Web Server Management: Securing Access to Web Servers

Jon Warbrick
University of Cambridge Computing Service

jw35@cam.ac.uk

Web Server Management: Securing Access to Web Servers
by Jon Warbrick

This course covers the "HTTPS" (secure HTTP) protocol, which can protect communication between web browsers and web servers. This is presented from the point of view of a web server administrator who wishes to configure servers to support such communication. The course includes an outline of the operation and features of the protocol, and covers the practical configuration of an Apache server under Linux. The general principles covered apply to Apache on other platforms, and to other web servers, thought the details will vary.

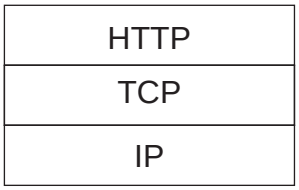
The course covers other aspects of web server security only in passing, and does not cover general web server installation or configuration issues. A basic understanding of the way that web servers operate, along with some experience of configuring and administering such servers, either on shared or personal machines, is assumed.

The course web site at `http://www-uxsup.csx.cam.ac.uk/~jw35/courses/using_https/` contains an up-to-date copy of these notes and related resources. Requests for assistance by members of the University on the material covered here can be e-mailed to `<web-support@ucs.cam.ac.uk>`

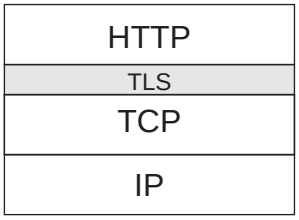
Chapter 1. Orientation

What is HTTPS?

HTTP (no "S") is the protocol spoken between web browsers and web servers. It is used to submit requests from browsers to servers and to carry responses back. In addition to carrying the raw documents, HTTP also carries "meta information" about those documents - size, document type, expiry date, etc. HTTP uses the IP protocol family's TCP layer to provide the browser-server communication. HTTP can be visualised as running "on top of" TCP, which in turn runs "on top of" IP.



Because of their birth in academic collaboration, neither HTTP nor TCP provide much in the way of security. HTTP traffic travels as clear text across the communication networks where it can easily be intercepted. TCP connections are made to hosts only identified by network names that are relatively easily subverted. The only standard facilities for identifying users depend on transmitting user names and passwords in clear with every request. While all this is fine when accessing the majority of web content - which is in any case freely available - it is unsuitable for those applications where any sort of security or confidentiality is required. HTTPS is HTTP running on top of either the TLS (Transport Layer Security) or SSL (Secure Sockets Layer) protocol. These sit between an application protocol and the TCP layer and provide additional security features.



The interface between an application protocol and TLS or SSL is modelled on that between an application protocol and plain TCP. This makes it fairly straightforward to add TLS or SSL functionality to existing programs and so they are often used to secure protocols other than HTTP, such as POP (the Post Office Protocol), IMAP (Internet Message Access Protocol), SMTP (the Simple Mail Transport Protocol) and LDAP (Lightweight Directory Access Protocol). Note however that SSH (the "Secure Shell") does not use TLS or SSL though it uses similar cryptographic components. Use of HTTPS can be recognised by "https://..." URLs and key or closed padlock icons in common browsers. Recent versions of Firefox turn the browser address bar yellow when accessing sites using HTTPS.

SSL was originally developed by Netscape and released as SSL version 2. It was significantly redeveloped to form SSL version 3, and then further developed and documented in RFC2246 as TLS. There are significant differences between these three variants (in particular between SSL v2 and SSL v3) but it is possible for implementations to inter-operate providing they share support for at least one variant. We will refer to the whole family as TLS throughout the rest of this document.

What does HTTPS give you?

Client-server, end-to-end encryption

All the HTTP traffic between the client and the server is encrypted, preventing anyone from understanding it even if they can intercept it.

Message Integrity

Integrity checks ensure that the messages making up the HTTP traffic cannot be altered in transit, neither can messages be added or removed from the sequence, without detection.

Strong authentication of the server

Simply providing encryption and message integrity gives little security if you do not know who the other party in the conversation actually is. With plain HTTP, your only assurance is that your browser has probably connected to the host whose name appeared in the URL you followed. This may not be the case (for example it is easy to subvert the name-to-address mapping process), and in any case it is difficult to tell who is actually operating any particular server. It is also fairly easy to mount a “man in the middle” attack against plain HTTP.

Under HTTPS, all servers offer the browser a cryptographic “certificate”. These certificates are issued by trusted third parties and contain information that identifies the server and the organisation operating it. How this works, and how the trusted third parties are nominated, will be covered later.

Optional authentication of the browser user

Optionally, HTTPS also allows the browser to supply a certificate to the server. This can provide strong authentication of the identity of the browser user, but this feature is rarely used, probably because of the difficulty of issuing such certificates to all users. Certificates are also large, making it difficult for mobile users to have them to hand when needed. Embedding certificates in portable tokens is one approach to solving these problems. This issue is addressed further below.

A heads-up about security in general

Before diving into details of cryptography, it is appropriate to first step back for a view of computer security in general. "Security is a process, not a product" (Bruce Schneier in *Secrets and Lies*, Wiley Computer Publishing, 2000) and while HTTPS can be a useful component of that process it is dangerous to think that it provides security in and of itself. It is also important to understand the “threat model” as it applies to

your intended application: what are you protecting?; from whom?; what resources do they have available?; how much are you willing to pay? Given that you are interested in HTTPS, it is reasonable to assume that you are considering handling some sort of sensitive data via a web server. So consider:

- TLS *only* protects the data during transmission. What happens to the data once it is received?
- ... or even before it is sent?
- Is the computer running your webserver itself secure from outside attack? Is it up-to-date on patches? What else does it do?
- Is your webserver (and any computers to which it passes information) physically secure? Are staff who have legitimate access to it trustworthy? Can the cleaners read data from the server before anyone arrives in the mornings? Etc., etc.

Remember too that there may be legal requirements if you process some forms of data. If you process data that relates to identifiable living human beings then the provisions of the Data Protection Act 1998 will apply to that processing. If you are responsible for encrypted data then the Regulation of Investigatory Powers Act 2000 may apply and could require you to decrypt data under some circumstances, or even to hand over your encryption keys.

The problem with politics

Computer cryptography has been, and to some extent continues to be, dogged by two particular legal/political issues:

- Patents: many important cryptographic algorithms are (or have been) subject to software patents in some countries - particularly in the USA but also in Europe. Until recently this included most algorithms implementing public key cryptography, which is vital to TLS's operation. However one of the important patents (that on the RSA algorithm in the USA) expired in late 2000 and this has simplified the situation.
- Munitions: software implementing strong cryptography is regarded by many countries as being indistinguishable from guns and explosives, at least as far as import and export restrictions are concerned. Again the USA used to be particularly difficult in this regard, making it almost impossible to export software from the US that implemented strong cryptography. This situation improved in 1999 when the US administration significantly relaxed their restrictions, though some remain.

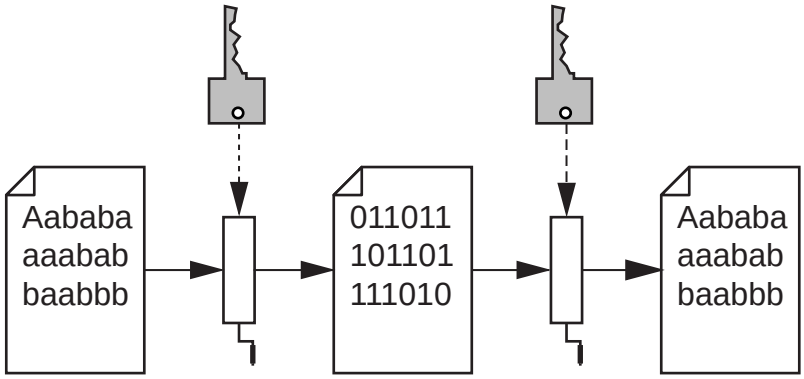
This has lead to all sorts of oddities. For example, it is quite common to find that cryptographic software has been deliberately developed in countries other than the US, both to avoid export restrictions and to avoid problems with patents. It is also still common to come across references to "Export Grade" ciphers (weak ciphers that were permitted to be exported before 1999), or "US domestic grade" ciphers (all the rest). Additionally some cryptographic protocols use what can seem to be unusual algorithms to avoid patent restrictions.

Chapter 2. A crash course in cryptography

For our purposes (dealing with TLS) there is a small amount of cryptography that you need to know about. To avoid the all too common problem with computer security of "shooting yourself in the foot" you really *do* need to understand this much. Please appreciate that what follows is a broad-brush outline that glosses over an embarrassingly large amount of detail.

Symmetric ciphers

These are what most people think of as codes: using a well-known algorithm and a secret key to encode information, which can be decoded using the same algorithm and the same key.

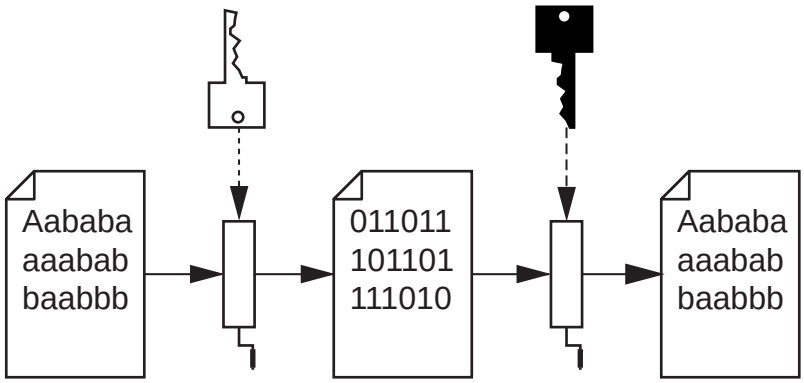


Notice that it is only the key that is secret; encryption schemes that depend on the algorithm remaining secret are not regarded as satisfactory. Some of the symmetric algorithms used by TLS include RC2, RC4, DES (The US "Data Encryption Standard"), Triple-DES, and IDEA, many with a variety of key lengths. With all ciphers, the longer the key the harder they are to break. DES, which has a 56 bit key, is now routinely cracked (admittedly using specialist hardware) in a few days. Key lengths of 128 bits currently seem reasonable for information that you want to remain secret for the foreseeable future. TLS also supports a "NULL" encryption algorithm, intended for testing, which should be disabled in a live environment since otherwise an attacker might trick both client and server into negotiating use of this algorithm.

A significant problem with symmetric ciphers is that it is difficult to transfer the keys themselves securely.

Public-key ciphers

In public key cryptography (also known as asymmetric cryptography) keys come in pairs. Data encrypted with one key can only be decrypted using the other key from the pair, and it is not possible to deduce one key from the other. This helps to solve the key distribution problem since you can create such a key pair, publicise one of the keys widely (your "public key") and keep the other a closely guarded secret (your "private key"). Anyone can then send you data encrypted with the public key and only you, as the holder of the corresponding private key, can decrypt it.



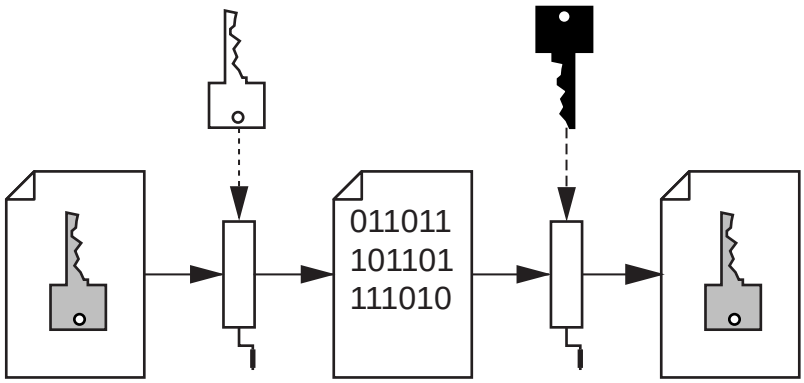
You can also test that someone really does have access to a particular private key, by inventing some random text and asking them to encrypt it. If you can decrypt the result using their public key, and providing that they have kept their private key private, then you can make some assumptions about who they are.

The most well known public key algorithm, and one used extensively by TLS, is RSA.

There are however two big problems with all known public-key algorithms. One is that they are *much* more complex than symmetric algorithms and so are slower and/or require much more computer power to implement. The other is that the keys need to be much longer to ensure security - current thinking suggests that public keys should be at least 1024, and perhaps 2048, bits long.

Key exchange

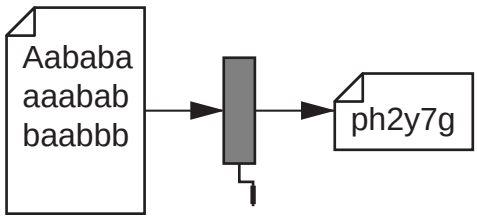
Because public key cryptography is so much slower than symmetric key cryptography, it is best to save it for situations where its facilities are really needed and to fall back to symmetric cryptography wherever possible, in particular for doing encryption of data in bulk. To do this, communicating parties need to securely establish a shared symmetric encryption key. One way to do this is to use public key cryptography: I can generate a temporary key, encrypt it with your public key and send it to you. As the holder of the corresponding private key, you (and only you) can decrypt the message and then we can both use the temporary key to communicate.



An alternative to this, and something additionally supported by TLS, is the "Diffie-Helman algorithm" which, unlikely though it seems, allows two communicating parties to establish a shared secret even if their communication is being monitored.

Message digests

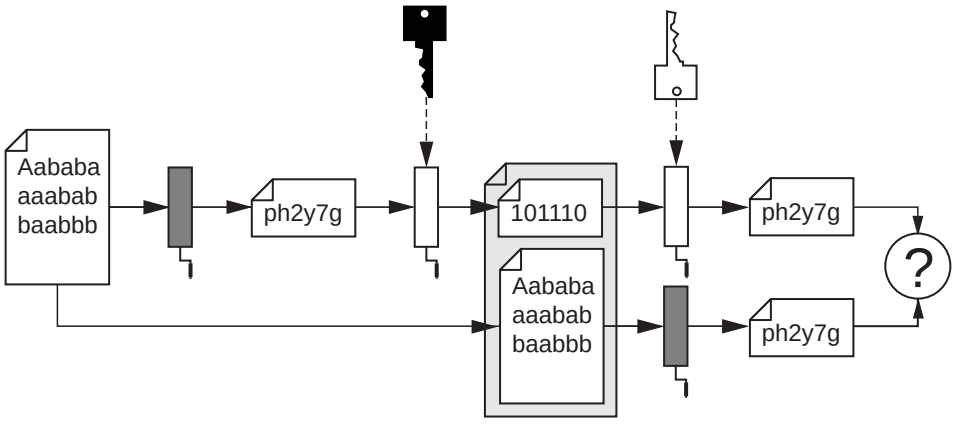
A message digest (sometimes called a hash) is a small fixed length "summary" derived from a longer piece of data. For a cryptographically useful digest, any change to the data results in a different summary and it is effectively impossible to generate a block of data to match a particular digest. You can ensure that a message you have received has not been tampered with if you can calculate the message's digest and compare it with one created before transmission.



TLS uses the MD5 and SHA-1 digest algorithms. MD5 produces a 128 bit result, SHA produces a 160 bit result. Many applications are now moving to SHA-1 though MD5 is still in widespread use.

Digital signatures

A digital signature is applied to a document by first calculating a message digest of the document, and then encrypting that digest (along with other information) using the signer's private key. Anyone can then be sure both that the document has not been altered since signing, and that the document was signed by the holder of the appropriate private key, by decrypting the digest using the signer's public key and comparing it to a freshly calculated digest.



The RSA public key algorithm is commonly used to do this. An alternative is DSA (the US Government's Digital Signature Algorithm) which operates in a slightly different way and which was designed specifically for creating digital signatures.

Public key certificates

One complication in using public key cryptography is ensuring that a particular public key belongs to the person you think it does. In a small organisation you might be given a copy of a public key by its owner who you already know, or perhaps by a

third party who is willing to vouch for the identity of the key owner. But this does not work on a global scale.

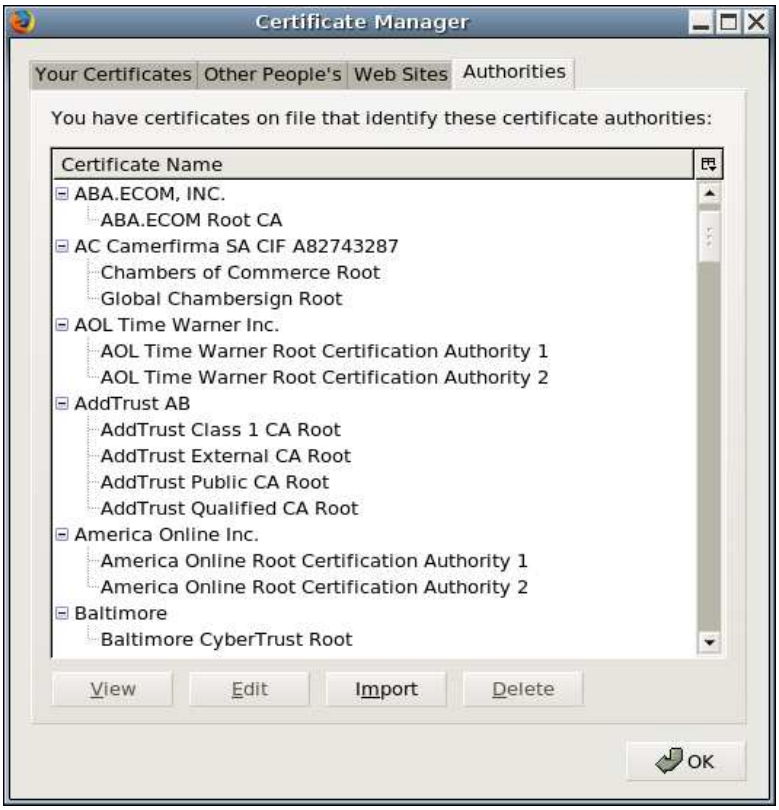
Public key certificates resolve this. Such a certificate consists of a public key and sufficient information to identify the owner of that key, the whole thing digitally signed by some third party who everyone chooses to trust. In this way an identity and the corresponding public key pair are bound together.

TLS, and almost all other applications of public key certificates, uses the X.509 certificate format. The X.509 standard was part of the much larger CCITT X.500 directory project, which has largely fallen by the wayside though current standards like LDAP preserve some of its more useful features. X.509 certificates were invented to address security needs elsewhere in X.500 and have subsequently been adopted for other applications. While this is largely an irrelevance, these certificates do have some odd features (as we will see) which they owe to this strange background.

Certification Authorities and "Public Key Infrastructure"

A careful reader will have noticed a "chicken and egg" problem with Public Key Certificates: to verify the trusted third party's digital signature we will need their public key. We could get this from a further certificate (indeed TLS has support for such "certificate chains"), but that is just putting off the moment of truth.

TLS, at least as applied to webserver access, takes a pragmatic approach to this problem. A number of organisations around the world have set themselves up as Certification Authorities (CAs) and they issue signed certificates on a commercial basis. To make this all work they have arranged for their own certificates to be distributed along with the common web browsers, and for the browsers to be configured to implicitly trust these certificates. When you install Internet Explorer, or Firefox, or Safari, you also install several tens of such "Root Certificates" and choose (whether you realise it or not) to implicitly trust the associated CAs. For example see "Edit -> Preferences -> Advanced -> Encryption View Certificates -> Authorities" in Firefox 2.



This is fairly worrying, though the widespread distribution and use of these certificates means that any blatent misuse is likely to be noticed. However it is the case that the keys coresponding to the certificates distributed with the major browsers have as a result become extreemly valuable and change hands from time to time. There are now a number of CAs who do not necessiarally check as carefully as they might that the details in certificates that they issue are reliable. The "Extended Validation" scheme (see the Section called *Extended Validation* in Chapter 5) now being promoted by some of the larger CAs is to some extent an attempt to address this.

A system for issuing and revoking certificates, distributing root certificates, etc., to make the widespread use of public key cryptography possible is sometimes called a “Public Key Infrastructure” (PKI).

The TLS process

Given the cryptographic building blocks that we have now discussed, establishing an HTTPS connection turns out to be fairly straightforward. At least at the high level at which we are working - in practise there is quite a lot of additional complication to guard against various possible attacks.

- The client web browser initially connects to the server on an agreed TCP port (443 by default)
- The client and server agree mutually available TLS/SSL protocol versions, cipher specifications, compression algorithms, etc.
- The server sends its public key certificate to the client
- The client verifies the server certificate (can the client verify the signature? does the client trust the CA who signed the certificate? is the website identified in the certificate the one that is being accessed? has the certificate expired?)

- The client and server agree a shared secret, either by using the server's public key from its certificate or otherwise
- The client and the server use the secret to create the same symmetric encryption key
- The client and the server switch to communicating using the previously agreed symmetric cipher and the key just established. Sequence numbers included in the encrypted message exchanges ensure that components can not be removed or replayed.

The downside of using HTTPS

The increased security that HTTPS brings might suggest that it should be used as a matter of course, but there are drawbacks that must be considered.

- Pages accessed by HTTPS can never be cached in a shared cache. Since the conversation between browser and server is encrypted, intermediate caches are unable to see the content to cache it. Worse, some browsers will not even cache HTTPS documents in their local per-user caches. Worse still, since it is dangerous to mix HTTPS and HTTP content on the same page (there are some scripting attacks that can allow a script in one component of a page to read data from another), even embedded icons and pictures have to travel encrypted and therefore can not be cached. The lack of local caching can lead to problems in Internet Explorer that can make it impossible to save documents to disk or to open them in external applications (see for example http://support.microsoft.com/kb/812935)
- The encryption/decryption represents a computation overhead for both server and browser. Most modern client systems will probably not notice this, but on a busy server handling multiple simultaneous HTTPS connections this could be a problem.
- Some firewall or proxy systems may not allow access to HTTPS sites. Sometimes this is simply because the administrators have forgotten to allow for HTTPS. However sometimes it is a conscious security decision: since HTTPS connections are end-to-end encrypted, they can be used to carry any traffic at all. Allowing them through a firewall, which then has no way to look inside the data stream, could allow any sort of data transfer (in either direction).
- Cost. Commercial CAs charge something like £100-200 per year for issuing certificates. And you need at least one for every site that you secure, because the hostname (as it appears in URLs) forms part of the certificate. There is also the "hidden" administrative cost of applying for the certificate and of arranging for its renewal each year.

Chapter 3. Creating keys and certificates

Most Unix-based programs that use TLS, and some Windows ones, use the OpenSSL package for cryptographic support (http://www.openssl.org/). OpenSSL provides command line programs that manipulate keys and certificates, and a cryptographic library used by these utilities and by programs such as Apache. OpenSSL is a development of an earlier package called SSLeay and this older name still appears occasionally.

Red Hat Linux, Fedora and SuSE Linux include OpenSSL as a package (normally in the `openssl` RPM). Debian provides pre-built packages that you can install for this functionality. Other Linux and Unix installations may be similar. OpenSSL can be built from source, which is available from http://www.openssl.org/source/.

OpenSSL can be built for Windows, but requires development tools that are not normally available by default. Binary copies of OpenSSL for Windows can sometimes be found with a web search - at present copies appear to be available from http://hunter.campbus.com/. Beware that having multiple copies of the OpenSSL .dll files on the same Windows server can lead to problems that are difficult to isolate.

Most command-line interactions with OpenSSL use the **openssl** command, which itself accepts a sub-command and a range of command-line arguments. These sub-commands and arguments can be confusing and there are often many different ways to achieve the same thing. On Unix systems the manual entry for **openssl** (**man openssl**) and for the individual sub-commands can be helpful. The examples below are taken from a Unix system - appropriate changes will be needed under Windows.

Creating a RSA public key pair

To generate an RSA key pair we use the **genrsa** sub-command.

```
$ openssl genrsa -des3 -out WWW.key 2048
Generating RSA private key, 2048 bit long modulus
...................+++
...................+++
e is 65537 (0x10001)
Enter pass phrase for WWW.key: password
Verifying - Enter pass phrase for WWW.key: password
```

Arguments used

`-des3`

encrypt the result using DES3

`-out`

store the result in this file

`2048`

requested key length

openssl requires a source of randomness in order to generate these keys. On modern Unix systems this is normally derived automatically from a random number source in the kernel. On other systems, and under Windows, it may be necessary to use the `-rand` argument to supply **openssl** with one or more files containing rapidly changing data.

The generated RSA key pair is encrypted using the supplied pass phrase, since the private component of the pair must remain private. The pass phrase itself must therefore be kept secret but must also not be forgotten, or lost when the only person who

knows it leaves, etc. Without it the keys (and any certificates based on these keys) become useless.

Viewing the key pair

The keys are encoded using plain text in a format sometimes called "PEM". The underlying binary format, which is also sometimes seen, is normally called "DER". The file itself is not particularly interesting to look at.

```
$ cat WWW.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,598A999A83DBC95A

IjmHjsD07Xf09XUbYbmPzhqM7SGWu8Cc1v6Km4RBR0o6D1PqFcDFJxOaGbAcAmQK
BBluOzqCB0yBnxCK2ICdnY3WUGN1veqXZQNWGhNP/hHIwyrReuHkzqZHerJxNE9x
290qMDfGPGtZg9CEAHnw3FEr/h0pYdTL/OjShweCKBmypvyj1II4GKSqCzhcl+zq
PPER0Z3vgqrdTYhRH94ixt+agPV+lAvtLASBPWZ+0rFV5LbMXds+sFg6qr1sbOUH
a2aXuGRBGgwyaKaP4cBT86GVd0F1Qt+g2YOSimzmvP+m81liB2hQvnroYHjRroJf
bMp8i+fBOTQja5F5RCFE2g4A6TygVW7qgED6E4XoCkRnrMEHHReTRD5X6k6YHbaG
HL5Earb/jrVOudNPDXuDdQV4V4ebXlk1YvD4Dz7XN7iWiS9eyjUnHvPp0iU4x25I
UzvINgyYvvKKRJNfo/n4qYt0oMfOSztAIQMB+grgAThDN/bDSNLzbUNt4vHAruKt
YfdR0BBfWogNIanBNxdU2GOAsxyMtliCjRnU8LmCFfZ5N3x35uFMJFJvzqsB0l5R
9RaOVWe2oxSWY1wbo7HM9OHDBldHTonXm9pSPobYeE0AwAyXmbV/KeQqYrw11HvQ
UVPQWWUhWqxqxdPOC4Gj1x4nKk6vnnnG/vaPBDOSd90J+MYVpqfmenbt8eWvs0TL
59b5JBUFJ882IocH/TrSnPKBkkk+UPbtZNFePvLq9tH6opyRm30b2Lrg5jv91vAB
wqveQndhCBC6t2CsgHYVp8neWak3yc9/pGyxf1Vg5+DpT4ao1X6BaJsMjKL8zhfW
nspuLy8MIi5J9yE+GFj0AhvuUVyOGSKIEWdZiYnlMuvwo0GVNorDidXkPuhGhkce
RXkyy/tiIjXeE+7qZW6zZRUSA6NVoMHbXqrQ+OowWm/mIdwMgRmJQbLpGEgpo5+Q
i4kHDbP1BC6hEFSj1pabs35V3wNF5dO0dIFg9D1TKUKUzKQGZKZUf9aKpDOVFcmM
Z6GE4QQZ8jkkROHXkAX/ZoszAkNXfkjz2mU7P+a4XAQ6e0szON9mwsNyq/AuLx98
Zm2ZtUZBWldKdKqtIDg9hX+QPuvaWKT21ojvC7Qq2AqjXuwYHwbfFSdD9X5fpkJc
qPdGRp4crjai4gknORRsf293TcEbZShriTCCI7+1TlS/2NkyKzM/v5CWhpi6xnuo
wwdWTi4ulq4aAEIIG+VfSlyJEB5vhNHrNLv493M/AKopJc/22d53FRV3sfRaSLm5
JZ/dA5P4cfIEpbqetxde74FLVGd2BfooIxjxHug68nzdJGP794BAFFAfOxllyyGv
8NW0yY9B/iaN95Qm+D47/9spCwGJCtQhYRW0j1DzJdEuGDHwDlRYuwr9iXq2bl0Y
6uZ413sVlxAnBLvouYf4JxRMiR+Pl2ZyOr2fIlAeeAvZXkvlmJb1UhD96vqVWTAP
lnR9D9CbdhlsZFwgl+Mwo/1OSE9dfyhw5DyrD7/B8df2nffASngniaGv+LBXLF3Y
EYF4akBMjB3NR3crMCvZHhDoXmDrBCqQdz0bI3ubeWoFiTWDxzMRug==
-----END RSA PRIVATE KEY-----
```

However we can look to see what is actually in the key file using the **openssl**'s **rsa** sub-command. To do this we need the pass phrase, and doing this in public would normally be a bad idea since it reveals the private key.

```
$ openssl rsa -in WWW.key -noout -text
Enter pass phrase for WWW.key: password
Private-Key: (2048 bit)
modulus:
    00:c7:62:3b:8c:8c:4a:5d:7f:08:1d:51:96:e9:1b:
    3e:92:ab:a8:97:4f:de:c9:a0:42:c3:61:bf:72:48:
    9e:2d:78:ea:f0:3b:ba:0e:e9:02:2f:9c:14:07:9f:
    fd:37:a0:a2:22:e2:c7:b8:a7:ec:eb:6b:e3:81:da:
    17:0f:dd:e9:90:6e:aa:4a:e0:8e:4c:f0:b1:2b:2a:
    41:0e:65:e0:b4:c0:29:e2:61:86:8b:09:ea:00:15:
    ad:38:5a:8f:92:83:28:28:67:ec:69:47:3e:98:b5:
    a8:6f:ef:ae:3e:bb:81:80:9d:83:c2:89:a4:77:c7:
    17:eb:01:1d:69:36:20:33:86:69:8f:9c:f0:dc:cf:
    c2:38:e7:27:86:28:85:9d:36:86:e1:2c:77:ba:97:
    e6:a4:a8:8c:0f:8e:2e:d0:45:d6:5f:a3:53:bd:c2:
    10:19:80:d3:33:8a:0e:2a:4c:3c:98:74:cb:c7:48:
    10:a9:09:0d:44:e3:79:47:d9:2a:08:38:eb:7e:4f:
```

```
    f1:58:96:c8:2f:8e:70:6e:37:10:02:7e:f9:82:16:
    c0:7e:a2:9f:07:76:4e:65:27:c6:4b:1a:12:1a:e5:
    49:ef:ee:e6:fc:7d:4b:cd:22:64:ac:ac:a0:d6:31:
    a1:c8:18:01:ad:9e:ef:c9:4e:06:c6:96:85:d4:90:
    0a:e1
publicExponent: 65537 (0x10001)
privateExponent:
    63:c8:17:81:29:1c:76:5a:02:97:99:a3:6a:99:85:
    e1:25:23:44:46:66:7a:85:47:a4:3c:20:f1:72:c2:
    26:83:a3:20:02:e4:04:5e:3c:07:d3:96:7a:92:68:
    c9:14:0c:d0:64:aa:0b:11:8f:11:ea:76:7b:1f:c7:
    f6:da:d9:ee:bc:53:61:11:ac:65:78:f7:51:60:de:
    19:f4:86:56:2e:ed:47:2c:03:87:45:b8:e3:bd:f5:
    68:84:79:e1:9a:dd:d8:0a:da:57:7d:9e:28:12:91:
    6f:23:86:12:43:08:76:73:5d:e3:57:bb:05:6e:8f:
    db:be:3d:17:d0:4c:a1:3b:ba:1d:21:19:30:cb:7c:
    14:a0:dc:17:4f:83:a2:99:2a:c0:e8:3d:a4:db:76:
    bc:d4:34:70:5e:21:02:32:cb:ae:d7:ec:43:af:46:
    e2:f9:4f:e0:a9:b5:dd:d6:e0:26:8f:0c:97:2f:cc:
    21:0b:70:2c:8c:8d:bd:b2:78:44:1b:d3:97:5b:65:
    21:e6:4e:6d:f0:93:a6:7d:6e:f4:be:0a:16:5e:09:
    92:70:24:95:4a:ca:97:e2:36:eb:71:a6:ae:0f:2a:
    79:25:75:8e:b3:49:23:26:d9:10:e4:12:36:d8:82:
    81:d0:72:a2:66:dc:0f:70:ca:e2:29:02:65:33:32:
    a1
prime1:
    00:f8:cd:68:a5:1d:91:f9:d8:57:f4:21:4c:bb:de:
    87:65:11:3b:49:40:78:28:9f:92:ee:b1:99:6a:ac:
    54:16:d0:c7:21:66:02:68:8f:d4:c5:86:46:1e:f3:
    a2:a6:64:73:87:75:1a:67:98:e4:50:62:0c:b7:de:
    e5:47:c4:4b:9b:5f:08:bd:af:1e:71:0d:11:44:5f:
    f3:0b:90:2e:b1:bb:16:0d:34:19:db:ea:2e:27:96:
    c3:a4:e8:c2:0f:73:fd:0a:11:3c:71:6e:bc:a3:19:
    41:bd:30:c5:de:f8:38:45:fd:27:3a:76:cc:65:e5:
    1f:08:63:31:e0:12:94:43:8d
prime2:
    00:cd:26:d6:49:24:10:cd:2b:35:d2:e7:22:0b:63:
    12:ff:b3:c9:ca:9b:55:be:2c:76:80:1f:aa:3a:db:
    77:20:88:da:64:8c:c4:25:57:af:5f:32:35:99:83:
    a6:0f:0c:d0:0d:8e:8a:bc:9d:e0:62:78:0e:53:ce:
    23:bf:1f:01:c7:ec:d5:0d:6f:d6:f8:4c:39:60:c3:
    c7:4e:c8:8a:14:92:30:d4:21:e2:db:f4:96:f0:91:
    c0:ba:13:3d:68:a3:95:56:3c:d4:88:29:12:91:d4:
    5d:11:e3:7c:34:a1:3e:24:f7:24:82:31:4c:d8:4d:
    34:ac:68:b3:9e:23:59:c5:a5
exponent1:
    00:96:a3:c7:b8:31:2f:31:16:cc:2a:03:ff:71:c0:
    4a:39:e7:34:fe:25:0a:9b:8e:02:68:83:1f:60:76:
    f6:72:d9:f5:b7:43:0c:32:42:e3:90:b4:bb:c0:01:
    c3:78:fb:58:f7:aa:ef:51:ca:40:72:6a:eb:48:68:
    ac:69:c7:6f:ff:a2:8a:a8:4e:5f:20:13:c9:60:9c:
    b7:8b:48:c0:fc:db:49:7e:b5:0c:f3:19:d6:d8:21:
    70:53:68:9a:16:c1:23:73:f4:fb:a3:b2:68:84:57:
    c6:75:c6:12:07:ee:42:24:1e:22:a2:43:4b:7e:66:
    3b:63:d8:ab:59:ff:e5:c5
exponent2:
    00:a3:75:80:63:c2:a2:c8:76:d7:69:f5:d3:c0:72:
    ee:5e:62:e8:33:d0:d4:de:b4:1a:af:37:8b:b1:5d:
    d0:6b:51:df:81:22:4f:de:d9:20:d8:9e:ee:ea:24:
    65:19:b4:c1:c9:2b:7c:0b:91:57:89:dd:d2:bc:9f:
    91:07:e5:32:cc:13:3e:26:78:a8:36:2a:b5:c5:0d:
    f9:2e:22:c7:32:60:d1:1b:14:ec:e7:08:d9:83:50:
    fe:d8:c4:1f:b7:d2:2f:59:09:1a:e6:6a:a3:6b:22:
    64:0d:ae:cd:f6:39:4b:84:b4:8e:98:55:a3:be:ec:
    b5:3d:72:27:3b:a7:3b:0e:29
coefficient:
```

```
    4c:08:15:e6:d6:9b:0b:42:a9:10:3c:1a:78:9b:9b:
    74:99:8c:b3:c1:6a:c4:d3:ea:af:d5:2a:ae:8b:78:
    a6:10:62:94:b1:7f:87:98:f2:a6:77:bc:f4:3c:13:
    10:3e:ce:94:bd:64:9d:18:7c:cd:5e:41:52:04:60:
    d9:ac:60:c7:a4:5f:5b:f8:53:19:81:a5:f9:17:f1:
    67:88:a9:c1:21:2e:d9:7f:3b:f7:e5:12:56:20:42:
    7f:0c:d1:23:95:78:a9:e4:d9:cb:dd:fb:7f:b1:e1:
    b4:59:8b:20:64:73:e4:02:c9:01:dc:ee:64:a1:ae:
    de:47:20:36:e1:a6:5b:3a
```

Arguments used

`-in`

the name of the file containing the key

`-noout`

do not output the key itself

`-text`

display the contents of the key file as text

Creating a CSR

From the key pair, we create a "Certificate Signing Request" (CSR) to send off to our chosen CA. The CSR contains the server's public key, and the other details that are to be included in your certificate, the whole thing signed by the server's private key.

The various strange field names, "Organization Unit Name", "Common Name", etc., are an inheritance from X.500. It does not normally matter greatly what you supply for each component, though your chosen CA may decline to sign your certificate if what you supply is wrong or not what they expect. However the component called "Common name" *must* match exactly the host name of your server, otherwise browsers will complain. For a host with several names (`www.department.cam.ac.uk/nymph.department.cam.ac.uk`) this should be whatever is going to appear in the URLs actually used to access the secure server.

```
$ openssl req -new -key WWW.key -out WWW.csr
Enter pass phrase for WWW.key: password
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:England
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:University of Cambridge
Organizational Unit Name (eg, section) []:Computing Service
Common Name (eg, your name or your server's hostname) []:clt1.csi.cam.ac.uk
Email Address []:jw35@cam.ac.uk

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Arguments used

`-new`

used when creating a new CSR, rather than processing an existing one

`-key`

the name of the file containing the key pair

`-out`

name of the file to receive the CSR

Creating a key and a CSR at the same time

It is possible to combine creation a key pair and CSR in a single command.

```
$ openssl req -new -newkey rsa:2048 -out WWW.csr -keyout WWW.key
Generating a 2048 bit RSA private key
..........................................................+++
.....................................................................+++
writing new private key to 'WWW.key'
Enter PEM pass phrase:password
Verifying - Enter PEM pass phrase: password
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-state]:England
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:University of Cambridge
Organizational Unit Name (eg, section) []:Computing Service
Common Name (eg, your name or your server's hostname) []:clt1.csi.cam.ac.uk
Email Address []:jw35@cam.ac.uk

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Arguments used

`-new`

used when creating a new CSR, rather than processing an existing one

`-newkey`

specification of the key to generate

`-out`

name of the file to receive the CSR

`-keyout`

name of the file to receive the key

Viewing the CSR

As with the key file, the CSR file itself is not very interesting but we can use the **req** sub-command to see what is actually inside it.

```
$ openssl req -in WWW.csr -noout -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=GB, ST=England, L=Cambridge, O=University of Cambridge,
                 OU=Computing Service, CN=clt1.csi.cam.ac.uk/emailAddress=
                 jw35@cam.ac.uk
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:a8:0a:7f:25:9c:1d:b0:e9:0c:c8:24:6d:d6:fd:
                    00:01:9d:73:d1:c2:38:73:ec:16:de:78:19:d1:69:
                    c1:1c:98:65:e1:87:aa:db:1f:47:97:9f:65:21:4b:
                    02:a3:cb:ea:76:6b:ad:b7:2e:b8:c9:5e:a0:d9:14:
                    cb:7d:32:88:6b:ed:7e:05:3e:f3:bb:ee:23:83:a1:
                    bb:e7:4d:bc:04:44:bb:36:b6:79:34:31:25:ec:84:
                    49:1f:29:0b:00:d8:1e:c2:6a:e5:5a:f2:87:e1:40:
                    e3:7f:1f:8c:5f:5e:ca:78:a1:60:71:77:99:82:a1:
                    b1:6b:09:27:56:7b:fb:24:f5:80:f2:89:fa:c1:a6:
                    27:a6:b2:f1:e2:06:7b:e5:34:db:f9:cd:8b:01:be:
                    ed:f1:70:02:ac:04:36:b0:bf:8d:e0:0e:9f:5a:a3:
                    ac:bf:b6:56:d0:8c:0d:17:78:2d:1d:bc:89:68:67:
                    32:82:b8:26:77:a0:49:56:f1:ca:71:eb:2b:a4:7f:
                    8e:d3:b8:1d:62:d4:f1:cb:40:c6:94:eb:21:e4:3a:
                    fe:7b:2c:7a:27:d8:ae:db:f5:d4:c4:b7:9b:a0:61:
                    56:aa:5a:fa:80:cb:0c:9a:66:41:ce:73:3f:c3:0e:
                    90:98:71:4f:49:2b:21:c2:28:5c:be:b2:25:40:0f:
                    bc:eb
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha1WithRSAEncryption
        31:e7:c6:9e:4b:93:10:d7:60:6e:43:14:8d:34:74:0c:69:74:
        e9:61:4d:65:e5:ad:67:bf:a2:63:3b:d6:f5:ee:1b:e0:75:01:
        91:ea:e8:84:8b:65:48:80:d7:aa:8c:0a:c1:18:5f:9e:3c:e8:
        0f:35:70:9d:41:7d:b3:31:11:59:a4:81:74:cd:12:b3:7d:81:
        4a:2d:06:a2:51:b8:e2:86:4a:a9:39:dc:a3:59:a0:3a:67:b2:
        70:40:a4:5b:3d:4b:de:bc:df:a8:37:e9:5f:7e:07:0f:90:84:
        43:df:23:f5:ea:80:c3:2b:98:cd:11:78:7d:1b:88:e5:a0:95:
        d6:ca:9e:dd:37:17:17:73:b3:f4:b2:dd:84:71:c7:0f:e4:12:
        ee:b2:6d:7a:9d:9d:68:89:64:57:d3:f1:a5:2d:16:bc:1d:63:
        7b:a7:13:49:b1:fb:14:26:d6:fe:17:30:df:be:4d:5b:67:57:
        d4:76:e2:44:4e:96:7e:53:84:7a:af:24:1c:9e:45:b4:11:02:
        08:36:b3:d8:3c:06:3b:ed:b7:0d:ab:aa:04:02:c3:7e:15:8d:
        98:bc:47:34:14:09:ff:76:20:7d:75:0c:93:24:93:94:7c:1c:
        0e:90:8c:f4:7f:bf:74:20:30:6c:ff:ed:dc:83:96:28:77:92:
        ac:d2:d9:c4
```

Arguments used

`-in`

the name of the file containing the key

`-noout`

do not output the key itself

`-text`

display the contents of the key file as text

Getting a real certificate

Having created the CSR, you send it off to your chosen CA (typically by pasting it into a web form or including it in an e-mail) together with other information about the organisation running the site, and details of how you plan to pay for your certificate. The CA will probably want further supporting hard copy documentation, in particular to demonstrate that you really represent the organisation described in the request and that the domain name entered in "Common Name" is registered to you.

Within the University of Cambridge, the Computing Service acts as an agent for a global CA (currently Thawte Consulting) and is able to locally administer certificates for computers with hostnames in cam.ac.uk. See http://www.cam.ac.uk/cs/tlscerts/ for further details of this arrangement.

Once they are satisfied (and you have paid them) your CA will send your certificate back to you, probably by e-mail or as a password-protected web download.

Viewing the certificate

The **x509** sub-command will let us see what is inside the certificate.

Note how validity dates are included in the certificate. These are an important security measure, since they limit the amount of time a stolen certificate will remain a threat. Unfortunately they also provide a tool that allows CAs to extract money from clients on a regular basis.

```
$ openssl x509 -in WWW.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            69:1f:68:82:22:df:92:cf:b8:f0:e1:2c:23:19:b6:8d
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=ZA, ST=FOR TESTING PURPOSES ONLY, O=Thawte Certification,
                OU=TEST TEST TEST, CN=Thawte Test CA Roo
        Validity
            Not Before: Mar 15 13:50:40 2007 GMT
            Not After : Apr  5 13:50:40 2007 GMT
        Subject: C=GB, ST=England, L=Cambridge, O=University of Cambridge,
                 OU=Computing Service, CN=clt1.csi.cam.ac.uk
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:a8:0a:7f:25:9c:1d:b0:e9:0c:c8:24:6d:d6:fd:
                    00:01:9d:73:d1:c2:38:73:ec:16:de:78:19:d1:69:
                    c1:1c:98:65:e1:87:aa:db:1f:47:97:9f:65:21:4b:
                    02:a3:cb:ea:76:6b:ad:b7:2e:b8:c9:5e:a0:d9:14:
                    cb:7d:32:88:6b:ed:7e:05:3e:f3:bb:ee:23:83:a1:
                    bb:e7:4d:bc:04:44:bb:36:b6:79:34:31:25:ec:84:
                    49:1f:29:0b:00:d8:1e:c2:6a:e5:5a:f2:87:e1:40:
                    e3:7f:1f:8c:5f:5e:ca:78:a1:60:71:77:99:82:a1:
                    b1:6b:09:27:56:7b:fb:24:f5:80:f2:89:fa:c1:a6:
                    27:a6:b2:f1:e2:06:7b:e5:34:db:f9:cd:8b:01:be:
                    ed:f1:70:02:ac:04:36:b0:bf:8d:e0:0e:9f:5a:a3:
                    ac:bf:b6:56:d0:8c:0d:17:78:2d:1d:bc:89:68:67:
                    32:82:b8:26:77:a0:49:56:f1:ca:71:eb:2b:a4:7f:
                    8e:d3:b8:1d:62:d4:f1:cb:40:c6:94:eb:21:e4:3a:
```

```
                fe:7b:2c:7a:27:d8:ae:db:f5:d4:c4:b7:9b:a0:61:
                56:aa:5a:fa:80:cb:0c:9a:66:41:ce:73:3f:c3:0e:
                90:98:71:4f:49:2b:21:c2:28:5c:be:b2:25:40:0f:
                bc:eb
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        X509v3 Basic Constraints: critical
            CA:FALSE
        X509v3 Extended Key Usage:
            TLS Web Server Authentication, TLS Web Client Authentication
        X509v3 CRL Distribution Points:
            URI:http://crl.thawte.com/ThawtePremiumServerCA.crl

        Authority Information Access:
            OCSP - URI:http://ocsp.thawte.com

Signature Algorithm: sha1WithRSAEncryption
    aa:58:81:f6:c3:ad:4e:b6:40:dc:e4:8c:c8:4d:93:a0:02:e3:
    d7:2c:64:47:7c:91:35:d3:db:b5:0a:44:3c:32:67:bd:6f:a0:
    c5:c4:fb:89:96:de:fc:4b:5c:f3:a5:18:49:78:e4:e4:0c:23:
    94:7c:98:b8:93:2e:ab:53:f2:17:30:b6:08:95:94:22:3e:85:
    de:1f:4a:1e:9b:8b:1f:50:1c:0b:08:08:a5:45:ca:84:59:92:
    65:29:2b:79:b4:32:ca:67:21:01:72:9e:22:53:b7:a3:89:64:
    21:c9:bc:5d:32:52:5d:85:16:97:87:fe:ae:97:55:ab:c1:60:
    ab:e3
```

Arguments used

`-in`

the name of the file containing the key

`-noout`

do not output the key itself

`-text`

display the contents of the key file as text

Self-signed certificates

The cost and administrative hassle of arranging to have a CSR signed by a real CA are clearly not worthwhile if all you want to do is to experiment with using HTTPS. For this sort of application it is possible to create a "self signed" certificate in which, in effect, you assert your own identity. Such a certificate will not be trusted by browsers, and they will typically display warning messages when a site protected by such a certificate is accessed.

In addition to displaying a warning message, Internet Explorer 7 turns the browser address bar red when accessing such a site.

Encouraging general web site visitors to accept such warnings is extremely dangerous, since doing so undermines much of the security that TLS provides. However, self-signed certificates have their place, and one can be created by adding the `-x509` option to an **openssl req** command.

```
$ openssl req -new -newkey rsa:2048 -x509 -keyout self.key -out self.crt
Generating a 2048 bit RSA private key
.....................................................++++++
.............................................++++++
writing new private key to 'self.key'
Enter PEM pass phrase: password
Verifying - Enter PEM pass phrase:password

-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:England
Locality Name (eg, city) []:Cambridge
Organization Name (eg, company) [Internet Widgits Pty Ltd]:University of Cambridge
Organizational Unit Name (eg, section) []:Computing Service
Common Name (eg, your name or your server's hostname) []: clt1.csi.cam.ac.uk
Email Address []:jw35@cam.ac.uk
```

Arguments used

`-new`
: used when creating a new CSR, rather than processing an existing one

`-newkey`
: specification of the key to generate

`-x509`
: output a self-signed X509 certificate, rather than a CSR

`-out`
: name of the file to receive the certificate

`-keyout`
: name of the file to receive the key

Chapter 4. Configuring Apache to support TLS

Two major versions of Apache are in current use

- Apache version 1 is still supported and widely used but no longer being developed. Version 1 has never supported TLS directly, relying for this on one of two add-on packages: Apache-SSL (http://www.apache-ssl.org/) or mod_ssl (http://www.modssl.org/).
- Version 2 of Apache was released for general use in April 2002 and this includes support for TLS (based on mod_ssl from version 1) as part of its core functionality.

Apache version 1 with mod_ssl and Apache version 2 are both roughly equivalent in function as far as TLS is concerned, this course happens to use Apache version 2.2. Configuration directives used by Apache-SSL are similar but different in detail.

Red Hat Linux, Fedora and SuSE Linux include copies of Apache with TLS support. Versions of Red Hat Linux prior to version 8 include Apache 1 and mod_ssl, later versions of Red Hat Linux include Apache 2, as does Fedora. SuSE Linux provides both Apache 1 with mod_ssl and Apache 2 in version 9 and Apache 2 only in subsequent versions. Other Linux and Unix systems may be similar. It is also possible to build Apache 1 with mod_ssl, or Apache 2, from source, see http://www.modssl.org/docs/2.8/ssl_overview.html or http://httpd.apache.org/docs-2.0/install.html for details of requirements and procedures.

Apache for Windows can be built from source, but requires the commercial Microsoft Visual C++ compiler, version 5.0 or above. The Apache foundation make pre-built versions of Apache for Windows available, but at the moment these do not include TLS support. Binary copies of Apache for Windows including TLS support can sometimes be found with a web search - at present copies appear to be being maintained at http://hunter.campbus.com/ and http://www.apachelounge.com/download/

The examples that follow were taken from a Linux machine ruining SuSE Linux Enterprise Edition 10. Other Unix installations may differ slightly, for example in the paths used, but should be substantially the same. A Windows Apache installation will also be very similar, with obvious changes to pathnames and file locations. SLES includes SSL and TLS support for Apache inside the main apache2 package. In other Linux distributions the necessary support is sometimes in a seperate package, oftern called mos_ssl.

Basic Apache configuration

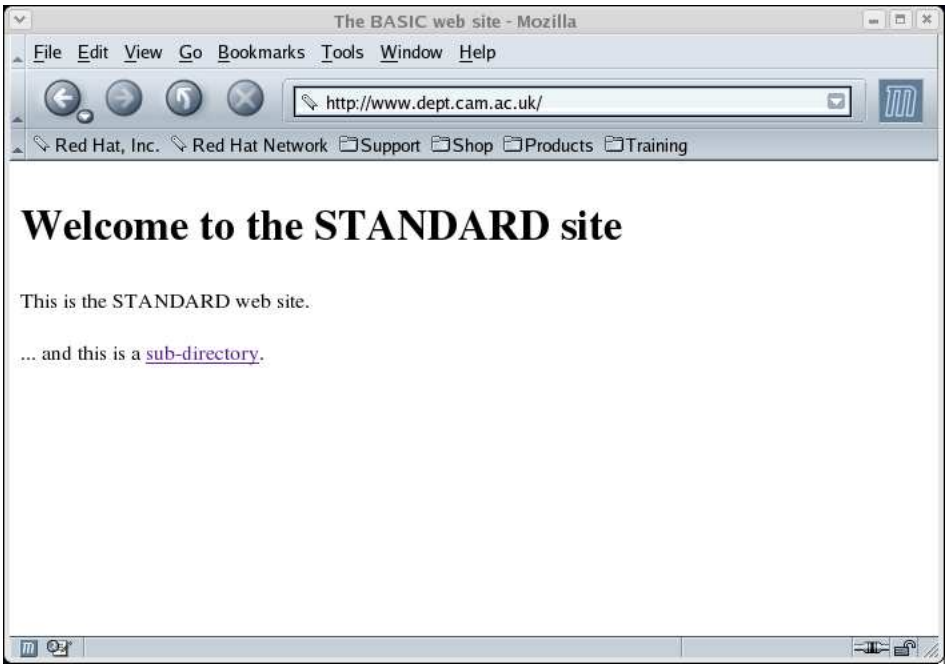
We need to build a configuration file that will instruct Apache to offer a HTTPS services. We start with a simple configuration that is just sufficient to provide a basic service over standard HTTP.

```
User        wwwrun
Group       www

# Load the modules needed for this file
LoadModule  mime_module        /usr/lib/apache2/mod_mime.so
LoadModule  dir_module         /usr/lib/apache2/mod_dir.so
LoadModule  setenvif_module    /usr/lib/apache2/mod_setenvif.so
LoadModule  log_config_module  /usr/lib/apache2/mod_log_config.so

Options None

# Set up MIME content type recognition
TypesConfig  /etc/mime.types

# Enable default documents for directory queries
```

```
DirectoryIndex  index.html

# Setup Logging
LogFormat  "%h %l %u %t \"%r\" %>s %b"  clf

# Listen on port 80 (default http)
Listen  80

<VirtualHost *:80>

  ServerName    www.dept.cam.ac.uk
  DocumentRoot  /srv/www/WWW
  CustomLog     /var/log/apache2/www.log  clf

</VirtualHost>
```

If we copy this configuration into place and restart Apache we should be able to access the site.

```
# cp conf.01 /etc/apache2/httpd.conf
# /etc/init.d/apache2 start
```



Virtual hosts and HTTPS

This configuration uses Apache's "Virtual Host" feature. According to the manual:

> The term Virtual Host refers to the practise of running more than one web site (such as www.company1.com and www.company2.com) on a single machine. Virtual hosts can be "IP-based", meaning that you have a different IP address for every web site, or "name-based", meaning that you have multiple names running on each IP address. The fact that they are running on the same physical server is not apparent to the end user.
>
> ...
>
> With name-based virtual hosting, the server relies on the client to report the hostname as part of the HTTP headers.

—The Apache Manual (http://httpd.apache.org/docs-2.2/)

It is common to use name-based virtual hosting for HTTP websites. However for HTTPS there is a problem with this. The name used to select the correct virtual host is carried in the HTTPS traffic and is therefore encrypted. Before it can be decrypted, the web server has to select an appropriate certificate to offer to the browser, but to do that it needs to know which site it is serving. This is a "Catch 22" situation. Becasue of this it is necessary to use IP-based virtual hosting if a single web server needs to deal with more then one HTTPS website. This is achieved by *not* including the `NameVirtalHost` in the configuration.

Initial HTTPS configuration

We can convert the basic configuration into one that supports an HTTPS site. We do this by listening on port 443 in place of port 80, loading the `ssl_module`, and amending the virtual host definition.

```
User        wwwrun
Group       www

# Load the modules needed for this file
LoadModule  mime_module        /usr/lib/apache2/mod_mime.so
LoadModule  dir_module         /usr/lib/apache2/mod_dir.so
LoadModule  setenvif_module    /usr/lib/apache2/mod_setenvif.so
LoadModule  log_config_module  /usr/lib/apache2/mod_log_config.so

Options None

# Set up MIME content type recognition
TypesConfig  /etc/mime.types

# Enable default documents for directory queries
DirectoryIndex  index.html

# Setup Logging
LogFormat  "%h %l %u %t \"%r\" %>s %b"  clf

# Listen on port 443 (default https)
Listen  443

# Include the SSL module
LoadModule  ssl_module  /usr/lib/apache2-prefork/mod_ssl.so

<VirtualHost *:443>

  ServerName    www.dept.cam.ac.uk
  DocumentRoot  /srv/www/WWW-SECURE
  CustomLog     /var/log/apache2/www.log  clf

  # Minimal SSL configuration
  SSLEngine  On
  SSLCertificateFile     /etc/apache2/ssl.crt/WWW.crt
  SSLCertificateKeyFile  /etc/apache2/ssl.key/WWW.key

</VirtualHost>
```

We will need to install this new configuration file, and to ensure that the certificate and key files are in place.

However as things stand there will be a problem: Apache will be unable to access the key since doing so requires the pass phrase. Apache can prompt for the pass phrase if it needs it, but this assumes a human operator will always be available when Apache starts up and this can be inconvenient, for example after a power failure. There are

various solutions to this problem (see the `SSLPassPhraseDialogue` directive), but they all involve storing a copy of the pass phrase somewhere. An alternative solution is to make a single copy of the key without its protecting pass phrase directly into the server's configuration directory. Under Unix, this file only needs to be readable by root and this may represent sufficient protection for many applications. To do this we use **openssl**'s **rsa** sub-command again.

```
# cp WWW.crt /etc/apache2/ssl.crt/
# (umask 077; openssl rsa -in WWW.key -out /etc/apache2/ssl.key/WWW.key)
Enter pass phrase for WWW.key:password
```

After a further restart, we can now access the new secure site.

Tuning the configuration

There are various additional things that we should probably add. A TLS "session cache" allows TLS sessions to be reused by subsequent connections. Apart from the efficiency gains, some versions of Microsoft Internet Explorer will not work if this is not enabled.

```
SSLSessionCache         shmcb:/var/lib/apache2/ssl_scache(512000)
SSLSessionCacheTimeout  600
SSLMutex default
```

We should also indicate how Apache can access a source of randomness for its cryptographic operations. On most Unix systems we can use `/dev/urandom` (which will probably be the built-in default anyway); Windows Apache has little choice but to depend on the generator built-in to the OpenSSL routines. See the Apache documentation for all the possible arguements to the SSLRandomSeed directives.

```
SSLRandomSeed startup builtin
SSLRandomSeed connect builtin
```

Working around browser bugs

Some browsers (Internet Explorer again) have bugs which prevent them from working correctly with some aspects of the TLS protocols. The situation is better with current browsers, but many old browsers can be a problem. We can work around this by adding the following directives to the HTTPS virtual host.

```
SetEnvIf User-Agent ".*MSIE.*" \
       nokeepalive ssl-unclean-shutdown \
       downgrade-1.0 force-response-1.0
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

These directives do two things: they alter some aspects of the HTTP and TLS protocols for MSIE, and they restrict the cryptographic primitives that will be used to those with widespread browser support. This latter is an unfortunate necessity, since it will prevent all browsers (not just MSIE) from using some useful strong ciphers.

Logging

We can enable additional logging to see what is actually happening in HTTPS transactions, in addition to the normal Apache request and error logs.

```
CustomLog /var/log/apache2/www-ssl.log \
        "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b"
```

This uses Apache's normal CustomLog directive to record information relevant to HTTPS for each connections.

%t
: Date and time

%h
: Remote host

%{SSL_PROTOCOL}x
: SSL Protocol in use

%{SSL_CIPHER}x
: SSL Cipher in use

\"%r\"
: First line of request

%b
: Bytes sent

Note that you will need to make some arrangement to rotate these newly-defined additional log files, probably by extending whatever system you use for existing webserver logs. Otherwise they will grow indefinitely.

Doing HTTP and HTTPS for the same hostname

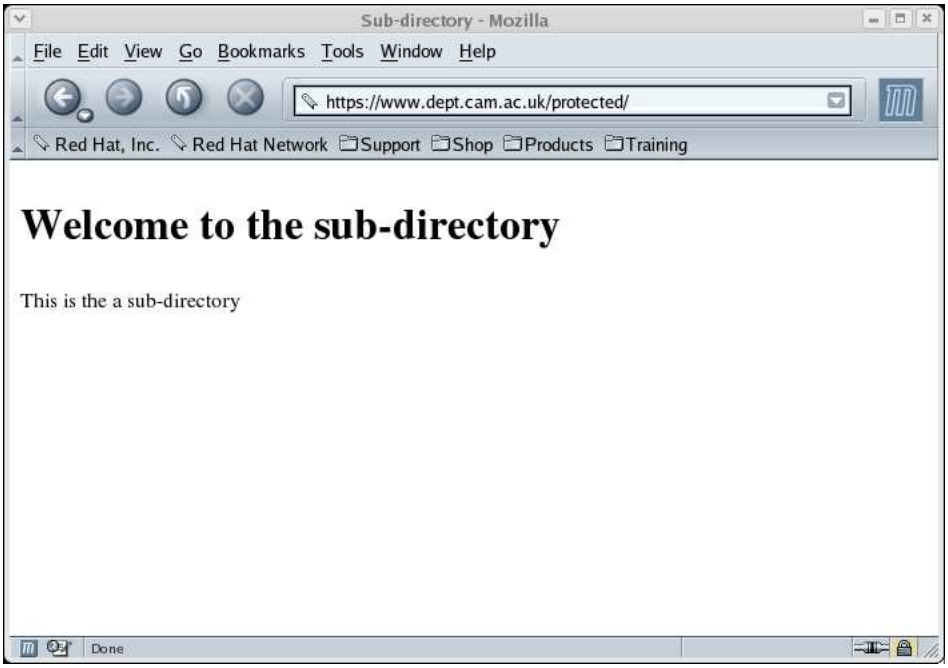
This configuration gives us a site that is only accessible by HTTPS. We might alternatively want to serve secure and insecure content from a single site, but with the content of one or more directories only available by HTTPS. We can do this by

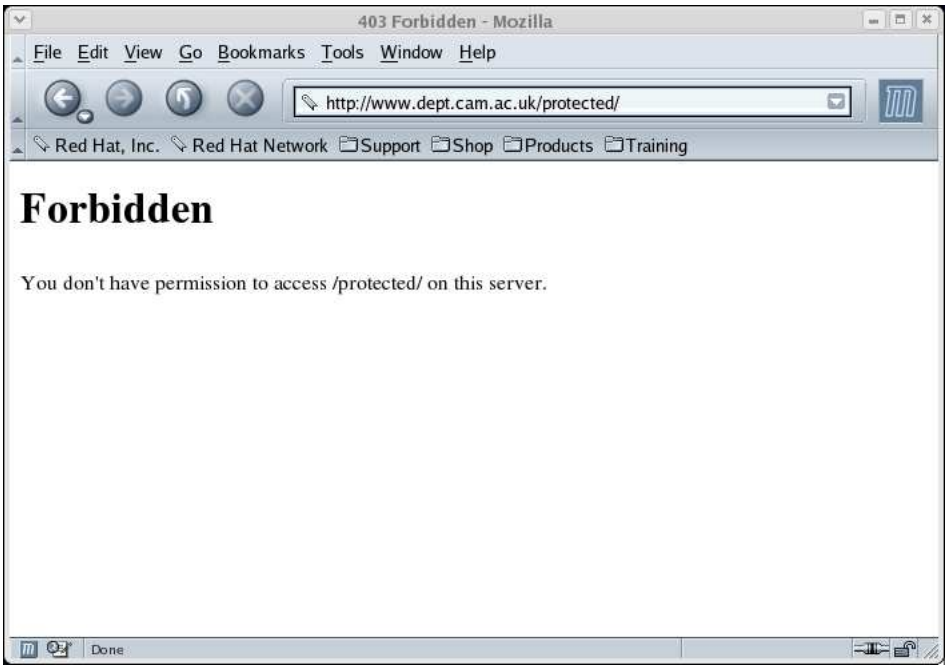
adding virtual server that accesses our content by HTTP and arranging for both virtual servers to use the same `DocumentRoot`.

```
Listen  80

<VirtualHost *:80>

  ServerName    www.dept.cam.ac.uk
  DocumentRoot  /srv/www/WWW
  CustomLog     /var/log/apache2/www.log  clf

</VirtualHost>

<VirtualHost *:443>

  ...
  DocumentRoot  /srv/www/WWW
  ...
```

Then we add the following outside all the VirtualHost blocks.

```
<Directory /srv/www/WWW/protected>
  SSLRequireSSL
</Directory>
```





Other variants on access control, for example requiring particular cipher suits, are also possible - see the `SSLRequire` directive.

Client Certificates

An optional feature of TLS is that the client can supply a certificate to the server and demonstrate its possession of the corresponding private key. This can be used for strong authentication of the user without having to transmit passwords to the server.

To obtain a client certificate, a special web form on a CA's web site causes a browser to generate a key pair and to transmit the public half in a CSR to the CA. The private half is stored in the web browser, protected by a pass phrase. In due course the CA returns the certificate which is also stored in the browser. When a server requests client authentication as part of an TLS session negotiation the browser retrieves the certificate and accesses the private key by requesting the pass phrase from the user. The browser can demonstrate that it has access to the private half of the key corresponding to the certificate by using it to encrypt random data supplied by the server. The server has access to all the information from the certificate and can make access control decisions based on that information.

Assuming we have access to a source of client certificates, we can include support for them by adding the following:

```
CustomLog /var/logs/apache2/www-ssl.log \
  "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b \"%{SSL_CLIENT_S_DN_CN}x\""

SSLCACertificateFile  /etc/apache2/ssl.crt/personalCA.crt
SSLVerifyClient require
```

The new custom log line allows us record the "Common Name" part of the "Subject" name in the client certificate. We also need to copy the CA's certificate into place so that Apache can use it to validate the personal certificate it is offered by the browser:

```
# cp personalCA.crt /etc/apache2/ssl.crt/
```

Having done this, we can apply access control in a number of ways.

- It may be sufficient to restrict access, as we are doing at the moment, to people with personal certificates issued by this particular CA.
- Alternatively the `SSLRequire` directive can be used inside a directory block to apply restrictions based on a number of parameters, including all the fields from both the client and server certificates.

```
<Directory /srv/www/WWW/protected>
  SSLRequireSSL
  SSLRequire ( %{SSL_CIPHER} !~ m/^(EXP|NULL)-/ \
      and %{SSL_CLIENT_S_DN_O} eq "University of Cambridge" \
      and %{SSL_CLIENT_S_DN_OU} in {"Computing Service", "MISD"} \
      and %{SSL_CLIENT_I_DN_O} eq "**TEST Jon's Test CA company" )
</Directory>
```

- A further alternative is to add

  `SSLUserName SSL_CLIENT_S_DN_CN`

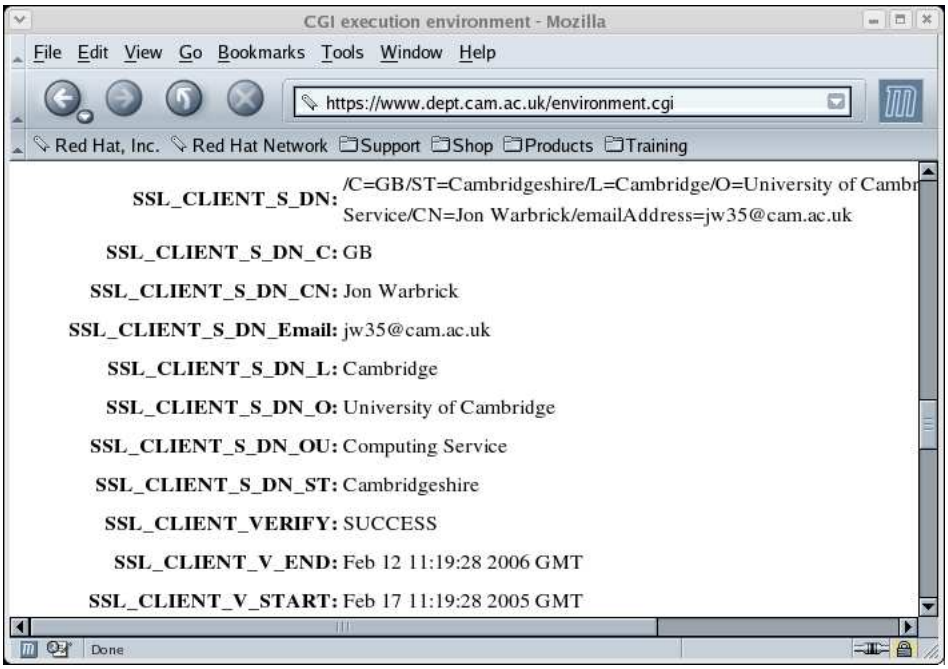
  which will cause Apache to treat the "Common Name" field from the user's certificate as if it had been entered in a normal HTTP authentication dialogue. As a result you can use standard Apache directives such as `Require` to control access on a person by person basis. This option was new in Apache 2.0 - in mod_ssl for Apache 1.3 the more limited 'SSLOptions +FakeBasicAuth' can be used to similar effect.

- Finally, adding

  `SSLOptions +StdEnvVars`

  will make environment variables containing details of the client and server certificates, TLS protocol in use, etc., available to CGI or other dynamic programs that handle requests.

In a real PKI, the CA would make available lists of certificates that have been compromised or lost. The `SSLCARevocationfile` directive lets you supply such a list to Apache so that it knows not to recognise such certificates.



Chapter 5. Other issues

Additional Directives

There are a handful of mod_ssl directives that we have not used yet. Brief references to most of them follow - see the mod_ssl documentation (http://httpd.apache.org/docs-2.2/mod/mod_ssl.html) for further information. Some of these options were not included in mod_ssl for Apache 1.3.

`SSLCertificateChainFile`

Some CAs issue certificates which are not signed directly by keys mentioned in browser root certificates. In theses cases one or more "intermediate certificates" are needed to link the server certificate to the appropriate Root certificate. These intermediate certificates are made available by the CAs and Apache needs to supply them to browsers. This directive identifies a file containing all the necessary intermediate certificates.

`SSLCADNRequestFile`

When a client certificate is requested by mod_ssl, a list of acceptable Certificate Authority names is sent to the client in the SSL handshake. These CA names can be used by the client to select an appropriate client certificate out of those it has available. The list of acceptable CA is normally all those in SSLCACertificateFile, but this directive allows a different list to be supplied.

`SSLCACertificatePath`
`SSLCARevocationPath`
`SSLCADNRequestPath`

These directives work like SSLCACertificateFile, SSLCARevocationFile, and SSLCADNRequestFile except that they identify directories containing certificate files, rather than the files themselves.

`SSLVerifyDepth`

Limits the number of intermediate certificates that will be used to verify the link between a client certificate and the appropriate CA root.

`SSLPassPhraseDialog`

Specifies various ways in which a pass phrase can be provided if needed to access a private key.

`SSLProtocol`

Allows you to chose which protocol out of SSLv2, SSLv3, TLSv1 or ALL will be accepted.

`SSLCryptoDevice`

This directive enables use of a cryptographic hardware accelerator board to offload some of the SSL processing overhead. OpenSSL support for the device is required.

`SSLHonorCipherOrder`

When choosing a cipher during an SSLv3 or TLSv1 handshake, normally the client's preference is used. If this directive is enabled, the server's preference will be used instead.

`SSLProxy...`

Various directives starting SSLProxy... allow Apache to be configured as a web proxy for SSL connections.

Note that `SSLOptions` accepts more options than have so far been mentioned, and `SSLRequire` can be used to implement a range of restrictions, not just ones related to client certificates. See the Aapche documentation for details.

Proxying HTTPS

Web proxies are an important fact of life in many Internet environments, and often provide the only means by which browsers can access the outside world. In order to support HTTPS, proxies implement a special HTTP method: CONNECT, documented in RFC 2817. On receipt of a CONNECT request, the proxy opens a TCP connection to a specified remote server and then simply passes data between the client browser and the remote server without modifying it. The client browser simply transmits its TLS data to the proxy for onward transmission to the remote server. While the proxy has access to all the data, it only sees the encrypted data stream and can do nothing with it. While this is a good thing from a security point of view it also means that none of the data can be cached.

Extended Validation

Internet Explorer 7 introduced support for 'Extended Validation' (EV) certificates. When accessing a web site that has one of these certificates from Internet Explorer the address bar turns green and a label appears that alternates between the name of the website owner, and the CA that issued their certificate. It is expected that other browsers will add similar support for these certificates in due course. In the meantime, EV certificates behave like any other certificate in other browsers.

The intension behing EV certificates is that browsers will only trust them if they are issued by 'trustworthy' CAs who have been through strict audit processes and who have robust process in place to correctly verify the identity of the people and organisations to which they are issueing certificates. This is to some extent an attempt to address the problem that browsers otherwise treat all certificates the same providing they are signed by a key coresponding to one of the trusted CA root certificates in the browser's store. Unfortunatly it is also the case that at present only a small number of the larger CAs are recognised as being able to issue EV certificates and, unsuprisingly, they all charge a significate premuim for such certificates. Since the right to issue EV certificates is vested in a trade organisation consisting mainly of these larger CAs it is unlikely that this will change anytime soon.

Server Gated Cryptography

Some Certification Authorities offer "special" certificates which claim to offer better levels of encryption than standard certificates. These are variously described as "HyperSign Certificates", "Global-Server-IDs" or "SuperCerts". These are all examples of a technology called "Server Gated Cryptography" (SGC) or "International Set-Up".

During the period of tight US restriction on the export of strong cryptography it was recognised that some applications, electronic banking being that most usually cited, really needed better cryptography than was available in export version browsers. Therefore versions of browsers from Netscape and Microsoft were shipped with strong cryptography code included but disabled by default. A small number of "approved" CA's were authorised to issue special certificates for websites of approved organisations which would unlock the strong cryptographic capability when communicating with these sites.

Since January 2000 the restrictions on export of cryptographic software have been largely removed and current browsers are able to use strong cryptography, assuming the server supports it (and most do). Therefore SGC certificates will only make a difference to connections established from old browsers, but old browsers must be assumed to contain bugs that make them unsuitable for applications where security is an issue. In addition, SGC certificates are typically *much* more expensive than standard ones, despite differing only by a few bits.

If strong encryption is necessary for a particular application then an alternative to using SCG certificates would be to configure web servers to reject weak encryption and to recommend a browser upgrade.

Appendix A. References and further information

Certification Authorities

Within the University of Cambridge, the Computing Service acts as an agent for Thawte and is able to locally administer certificates for computers with hostnames in cam.ac.uk. See http://www.cam.ac.uk/cs/tlscerts/

There are many Certification Authorities available; two of the most well known are

- BT (the UK agent for Verisign; formerly BT Trustwise and BT Ignite): http://www.btglobalservices.com/en/products/trustservices/
- Thawte: http://www.thawte.com/, now a Verisign brand.

UKERNA (the people who run JANET) have negotiated a reduced-price deal for TLS server certificates from Globalsign. See http://www.ja.net/CERT/certificates/ and http://www.globalsign.net. There were originally some "issues" with these certificates - for details see http://www-uxsup.csx.cam.ac.uk/~jw35/docs/globalsign.html - though they may now have been resolved.

General information on cryptography, SSL and HTTPS

SSL and TLS: Designing and Building Secure Systems, Eric Rescorla, Addison-Wesley 2001. ISBN: 0201615983. An extensive coverage of SSL and TLS as it applies to HTTPS and other protocols. Almost everything you could ever want to know about SSL can be found here, along with much that you probably did not want to know about.

Security Engineering, Ross Anderson, John Wiley and Sons Inc 2001;, ISBN 0471389226. An extensive coverage of security issues, including (but in no way limited to) computer security and cryptography.

Applied Cryptography, Bruce Schneier, John Wiley and Sons Inc; ISBN: 0471117099. The standard work on cryptographic algorithms.

Secrets and Lies, Bruce Schneier, John Wiley and Sons Inc; ISBN: 047125311. By the same author, reviewing computer security and the problems that occur when fallible humans start using otherwise "perfect" cryptography.

The mod_ssl *Introduction to SSL*, at http://www.modssl.org/docs/2.8/ssl_intro.html

The Apache 2 Apache SSL/TLS Encryption documentation, at http://httpd.apache.org/docs-2.2/ssl/

Introducing SSL and Certificates using SSLeay, Frederich Hirsch, at http://www.linuxsecurity.com/resource_files/cryptography/ssl-and-certificates.html.

Apache and Secure Transactions, at http://www.apacheweek.com/features/ssl

Introduction to SSL, at http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

Extended Validation Certificate, at http://en.wikipedia.org/wiki/Extended_Validation_Certificate

Server Gated Cryptography in the file `README.GlobalID` included with the mod_ssl source.

Software

Apache

http://httpd.apache.org/

mod_ssl

http://www.modssl.org/

Apache-SSL

http://www.apache-ssl.org/

OpenSSL

http://www.openssl.org/

Standards

RFCs

Many of the protocols and concepts mentioned in this course are described in RFCs. The University has a local copy of all RFCs at http://www-uxsup.csx.cam.ac.uk/pub/doc/rfc/ Relevant RFCs include

RFC1939

Post Office Protocol - Version 3

RFC2060

Internet Message Access Protocol - Version 4rev1 (IMAP)3

RFC2246

The TLS Protocol

RFC2459

Internet X.509 Public Key Infrastructure

RFC2616

Hypertext Transfer Protocol -- HTTP/1.1

RFC2660

The Secure HyperText Transfer Protocol (for HTTP over SSL)

RFC2817

Upgrading to TLS Within HTTP/1.1

RFC2818

HTTP Over TLS

PKCS series

The format of various files used to hold keys, certificate signing requests and the like, and some related algorithms, are defined in the PKCS series of documents published by RSALabs (the research arm of RSA Security). See http://www.rsasecurity.com/rsalabs/pkcs/index.html for links.

PKCS #1

RSA Cryptography Standard 1

PKCS #3

Diffie-Hellman Key Agreement Standard

PKCS #5

Password-Based Cryptography Standard

PKCS #6

Extended-Certificate Syntax Standard

PKCS #7

Cryptographic Message Syntax Standard

PKCS #8

Private-Key Information Syntax Standard

PKCS #9

Selected Attribute Types

PKCS #10

Certification Request Syntax Standard

PKCS #11

Cryptographic Token Interface Standard

PKCS #12

Personal Information Exchange Syntax Standard

PKCS #13

Elliptic Curve Cryptography Standard

PKCS #15

Cryptographic Token Information Format Standard

Other standards

SSL2: The SSL Protocol, Hickman, Kipp, Netscape Communications Corp., Feb 9, 1995

SSL3: The SSL 3.0 Protocol, A. Frier, P. Karlton, and P. Kocher, Netscape Communications Corp., Nov 18, 1996

X.509 certificates: ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework, June 1997.

ASN.1: CCITT Recommendation X.208: Specification of Abstract Syntax notation One. [see also A Layman's Guide to a Subset of ASN.1, BER, and DER, at ftp://ftp.rsasecurity.com/pub/pkcs/ascii/layman.asc (or .doc, .ps, .ps.gz)]