interpreting two-digit dates instead of xx50. This includes, as of late 1997, Netscape and SSLey. In addition the January 2038 limit for seconds expressed as a signed 32-bit quantity means they can't represent dates past this point (which will cause problems with Microsoft's four-decade validity periods). Implementations should therefore be very careful about keys with very long expiry times, for security as well as date handling,

## **Annex A - The Standards Designer Joke**

I resisted adding this for a long time, but it really needs to be present :-).

An engineer, a chemist, and a standards designer are stranded on a desert island with absolutely nothing on it. One of them finds a can of spam washed up by the waves.

The engineer says "Taking the strength of the seams into account, we can calculate that bashing it against a rock with a given force will open it up without destroying the contents".

The chemist says "Taking the type of metal the can is made of into account, we can calculate that further immersion in salt water will corrode it enough to allow it to be easily opened after a day".

The standards designer gives the other two a condescending look, gazes into the middle distance, and begins "Assuming we have an electric can opener...".

## Acknowledgements

Anil Gangolli, Deepak Nadig, Eric Young, Harald Alvestrand, John Pawling, Phil Griffin, and members of various X.509 and PKI-related mailing lists provided useful comments on usage recommendations, encoding issues, and bugs.