4 Authentication, Hashing and Digital Certificates

🗀 http://buchananweb.co.uk/security00.aspx, Select **Principles of Authentication**.

4.1 Objectives

The key objectives of this unit are to:

- Provide an understanding of various authentication methods, including the usage of biometrics.
- Define the usage of digital certificates and in private key signing.
- Define the integration of hash methods, and their applications.

4.2 Introduction

The previous chapter outlined the way data can be encrypted so that it cannot be viewed by anyone other than those it is intended for. With private-key encryption, Bob and Alice use the same secret key to encrypt and decrypt the message. Then, using a key interchange method such as Diffie-Hellman, Bob and Alice can generate the same secret key, even if Eve is listening to their communications. With public-key encryption, Bob and Alice do not have the this problem, as Alice can advertise her public key so that Bob can use it to encrypt communications to her. The only key that can decrypt the communications is Alice's private key (which, hopefully, Eve cannot get hold off). We now, though, have four further problems:

- How do we know that it was really Bob who sent the data, as anyone can get Alice's public key, and thus pretend to be Bob?
- How can we tell that the message has not been tampered with?
- How does Bob distribute his public key to Alice, without having to post it onto a Web site or for Bob to be on-line when Alice reads the message?
- Who can we *really* trust to properly authenticate Bob? Obviously we can't trust Bob to authenticate that he really is Bob.

For this we will look at the usage of hashing to finger-print data, and then how Bob's private key can be used to authenticate himself. Finally, we will look at the way that a public key can be distributed, using digital certificates, which can carry encryption key. This chapter will show the importance of authentication and assurance, along with confidentiality (Figure 4.1), and the usage of biometrics.

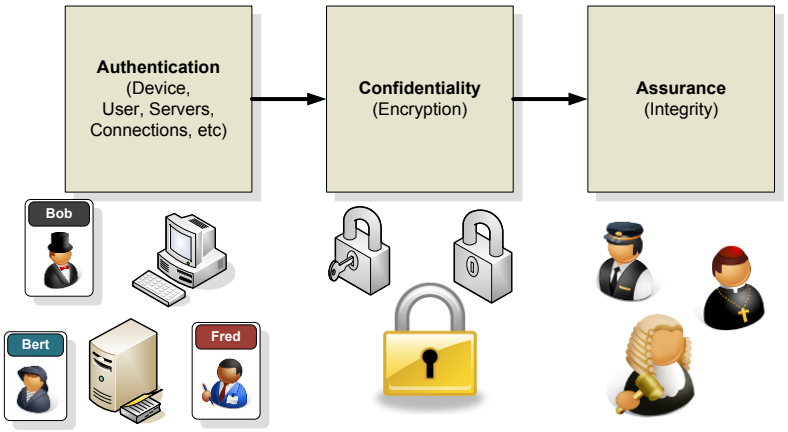


Figure 4.1 Authentication, confidentiality and assurance

A key concept in authentication is the way that different entities authenticate themselves, where it could be one end to the other, or with a mutual authentication. The main methods are: one-way server authentication; one-way client authentication; and mutual authentication (Figure 4.2). With one-way sever authentication, the server sends its authentication credentials to the client, such as a digital certification. The client then checks this, to see if it is trusted. This is the method used by SSL when a connection is made, which is used by the secure application protocols of HTTPS, FTPS, SSH, and so on.

Another key concept in authentication is that of end-to-end authentication, where the user authenticates themselves to the end service (Figure 4.3) or with intermediate authentication, where only part of the converstation between the entities is authenticated. The major problem with intermediate authentication is that only a device is typically authenticated, so a user could pretend to a valid user, by either spoofing a valid device, or by using the user's device. It is also possible to have both intermediate and end-to-end authentication, where intermediate devices can authenticate themselves to each other, where the client might also authenticate themselves to the server/service. This has the advantage of making sure that the route taken for the data packets goes through a valid route, such as for data packets between two organisational sites.

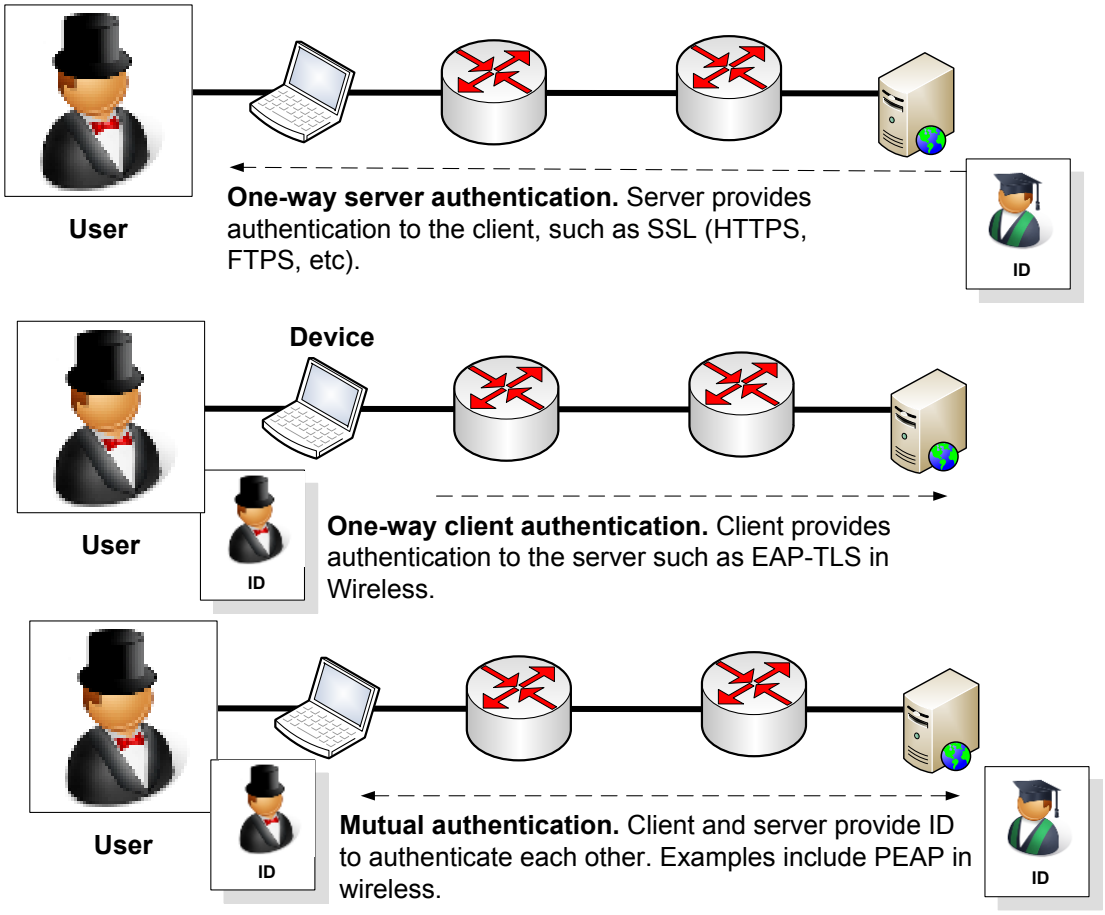


Figure 4.2 One-way and mutual authentication

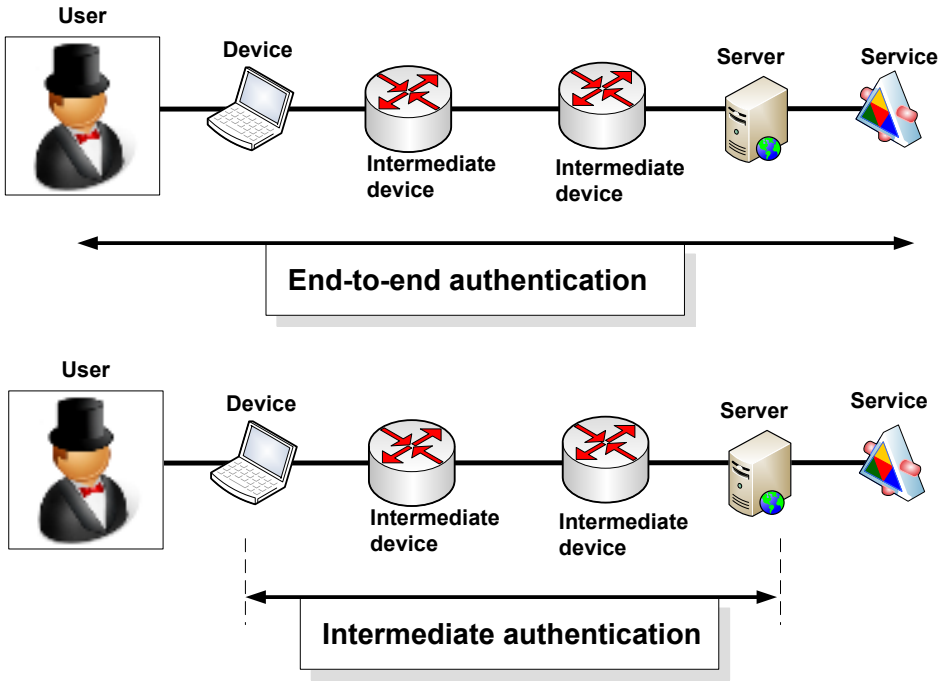


Figure 4.3 End-to-end authentication

4.3 Methods of authentication

There are many ways to authenticate devices, applications, and users, each with their strengths and weaknesses. These include:

- **Network/physical addresses**. These are a simple method of verifying a device. The network address, such as the IP address, though, can be easily spoofed, but the physical address is less easy and is a more secure implementation. Unfortunately, the physical address can also be spoofed, either through software modifications of the data frame, or by reprogramming the network interface card. Methods of authentication include DHCP, in which an IP address is granted to a host based on a valid MAC address.
- **Username and password**. The use of usernames and passwords are well known but are often open to security breaches, especially from dictionary attacks on passwords, and from social engineering attacks. In wireless networks, methods such as LEAP include a username and password for authentication, but this also is open to dictionary-type attacks.
- **Authentication certificate**. This verifies a user or a device by providing a digital certificate which can be verified by a reputable source. In wireless networks, such methods include EAP-TLS and PEAP. Sometimes it is the user/requester that has to provide a certificate (to validate the user), whereas in other protocols it is the server that is required to present a certificate to the user (to validate the server).
- **Tokens/Smart cards**. With this method a user can only gain access to a service after they have inserted their personal smart card into the computer and, typically, enter some other authentication details, such as their PIN code. In wireless networks, methods include RSA SecurID Token Card and Smartcard EAP.
- **Pre-shared keys**. This uses a pre-defined secret key. In wireless networks, methods include EAP-Archie.
- **Biometrics**. This is an improved method than a physical token where a physical feature of the user is scanned. The scanned parameter requires to be unchanging, such as fingerprints or retina images.
- **OpenID**. This type of authentication uses an URL (or XRI – Extensible Resource Identifier) to authentificate themselves from an trusted identity provider.

Unfortunately, there is often a trade-off between the robustness and authenticity of the method versus the ease of use, as illustrated in Figure 4.4. The move, though, is towards multiple methods of authentication, such as something you know?, something you have? and something you are? This is illustrated in Figure 4.5.

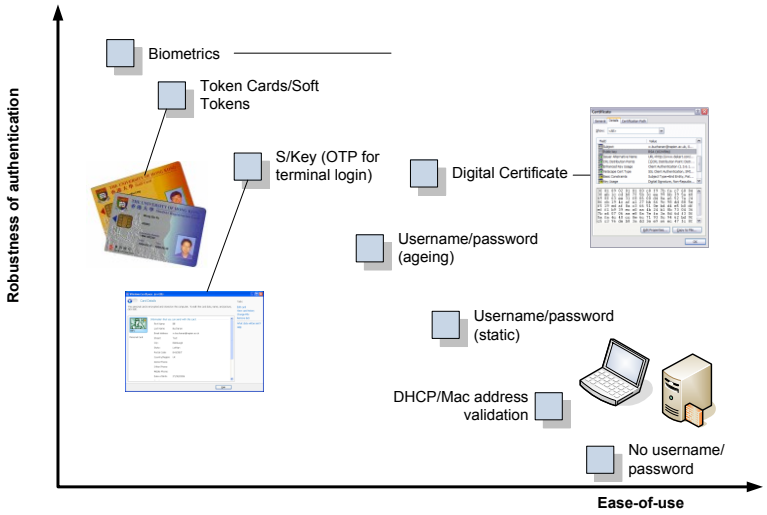


Figure 4.4 Robustness of authentication against ease-of-use

Figure 4.5 End-to-end authentication

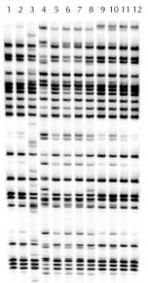
4.4 Biometrics

There are many reasons to identify a user, such as in financial applications, immigration and border control, social services, health care, network access, and in law enforcement. The accuracy of the authentication method, the cost of application, and the way that the user is scanned are obviously key factors. As we are detailing with human attributes, there are many ways of authenticating users, each of which have their weaknesses, and many are based on identifying an unchanging factor. Unfortunately traditional authentication methods, such as using passwords and digital certificates, are not perfect, and are typically open to abuse, especially with social engineering attacks. The use of biometrics, though, where a physical feature of a person is used, is thus an enhancement in secure environments, and in applications where username/passwords and physical device security are difficult. It is also a good

method in that users often do not need to memorize a password or secret phrase, or to carry a physical token. The key elements of any biometric technique are:

- **Universality**. This relates to the human features which translate to physical characteristics such as finger prints, iris layout, vein structure, DNA, and so on.
- **Distinctiveness**. This relates to the characteristics that make the characteristic unique.
- **Permanence**. This relates to how the characteristic changes over time. Typical problems might be changes of hair length and colour, over a short time, and, over a long time, skin flexibility.
- **Collectability**. This relates to the manner of collecting the characteristics, such as for remote collection (non-obtrusive collection), or one which requires physical or local connection to a scanning machine (obtrusive collection).
- **Performance**. This relates to the accuracy of identification, which is typically matched to the requirement. For example, law enforcement typically requires a high level of performance, while network access can require relevantly low performance levels.
- **Acceptability**. This relates to the acceptability of the method by users. For example, iris scanning and key stroke analysis are not well accepted by users, while hand scans are fairly well accepted. The acceptability can also vary in application domains, such as fingerprint analysis is not well liked in medical applications, as it requires physical contact, but hand scans are fairly well accepted, as they are typically contactless.

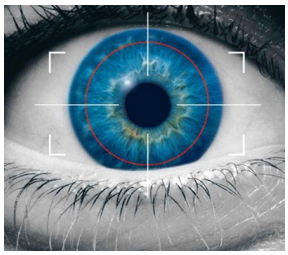
In the order of the typical correctness of the authentication method the key techniques are:

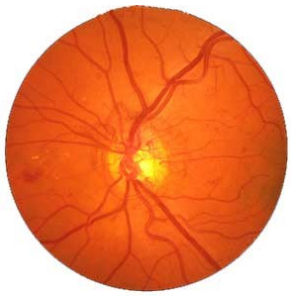


- **DNA**. This involves matching the DNA of the user, and is obviously one of the best methods of authentication, but has many legal/moral issues. It is typically only used in law enforcement applications, and also suffers from the fact that other information can be gained from DNA samples such as for medical disorders. It is also costly as a biometric method, but it is by far the most reliable. The time to sample and analyze is fairly slow, taking at least 10 minutes to analyze. Finally, the methods used to get the DNA, such as from a tissue or blood sample can be fairly evasive, but newer methods use hair and skin samples, which are less evasive.

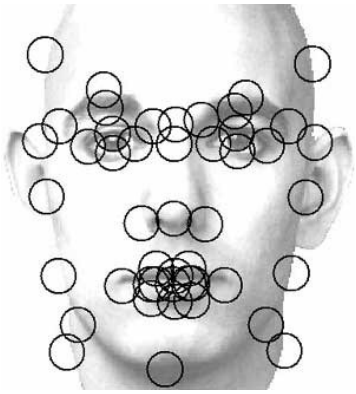
- **Finger prints**. This involves scanning the finger for unique features, such as ridge endings, sweat ports, and the distance between ridges, and comparing them against previous scans. It is one of the most widely used biometric methods, and is now used in many laptops for user authentication. Unfortunately, the quality of the scan can be variable, such as for: dirty, dry or cracked skin; pressure or alignment of the finger on the scanner; and for surface contamination. The main methods used include thermal, optical, tactile capacitance, and ultra-sound.

- **Iris recognition**. This method uses the fact that everyone has a unique iris, which is fairly complex in its pattern. This includes key characteristic markings such as the corona, filaments, crypts, pits, freckles, radial furrows and striations. It is one of the best methods of authentication, and it is extremely difficult to *trick* the system, such as with the eye of a dead person, or an artificial one. It is, though, affected by glasses which affect the quality of the image. There are, though, some ethical issues associated with this method, and it is fairly costly to implement, along with being fairly evasive in its usage, where the user must look into a special sensor machine (although mobile phones are now being fitted with iris scanners). The accuracy obviously depends on the resolution of the scanner, and the distances involved.

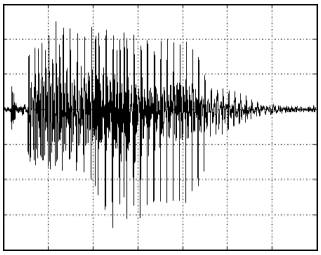




- **Retina scan**. This method shines a light into the eye, and then analyses the blood vessels at the back of the eye for a specific pattern. It is seen as a good method of authenticating users, but it does need careful alignment for creditable scans, and, because a light is shined into the eye, it may do some long term damage to the eye.
- **Face recognition**. This method scans the face for either a 2D or 3D image, and performs pattern matching to determine the likeness to a known face. Along with optical scanning, it can also use infrared (thermal) scanning, and, typically, tries to analyze a face the way that a human would. This includes the distance between the eyes, width of forehead, size of mouth, chin length, and so on. Unfortunately, it suffers from permanence factors that cause the face to change, such as facial hair, glasses, and, obviously, the position of the head. It can, though, be used as a remote sensor and an unobtrusive sensor, but the further the face is away from the scanner, typically, the poorer the matching process.



- **Hand geometry**. With this method a 2D or 3D image is taken of the hand, and the system measures key parameters, such as the length of the fingers, the position of knuckles, and so on. It is one of the most widely used methods, and is one of the most acceptable from a user point-of-view, but it can be inaccurate, and thus should be only used in low to medium risk areas. It also has the advantage that it is typically contactless, and can handle fairly high volumes of users. Its main application is typically in building/room access.



- **Vein Pattern**. This typically involves scanning the back of a hand when it is making a fist shape. The vein structure is then captured by infrared light. Finger view recognition is a considerable enhancement to this

where the user inserts their finger into a scanner, and produces good results for accurate recognition.

- **Voice recognition**. This involves analyzing speech against a known pattern for a user, as the resonance in the vocal tract, and the shape and size of the mouth and nasal cavities give a fairly unique voice print. Typically it is used with a limited range of words, such as for passwords or pass phrases. It has the advantage that it can be used remotely, especially in telephone applications, but degrades with background noise, along with changes to a users voice, such as when they have a cold, or when they've been over exercising their voice (such as after they have been singing for a length of time).
- **Keystroke**. This involves analyzing the keystrokes of a user, for certain characteristics, such as typing speed, typical typing errors, time between certain keys, and so on. It is, because of the thought of keyloggers, one of the least liked authentication methods, and also suffers from changes of behaviour, such as for fatigue and distractions. It can, though, also be matched-up with other behavioural aspects to more clearly identify the user, such as in matching up their mouse stokes, applications that they run, and so on.

- **Ear shape**. This involves analyzing the shape of the ear, and has not been used in many applications. It is normally fairly obtrusive, and can involve the user posing in an uncomfortable way.
- **Body odour**. This involves analyzing the body odour of a user for the chemicals they emit (knows as volatiles) from non-intrusive parts of the body, such as from the back of the hand.
- **Personal signature**. This involves analyzing the signing process of the user, such as for the angle of the pen, the time taken for the signature, the velocity and acceleration of the signature, the pen pressure, the number of times the pen is lifted, and so on. It is not the strongest method of authentication, as a signature pattern can be learnt, but it has the advantage that it can be integrated with the traditional method of signatures, and thus can be legally binding.

4.5 Message hash

The finger-printing of data was solved by Ron Rivest, in 1991, with the MD5 algorithm (Figure 4.6). It uses a message hash which is a simple technique which basically mixes up the bits within a message, using exclusive-OR operations, bit-shifts, and/or character substitutions. These are typically used to either provide: some form of conversion between binary and text; support the storage of passwords; or in authentication techniques to create a unique signature for a given sequence of data. The main techniques are:

- **Base-64 encoding.** This is used in electronic mail, and is typically used to change a binary file into a standard 7-bit ASCII form. It takes 6-bit characters, at a time, and converts them to a printable character.
- **UNIX password hashing**. This is used in the **passwd** file which contains the hashed version of passwords. It is a one-way function, so that it is typically not

possible to guess the password from the hashed code, but if the hashed code for the given word is known, it will always give the same hashed code. For example, the hashed version of "password" is "YigNs8zY3WzuY". Thus, as the /etc/passwd file is available in a plain text form, a user with this hashed code has the weak password of *password*. Weak passwords can obviously be beaten with a dictionary attack, where an off-line program can be used to search through a known dictionary of common words and which matches the hashed codes against the one in the passwords file. These problems have been partially overcome with a shadow password file (/etc/shadow) which can only be viewed by the administrator.

- **NT password hashing**. In most versions of Microsoft Windows, there was no password file, as in UNIX, and passwords was stored as password hashes in the Windows Registry. It is thus open to a dictionary attack in the same way that UNIX is exposed to it. Along with this, it has several other weaknesses which reduce the strength of the password. This includes converting the password into upper case between hashes, and in splitting it into two parts.
- **MD5.** This is used in several encryption and authentication methods, and is standardized in RFC1321. It produces a 32 hexadecimal character output (128-bits), which can also be converted into a text, such as shown in Figure 4.6.
- **SHA** (Secure Hash Algorithm). This is an enhanced message hash, which produces a 40 hexadecimal character output (160-bits). It will thus produce a 40 hexadecimal character signature for any message from 1 to 2,305,843,009, 213,693,952 characters. At present it is not computationally feasible to determine the original message from a SHA-1 function, or to find two messages which produce the same hash function, as illustrated in Figure 4.7. For SHA-2, it is possible to generate 256-, 384- or 512-bit signatures.

🕮 **Web link:** http://buchananweb.co.uk/security03.aspx [MD5/SHA-1]
🕮 **Web link:** http://buchananweb.co.uk/security03a.aspx [MD5/SHA-1 with Base-64]
🕮 **Web link:** http://buchananweb.co.uk/security03b.aspx [MD5/SHA-1 with salt]

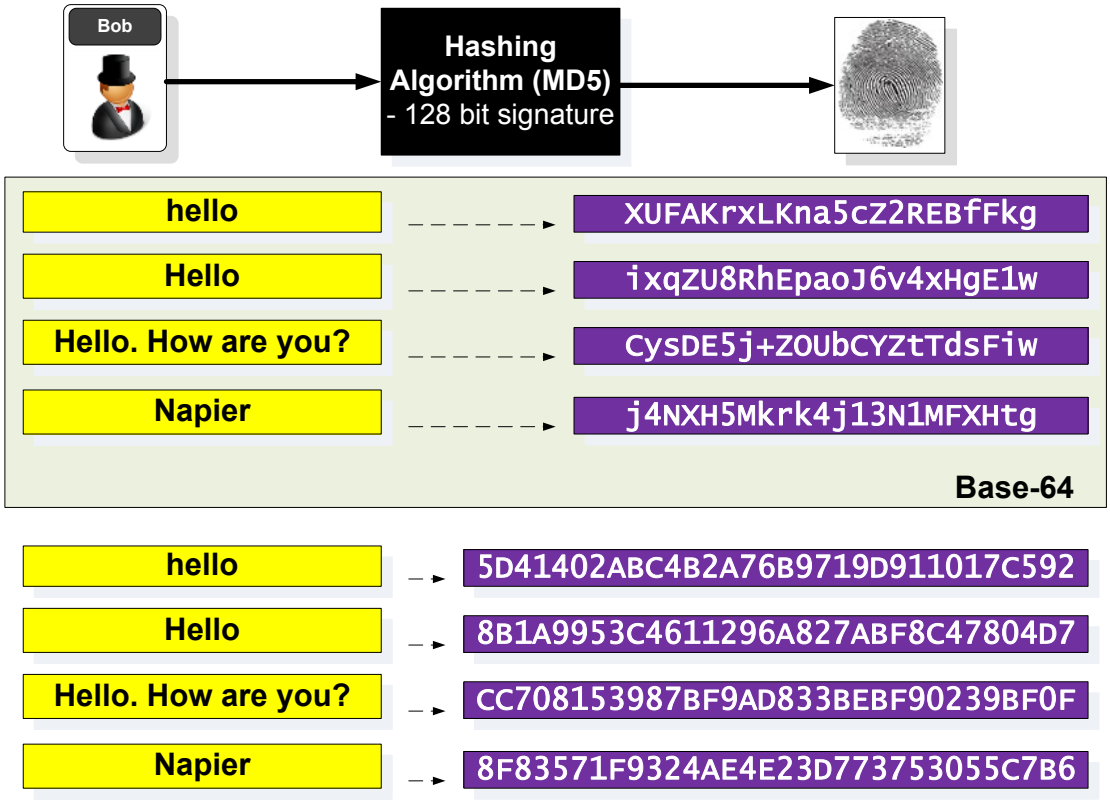


Figure 4.6 MD5 algorithm

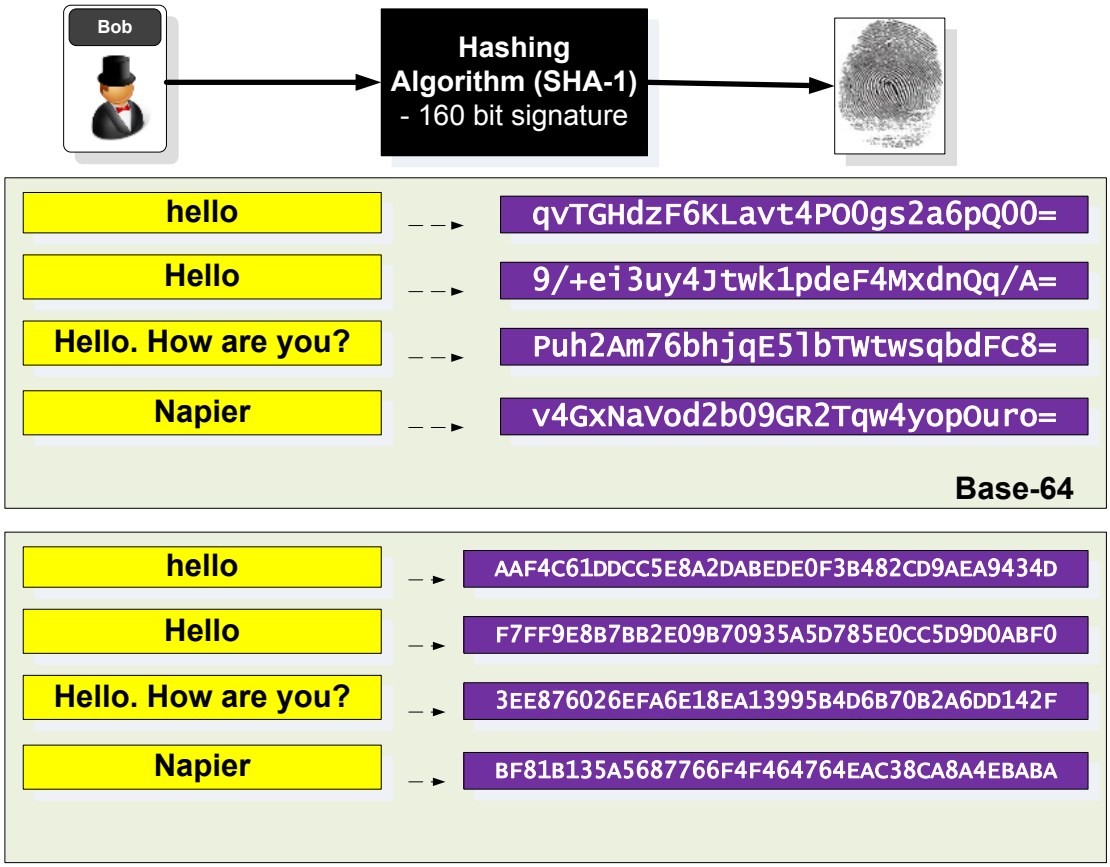


Figure 4.7 SHA-1 algorithm

For example, if a message was:

```
Hello, how are you?
Are you feeling well?

Fred.
```

then the MD5 hash for this is:

518bb66a80cf187a20e1b07cd6cef585

For example, the text:

```
Security and mobility are two of the most important issues on the In-
ternet, as they will allow users to secure their data transmissions,
and also break their link their physical connections.
```

gives:

91E2AB34D0B2DE28700A0E94071BCC46

where as:

```
Security and mobility are two of the mast important issues on the In-
ternet, as they will allow users to secure their data transmissions,
and also break their link their physical connections.
```

gives:

C0DA7FCC869C1E94687BF1CABAAB780B

It can be seen that one character of a difference changes the hash value, greatly. We can do the same for system and binary files, such as determining the message code for the DLL's in Windows\system32:

```
455D04D3EBDE98FB5AB92B7363DFF33D c:\windows\system32\6to4svc.dll
12B4C8208B5146C8D17F3F502E00A540 c:\windows\system32\aaaamon.dll
441086F355F0DEA94621984C9A3BE765 c:\windows\system32\acctres.dll
A9517EC6F8439595666925703 90C457F c:\windows\system32\acledit.dll
E92003F404A889BBADF70E8743E498B9 c:\windows\system32\aclui.dll
A68B17394C4C4DECFABEB1588E820590 c:\windows\system32\activeds.dll
9C752C5E1C5AB8A8F6D3BDA4CE87B82C c:\windows\system32\ActPanel.dll
27D39C82785A9DC831C4C2BAE5B6AE00 c:\windows\system32\actxprxy.dll
8DC922A2662C51E928B08BA50A7609F8 c:\windows\system32\admparse.dll
381915766C2A5E47A7DB95423CE09A16 c:\windows\system32\AdobePDF.dll
D05AB88927849DF74CF4F1C303DAEB4F c:\windows\system32\adptif.dll
…
```

This allows us to check that files have not been changed. The hash function is thus useful in creating a one-way function which cannot be reversed. UNIX passwords, for example, are hash functions. It has a wide scope of applications, from authenticating: users and devices; applications; and DLLs, to fingering data, files and even the complete contents of disk drives.

4.6 Authenticating the sender

The next two problems that we have is how to authenticate the sender, and also, how to prove that the message has not been tampered with in any way, even by the sender of the message. The main difference between the authentication and verification process from the encryption one, is that when Bob is sending a secret and authenticated email to Alice, Bob uses his **private-key** to encrypt an authentication message (which has been hashed), as illustrated in Figure 4.8. It can be seen that an MD5 hash is taken of the original message, and that this added to the message, and these are then encrypted with Alice's public key (Figure 4.9). This hash signature provides the authentication of Bob, and also that no-one has tampered with the encrypted message (as not even Bob can now decrypt the encrypted message). When received, the encrypted message is then decrypted (Figure 4.11) with the Alice's private-key. This gives the original message, and the encrypted hash signature. The only key which will decrypt this is **Bob's public key**, which will thus authenticate him as the sender, as only he will have the correct private key to initially encrypt the authentication message (Figure 4.12). Alice then computes the MD5 signature for the received message, and check it against the decrypted hash signature that Bob computed. If they are the same, the message has not been tampered with, and that it was really Bob that sent the email. The only major problem now is how do we send Bob's public key to Alice? The methods used with this will be covered in the sections on digital certificates (Section 4.6), PKI infrastructures (Section 4.6) and the usage of the Kerberos server (Section 4.8).

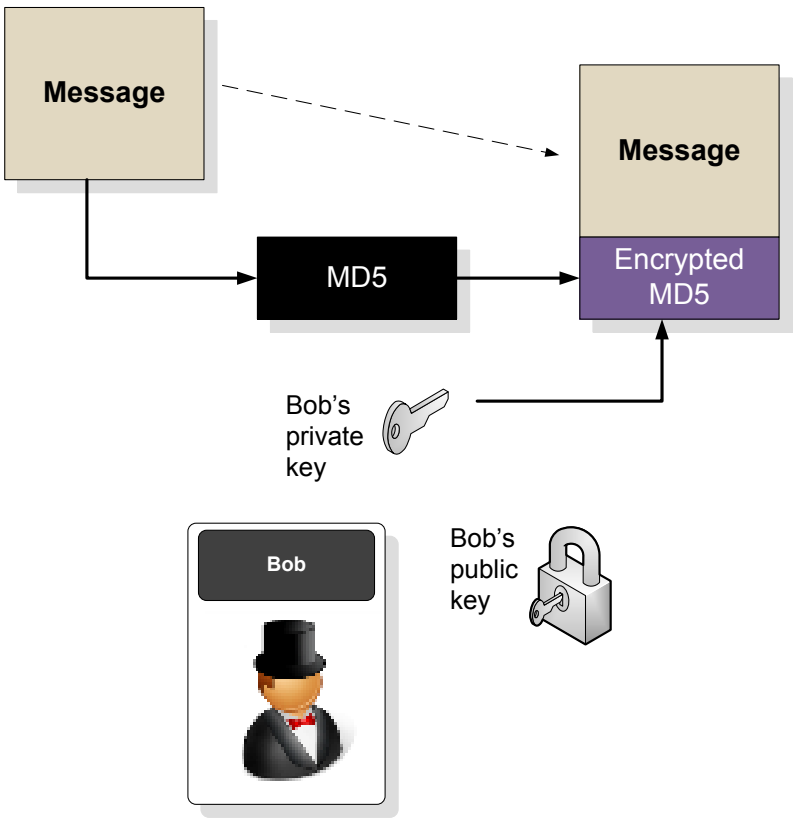


Figure 4.8 Initial part of authentication

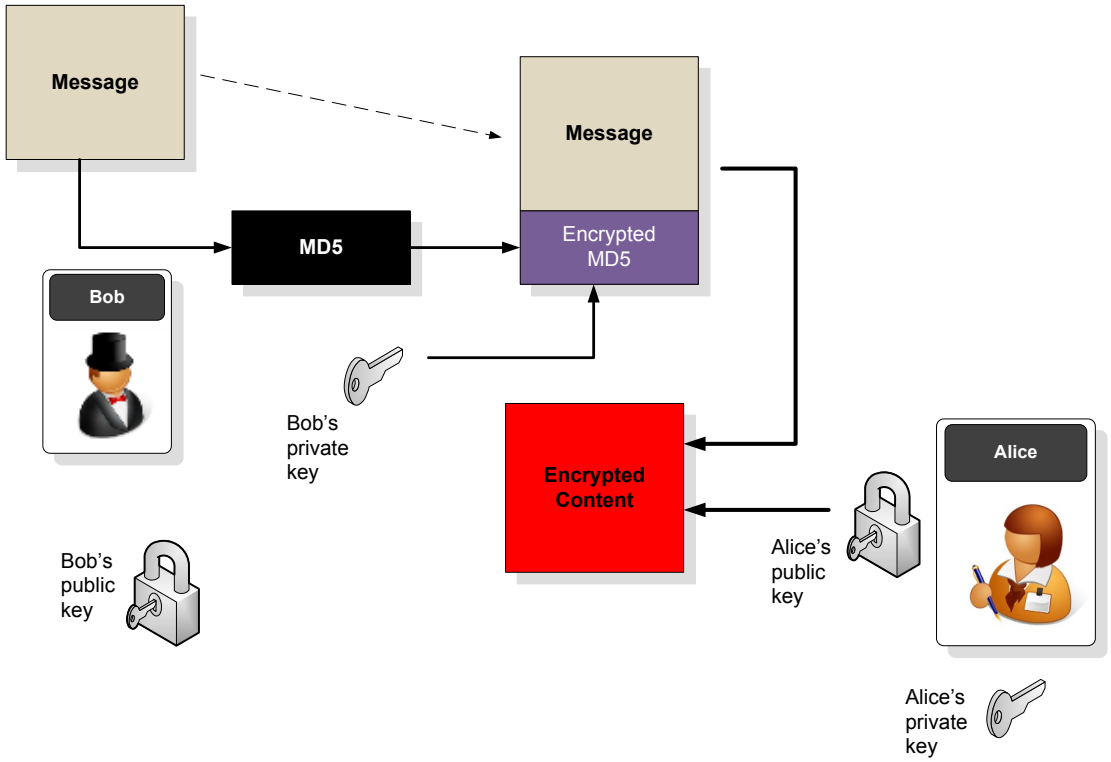


Figure 4.9 Encrypting

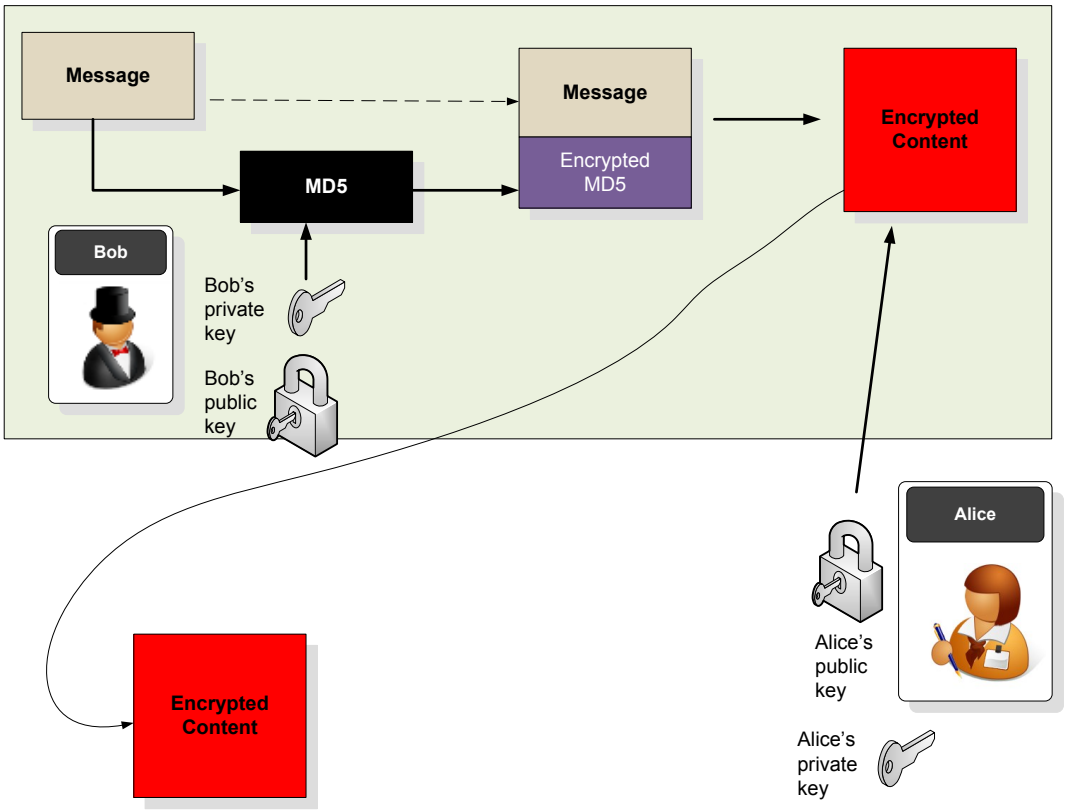


Figure 4.10 Decrypting

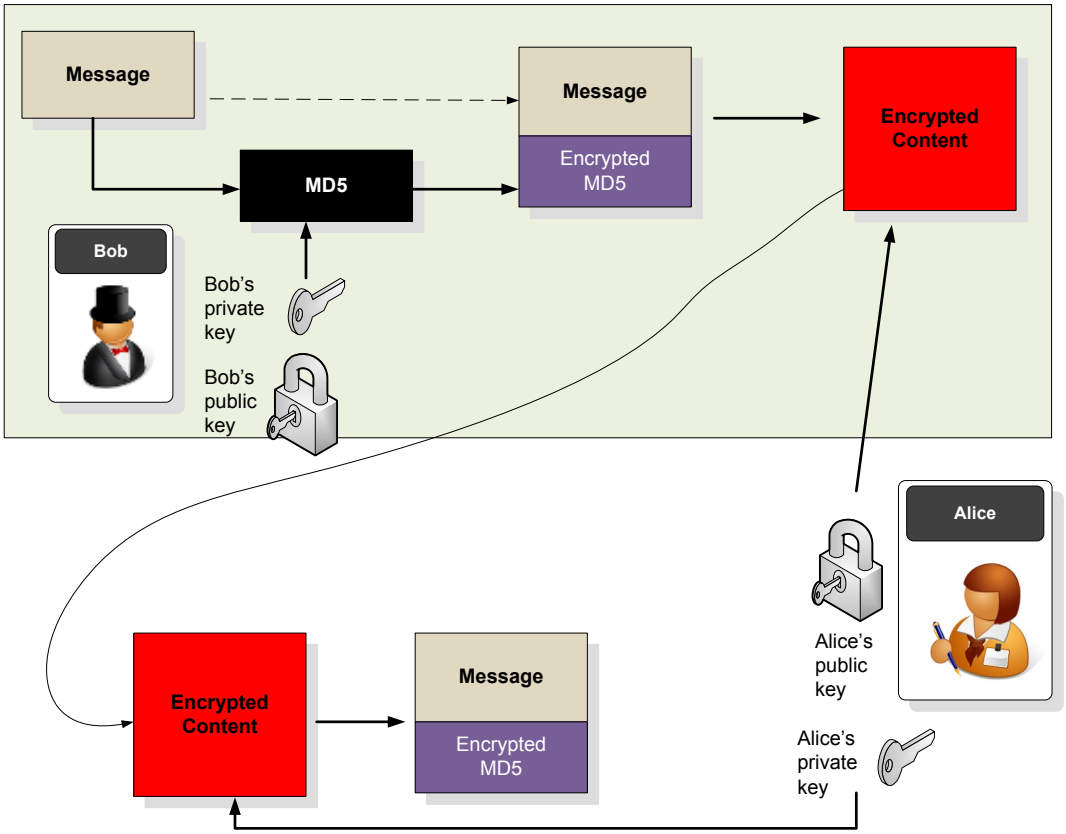


Figure 4.11 Verifying the sender

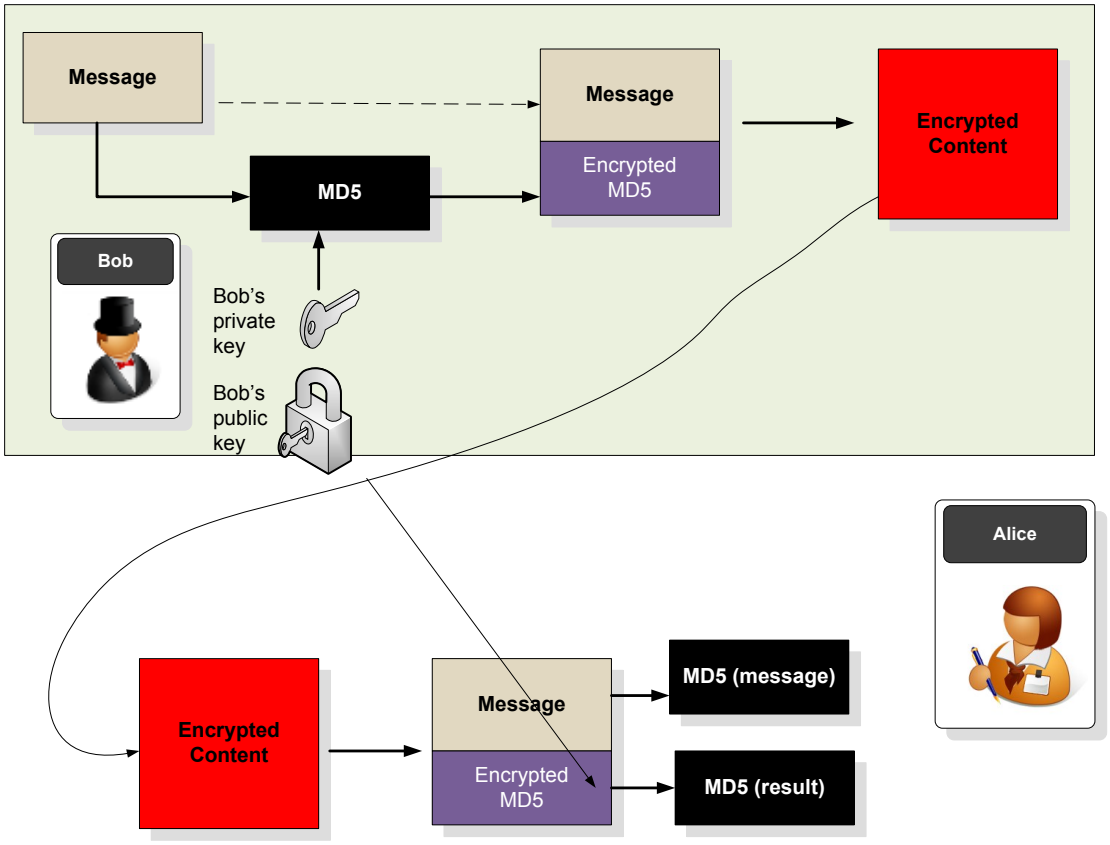
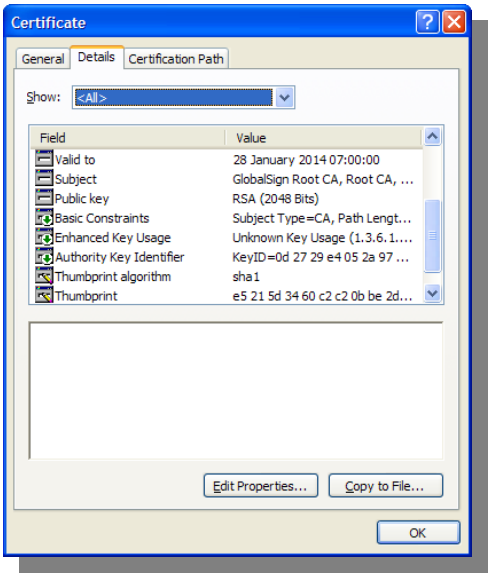


Figure 4.12 Verifying the sender

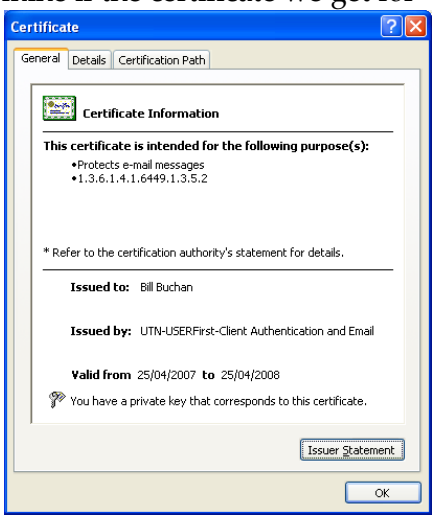
4.7 Digital certificates and PKI

From Section 4.5, we have seen that it is possible for Bob to sign a message with his private key, and that this is then decrypted by Alice with her public key. There are many ways that Alice could get Bob's public key, but a major worry for her is that who does she trust to receive his public key? One way would be for Bob to post his public key on his web site, but what happens if the web site is down, or if it is a fake web site that Alice uses. Also if Alice asked Bob for his public key by email, how does she really know that Bob is the one who is responding? Thus we need a method to pass public keys, in the verifiable way. One of the best ways is to provide a digital certificate which contains, amongst other things, the public key of the entity which is being authenticated. Obviously anyone could generate one of these certificates, so there are two ways we can create trust. One is to setup a server on our own network which provides the digital certificates for the users and devices within an organization, or we could generate the digital certificate from a trusted source, such as from well-known Certificate Authorities (CAs), such as Verisign, GlobalSign Root, Entrust and Microsoft. These are generated by trusted parties and which has their own electronic thumbprint to verify the creator, and thus can be trusted by the recipient, or not.



4.7.1 PKI and Trust

The major problem that we now have is how to determine if the certificate we get for Bob is creditable, and can be trusted. The method used for this is to setup a PKI (Public Key Infrastructure), where certificates are generated by a trusted root CA (Certificate Authority), which is trusted by both parties. As seen in Figure 4.13, Bob asks the root CA for a certificate, for which the CA must check his identity, after which, if validated, they will grant Bob a certificate. This certificate is digitally signed with the private key of the CA, so that the public key of the CA can be used to check the validity of it. In most cases, the CA's certificate is installed as a default as a Trusted Root Certificate on the machine, and is used to validate all other certificate issued by them. Thus when Bob sends his certificate to Alice, she checks the creditability of it (Figure 4.14), and if she trusts the CA, she will accept it[1]. Unfor-



[1] Unfortunately many people when faced with a certificate will not actually know if the CA is a credible one, or not, and this is the main weakness of the PKI/digital certificate system.

tunately, the system is not perfect, and there is a lack of checking of identities from CA, and Eve could thus request a certificate, and be granted one (Figure 4.15). The other method is to use a self-signed certificate, which has no creditability at all, as anyone can produce a self-signed certificate, as there is no validation of it. An example of this is shown on the right-hand side (on the previous page), where a certificate has been issued to Bill Buchan (even though the user is Bill Buchanan).

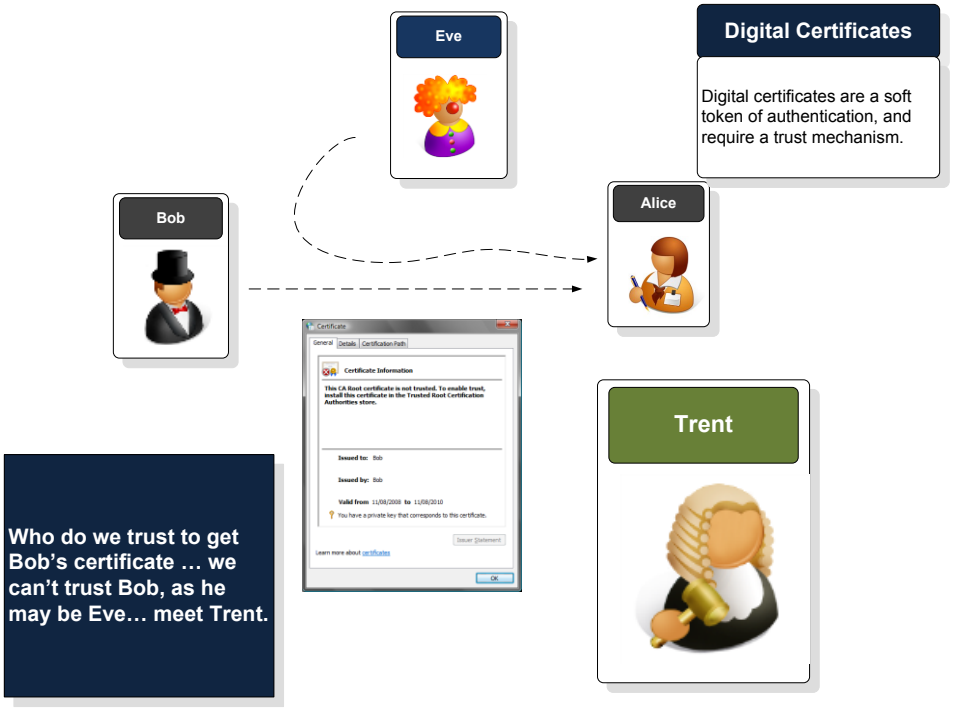


Figure 4.13 Getting a certificate

There are many cases of self-signed certificate, and of certificates which are not valid, faking the user.

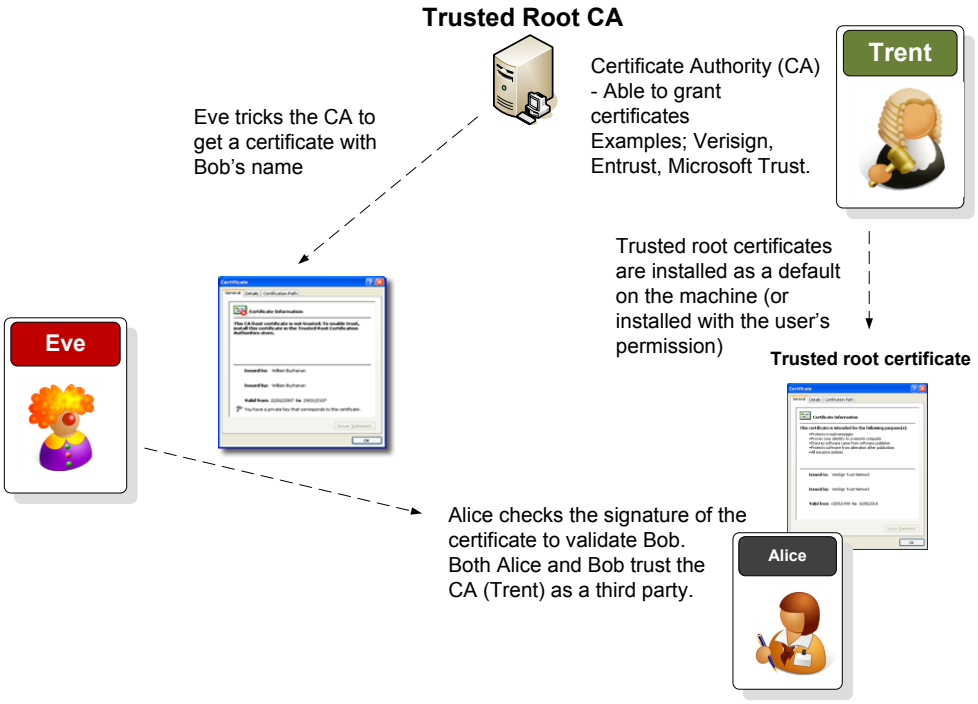


Figure 4.14 Alice checks the certificate

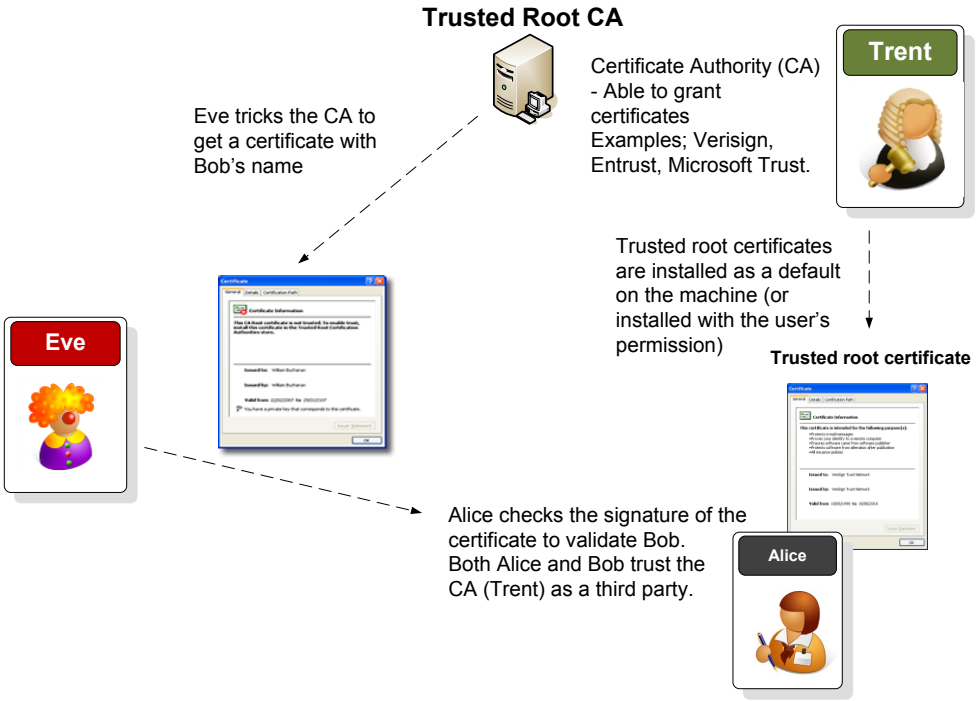


Figure 4.15 Eve spoofs Bob

Thus our trusted root CA, which we will call Trent, is trusted by both Bob and Alice, but at what level of trust? Can we trust the certificate for authenticating emails, or can we trust it for making secure network connections? Also, can we trust it to digital sign software components? It would be too large a job to get every entity signed by Trent (the root authority), so we introduce Bert, who is trusted by Trent to sign on his behalf for certain things, such as that Bert issues the certificate for email signing and nothing else. Thus we get the concept of an intermediate authority, which is trusted

to sign certain applications (Figure 4.16), such as for documentation authentication, code signing, client authentication, user authentication, and so on.

Note that there are typically two digital certificates in use. The one that is created by the CA that has both the private and public key on it (and can be stored on a USB stick, so that the encryption keys can be recovered at any time), and there is one that is distributed which **does not** have the private key (for obvious reasons).

4.7.2 Digital certificate types

Typical digital certificate types are:

- IKE.
- PKCS #7.
- PKCS #10.
- RSA signatures.
- X.509v3 certificates. These are exchanged at the start of a conversion to authenticate each device.

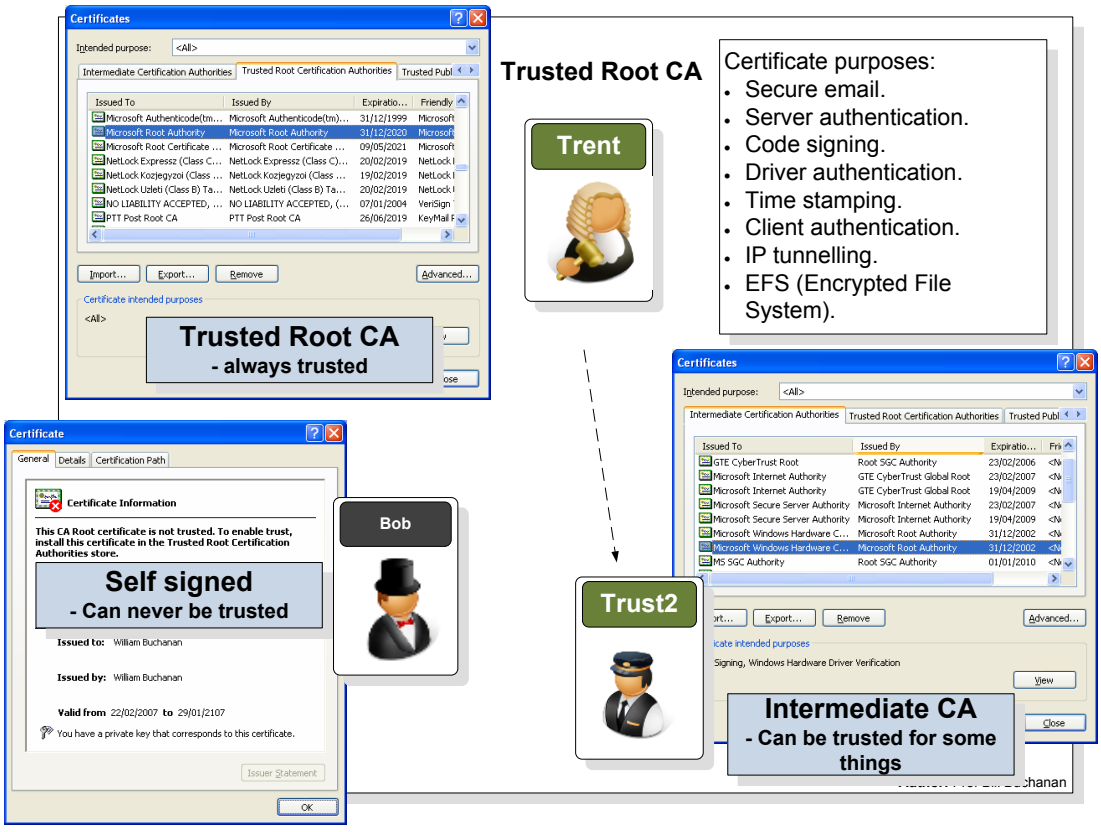


Figure 4.16 Trusted root CA, intermediate CA and self-signed

A key factor in integrated security is the usage of digital certificates, and are a way of distributing the **public key** of the entity. The file used is typically in the form of X.509 certificate files. Figure 4.17 and Figure 4.18 shows an example export process to a CER file, while Figure 4.19 shows the actual certificate. The standard output is in a

binary format, but a Base-64 conversion can be used as an easy way to export/import on a wide range of systems, such as for the following:

```
-----BEGIN CERTIFICATE-----
MIID2zCCA4WgAwIBAgIKWHROcQAAAABEujANBgkqhkiG9w0BAQUFADBgMQswCQYD
VQQGEwJHQjERMA8GA1UEChMIQXNjZXJ0aWExJjAkBgNVBAsTHUNsYXNzIDEgQ2Vy
dGlmaWNhdGUgQXV0aG9yaXR5MRYwFAYDVQQDEw1Bc2NlcnRpYSBDQSAxMB4XDTA2
MTIxNzIxMDQ0OVoXDTA3MTIxNzIxMTQ0OVowgZ8xJjAkBgkqhkiG9w0BCQEWF3cu
YnVjaGFuYW5AbmFwaWVyLmFjLnVrMQswCQYDVQQGEwJVSzEQMA4GA1UECBMHTG90
aGlhbjESMBAGA1UEBxMJRWRpbmJ1cmdoMRowGAYDVQQKExFOYXBpZXIgVW5pdmVy
…
H+vXhL9yaOw+Prpzy7ajS4/3xXU8vRANhyU9yU4qDA==
-----END CERTIFICATE-----
```

The CER file format is useful in importing and exporting single certificates, while other formats such as the Cryptographic Message Syntax Standard – PCKS #7 Certificates (.P7B), and Personal Information Exchange – PKCS #12 (.PFX, .P12) can be used to transfer more than one certificate. The main information for a distributable certificate will thus be:

- The entity's public key (Public key).
- The issuer's name (Issuer).
- The serial number (Serial number).
- Start date of certificate (Valid from).
- End date of certificate (Valid to).
- The subject (Subject).
- CRL Distribution Points (CRL Distribution Points).
- Authority Information (Authority Information Access). This will be shown when the recipient is prompted to access the certificate, or not.
- Thumbprint algorithm (Thumbprint algorithm). This might be MD5, SHA1, and so on.
- Thumbprint (Thumbprint).

The certificate, itself, can then be trusted to verify a host of applications (Figure 4.20), such for:

- Server authentication.
- Client authentication.
- Code signing.
- Secure email.
- Time stamping.
- IP security.
- Windows hardware driver verification.
- Windows OEM System component verification.
- Smart card logon.
- Document signing.

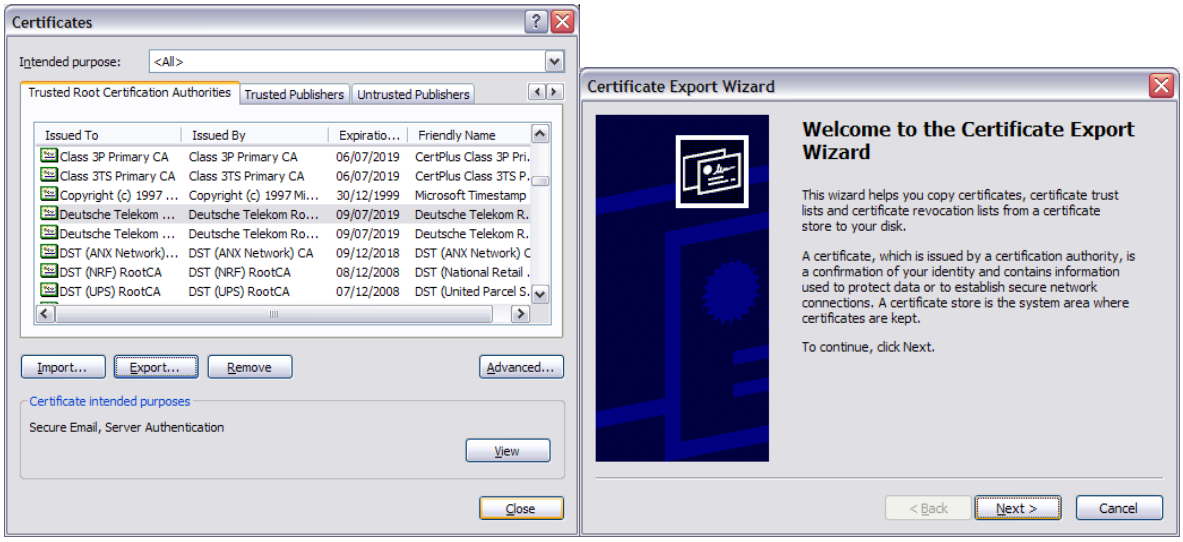


Figure 4.17 Exporting digital certificates

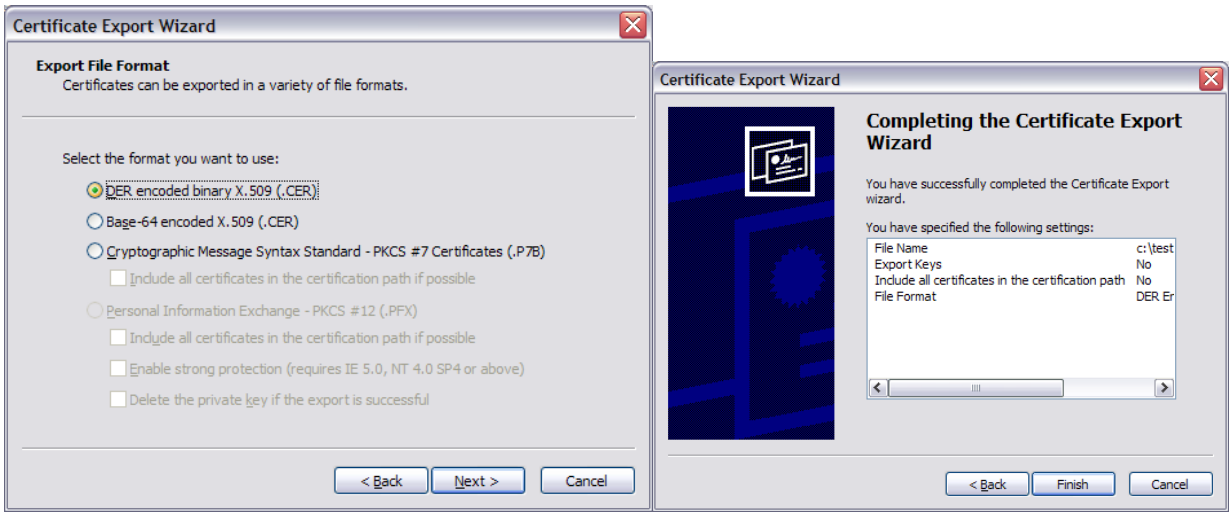


Figure 4.18 Exporting digital certificates

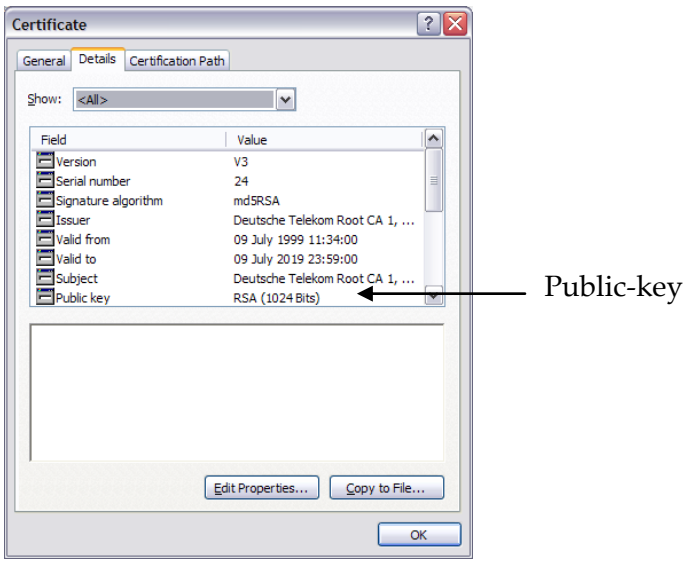


Figure 4.19 Digital certificates

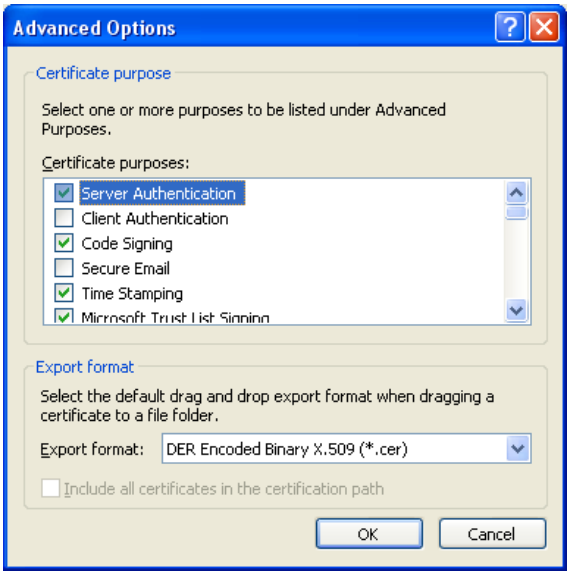


Figure 4.20 Options for signing

4.7.3 Digital Certificate reader

The C# code to read an X509 cer file is:

```
using System;
using System.Security;
using System.Net;
using System.Security.Cryptography.X509Certificates;

namespace ConsoleApplication3
{
   class Class1
   {
      static void Main(string[] args)
      {
         X509Certificate cer =  X509Certificate.CreateFromCertFile("c:\\test.cer");

         System.Console.WriteLine("Serial Number: {0}",cer.GetSerialNumberString());
         System.Console.WriteLine("Effective Date: {0}",
               cer.GetEffectiveDateString());
         System.Console.WriteLine("Name: {0}",cer.GetName());
         System.Console.WriteLine("Public key: {0}",cer.GetPublicKeyString());
         System.Console.WriteLine("Public key algorithm: {0}",
               cer.GetKeyAlgorithm());
         System.Console.WriteLine("Issuer: {0}",cer.GetIssuerName());
         System.Console.ReadLine();
      }
   }
}
```

And the output from this is:

```
Serial Number: C0DD5E19983C6F575EFE454E7E66AD02
Effective Date: 08/11/1994 16:00:00
Name: C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authority
Public key: 308185027E0092CE7AC1AE833E5AAA898357AC2501760CADAE8E2C37CEEB35786454
03E5844051C9BF8F08E28A8208D216863755E9B12102AD7668819A05A24BC94B256622566C88078FF7
81596D840765701371763E9B774CE35089569848B91DA7291A132E4A11599C1E15D549542C733A6982
B197399C6D706748E5DD2DD6C81E7B0203010001
Public key algorithm: 1.2.840.113549.1.1.1
Issuer: C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authori-
ty
```

It can be seen that this digital certificate defines the public key for the owner, and is thus a way for a user or organisation to distribute their public key. Thus if a user sends an authenticated message, they sign it with their private key, and the only key which will be able to decrypt it will be the public key contained in the digital certificate. The Microsoft .NET framework includes the digital signing for software components, which involves the creator signing them with their private key, and only the public key will be able to authenticate them. If this has changed, it will not be authenticated, or authorized.

🕮 **Web link:** http://buchananweb.co.uk/security10.aspx
🕮 **Web link:**
http://buchananweb.co.uk/e_presentations/digital_certificates_expired.htm
🕮 **Web link:**
http://buchananweb.co.uk/e_presentations/digital_certificate_exporting.htm
🕮 **Web link:**
http://buchananweb.co.uk/e_presentations/digital_certificates_showing_browser.htm

4.8 HMAC (Hash Message Authentication Code)

HMAC is a message authentication code (MAC) that can be used to verify the integrity and authentication of a message. It involves hashing the message with a **secret key**, and thus differs from standard hashing, which is purely a one-way function. As with any MAC, it can be used with standard hash function, such as MD5 or SHA-1, which results in methods such as HMAC-MD5 or HMAC-SHA-1. Also, as with any hashing function, the strength depends on the quality of the hashing function, and the resulting number of hash code bits. Along with this the number of bits in the secret key is a factor on the strength of the hash. Figure 4.21 outlines the operation, where the message to be sent is converted with a secret key, and the hashing function, to an HMAC code. This is then sent with the message. On receipt, the receiver recalculates the HMAC code from the same secret key[2], and the message, and checks it against the received version. If they match, it validates both the sender, and the message (Figure 4.22).

Let's say that the two routers in Figure 4.22 continually challenge each other to answer certain questions. Initially they negotiate a share secret key, such as "mykey" (or it could be set manually – but this will not be as secure) and negiotiate the HMAC type. So a challenge might be to "Multiply 5 and 4?". The answer would be 20, thus using HMAC-MD5, the quizzed device will return back E298452E0 CD44830FEE1DA1C765EB486 (*ref* http://buchananweb.co.uk/security01.aspx). The challenger will then do the same conversion, and if it gets the same HMAC code, it will know that the device on the other end is still the same one that it started the connetion with. If not, it will disconnect, and it looks as if the original device has been replaced with a spoofed one.

[2] Typically the secret key would either be generated by converting a pass phase into the secret key (such as in some wireless systems) or is passed through the key exchange phase at the start of the connection (such as with Diffie-Hellman).

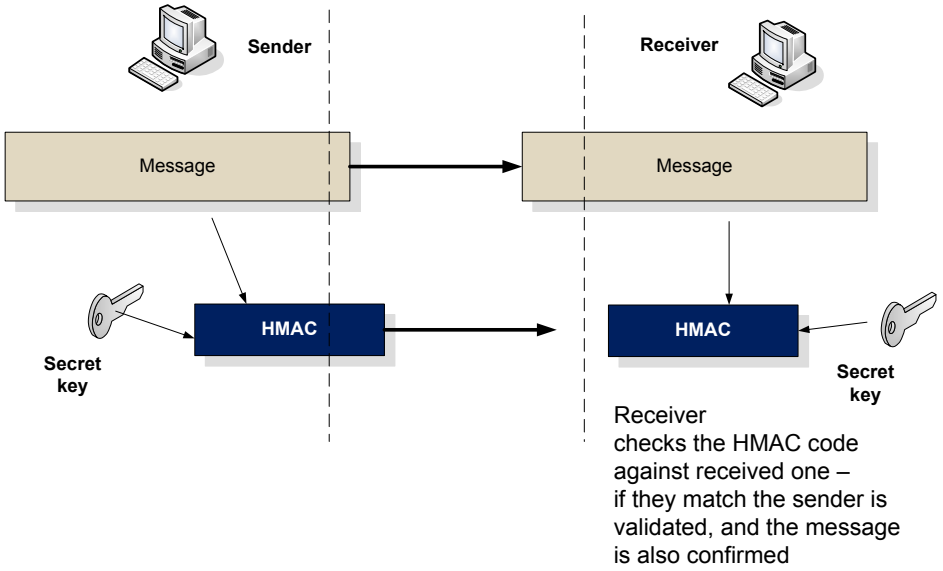


Figure 4.21 HMAC operation

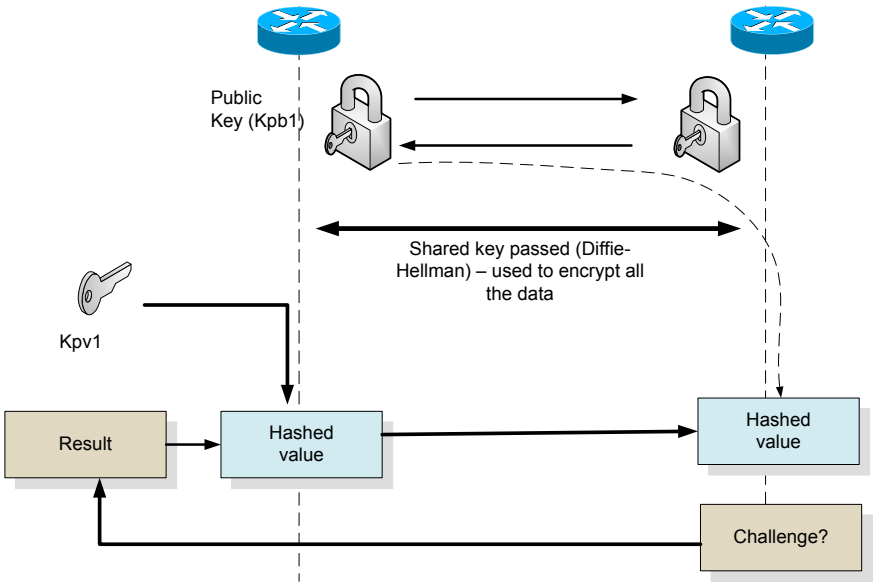


Figure 4.22 Using symmetric encryption and asymmetric authentication

The following gives some simple .NET code for HMAC conversion:

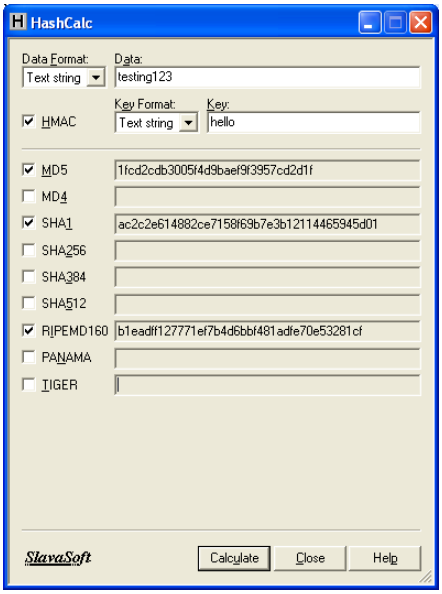
```
using System;
using System.IO;
using System.Text;
using System.Security.Cryptography;
// Verify with http://hashcalc.slavasoft-inc.qarchive.org/
// Verify: Message="testing123", key="hello" gives
ac2c2e614882ce7158f69b7e3b12114465945d01

namespace hmac
{
   class Class1
   {
      static void Main(string[] args)
      {
         string message = "testing123";
         string key = "hello";
```

```
        System.Text.ASCIIEncoding  encoding=new System.Text.ASCIIEncoding();

        byte [] keyByte = encoding.GetBytes(key);

        HMACSHA1 hmac = new HMACSHA1(keyByte);
        byte [] messageBytes = encoding.GetBytes(message);
        byte [] hashmessage = hmac.ComputeHash(messageBytes);
        Console.WriteLine("Hash code is "+ByteToString(hashmessage));
        Console.ReadLine();

     }
     public static string  ByteToString(byte [] buff)
     {
        string sbinary="";

        for (int i=0;i<buff.Length;i++)
        {
           sbinary+=buff[i].ToString("X2"); // hex format
        }
        return(sbinary);
     }
  }
}
```

For a key of "hello", and a message of "testing123" gives:

The HMAC-SHA-1 hash code is: `AC2C2E614882CE7158F69B7E3B12114465945D01`

This can be checked against a Hash calculator (on the right-hand side) to verify. The source code is at:



📖 **Source Code link:**
http:// buchananweb.co.uk/hmac2.zip

📖 **Web link:**
http://buchananweb.co.uk/security01.aspx

With HMAC, the text string is broken-up into blocks of a fixed size, and then are iterated over with a compression function. Typically, such as for MD5 and SHA-1, these blocks are 512 bytes each. With MD5 the output is 128 bits[3] and for SHA-1 it is 160 bits, which is the same as the standard hash functions. HMAC is used in many applications, such as in IPSec and in tunneling sockets (TLS).

4.9 Future of Authentication Systems - Kerberos

The major problem with current authentication systems is that they are not scalable, and they lack any real form of proper authentication. A new authentication architecture is now being proposed, which is likely to be the future of scalable authentication infrastructures – Kerberos. It uses tickets which are gained from an Identity Provider

[3] 128 bits equates to 32 hexadecimal characters (as 4-bits are used for each hex value). For SHA-1, there are 160 bits which gives 40 hexadecimal characters.

(IP – and also known as an Authentication Server), which are trusted to provide an identity to a Relying Party (RP). The basic steps are:

Client to IP:

- A user enters a username and password on the client.
- The client performs a one-way function on the entered password, and this becomes the secret key of the client.
- The client sends a cleartext message to the IP requesting services on behalf of the user.
- The IP checks to see if the client is in its database. If it is, the IP sends back a session key encrypted using the secret key of the user (MessageA). It also sends back a ticket which includes the client ID, client network address, ticket validity period, and the client/TGS (Ticket Granting Server) session key encrypted using the secret key of the IP (MessageB).
- Once the client receives messages A and B, it decrypts message A to obtain the client/TGS session key. This session key is used for further communications with IP.

Client-to-RP:

- The client now sends the ticket to the RP, and an authentication message with the client ID and timestamp, encrypted with the client session key (MessageC).
- The RP then decrypts the ticket information from the secret key of the IP, of which it recovers the client session key. It can then decrypt MessageD, and sends it back a client-to-server ticket (which includes the client ID, the client network address, validity period, and the client/server session key). It also sends the client/server session key encrypted with the client session key.

The Kerberos principle is well-known in many real-life authentication, such as in an airline application, where the check-in service provides the authentication, and passes a token to the passenger (Figure 4.23). This is then passed to the airline security in order to board the plane. There is thus no need to show the form for the original authentication, as the passenger has a valid ticket.

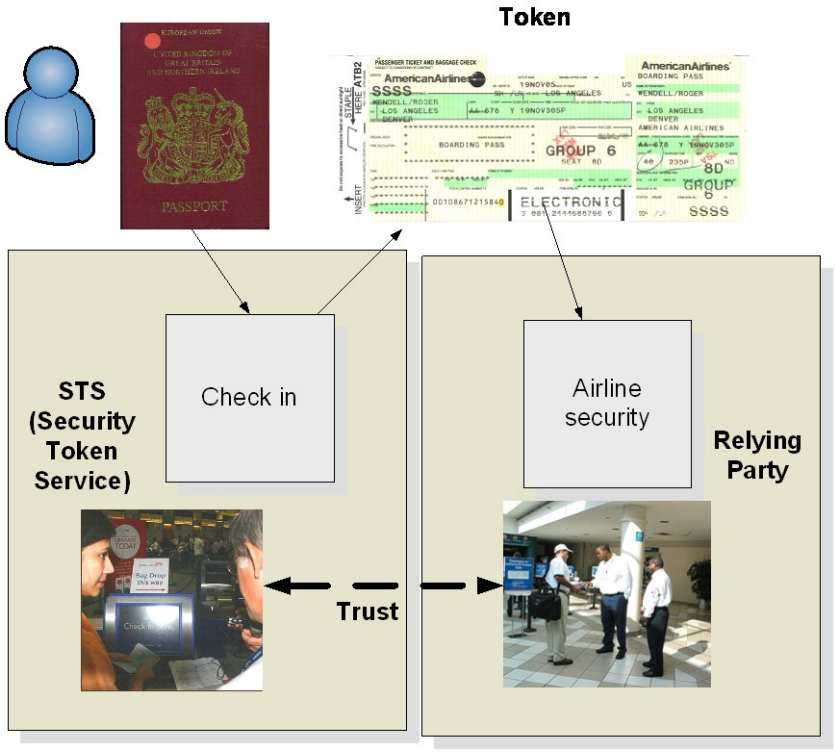


Figure 4.23 Ticketing authentication

4.9.1 Microsoft CardSpace

The Microsoft .NET 3.0 framework has introduced the CardSpace foundation framework, which uses Kerberos as its foundation. For this it defines a personal card, which is encrypted and created by the user, and contains basic users details on the user, such as their name, address, email address, and so on. A **managed card** is created by an IP (Identity Provider) and validates the user. The managed card thus does not keep any personal details on login parameters and bank card details (as these are kept off-site). The user can thus migrate one from machine to another, and migrate their card (Figure 4.24). A personal card, of course, does not require an IP, and a card can be passed directly to the RP (Figure 4.25).

For a managed card, the basic steps are defined in Figure 4.26. An additional advantage of CardSpace is that it supports both PKI authentication (using digital certificates) and Kerberos, using standard protocols (all of which XML-based and open protocols) – such as shown in Figure 4.27.

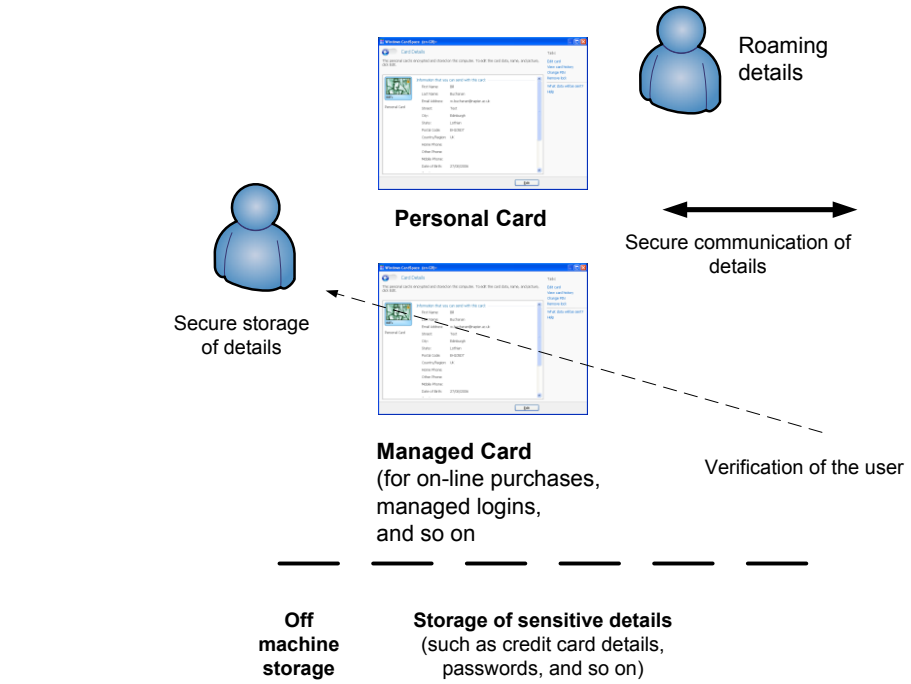


Figure 4.24 Personal and managed cards

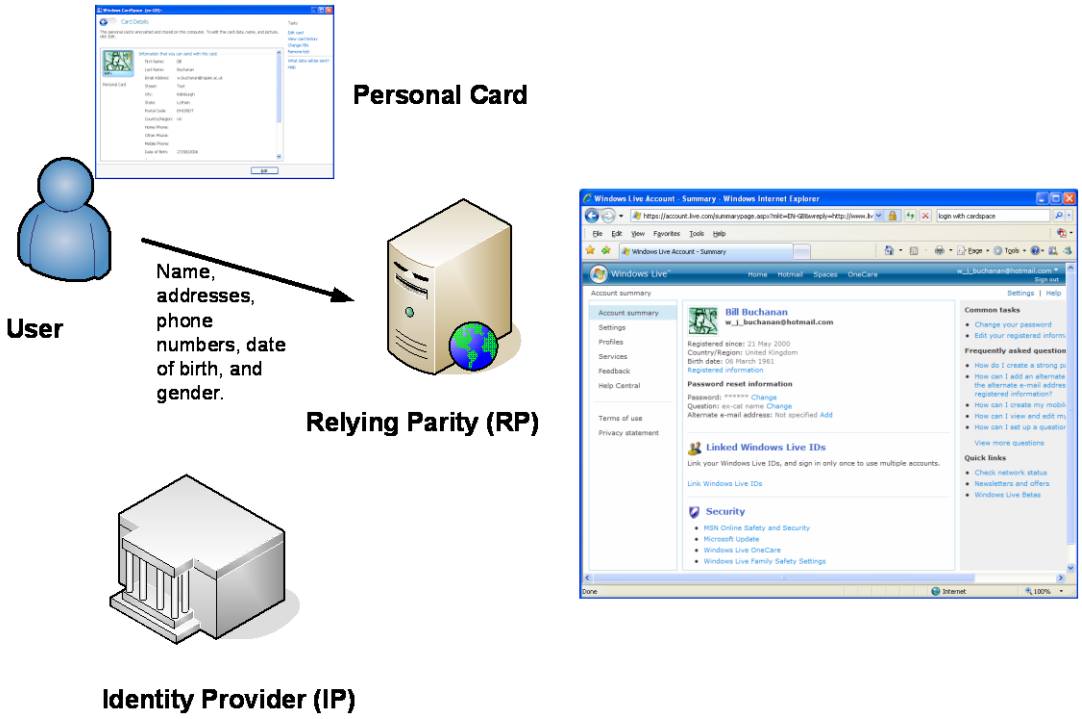


Figure 4.25 Personal cards

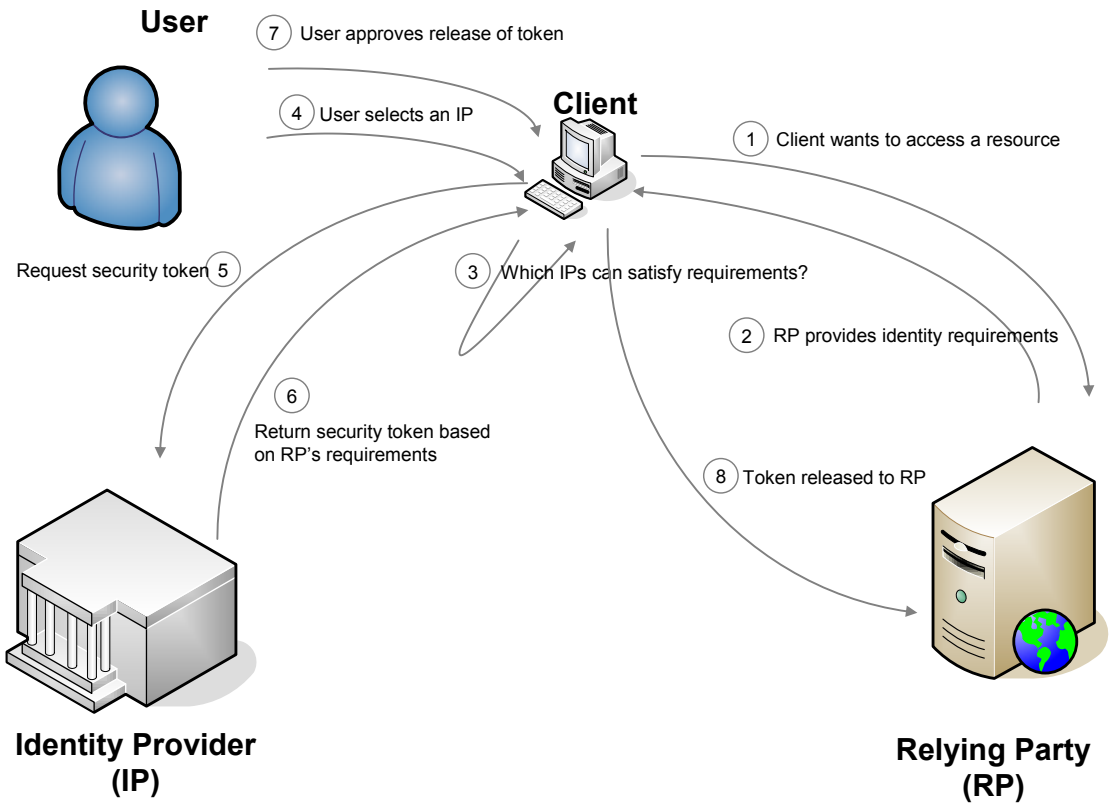


Figure 4.26 Managed cards

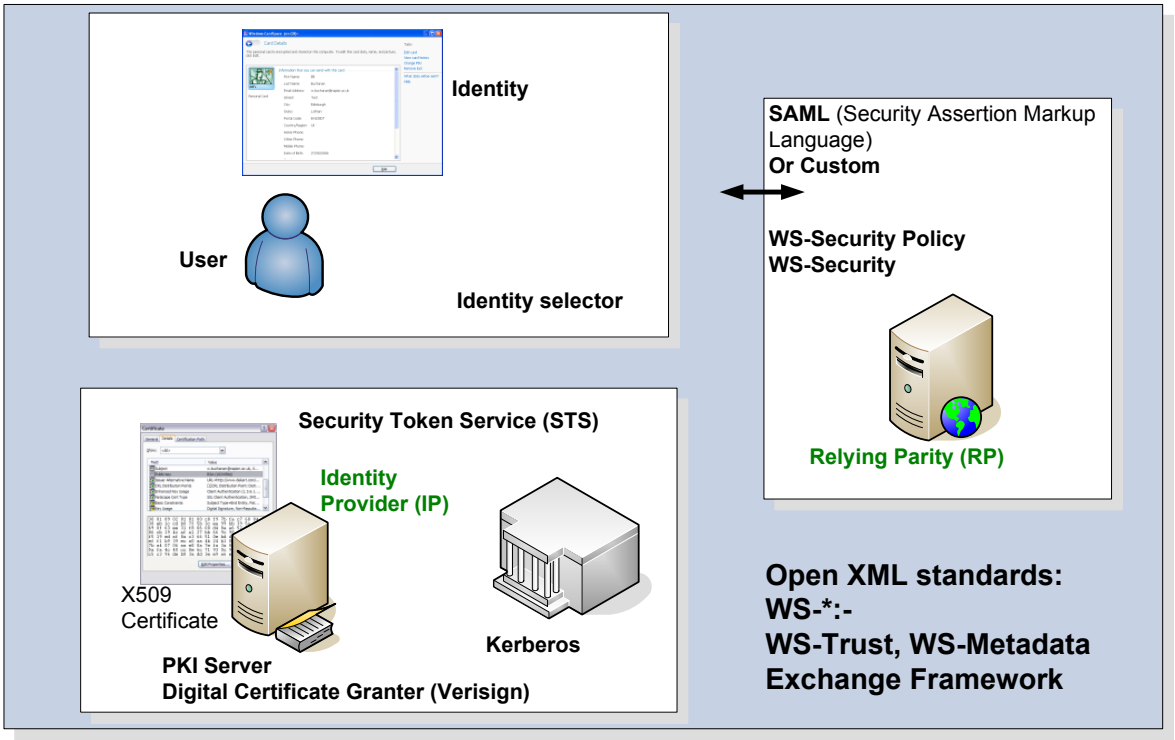


Figure 4.27 Standardized protocols

4.10 Email encryption

A popular type of email encryption is PGP (Pretty Good Privacy) which uses a public key to encrypt the data, and adds the private key of the user to provide authentication It can be seen, in Figure 4.28, that the first stage takes the text and produces an MD5 hash, which is encrypted, using RSA, with the user's private key. As the recipient has the user's public key, they should be able to decrypt it, and compare with the hash of the decrypted message. After a ZIP stage, the recipient's public key is then used to encrypt the output of the stage, which is then converted to ASCII characters using Base-64 (as required in standard email transmission). The recipient then uses their private key to decrypt the received message. After which they will determine its contents. Then they can use the sender's public key to decrypt the hashed value. This will then be compared with the hashed value from the message. If they are the same, then the message and the sender have been authenticated.

The true genius of PGP is the usage of unique key to encrypt the email message. The email is thus encrypted using IDEA and with a randomly generated key. Next the encryption key is encrypted with Alice's public key. At the receiver, all Alice has to do is to decrypt the IDEA key, and then decrypt it with it. The great advantage of this is that symmetric encryption/decryption is much faster and less process intensive than asymmetric methods. This is similar to someone locking up all the doors in a house, and the placing all the keys in a safe deposit box, that only one person holds the secret code for. Once the person has closed the door on the keys, even they cannot then get access to them, and only the person with the correct combination can get access to them. Each time we might create new keys, but the combination can stay the same.

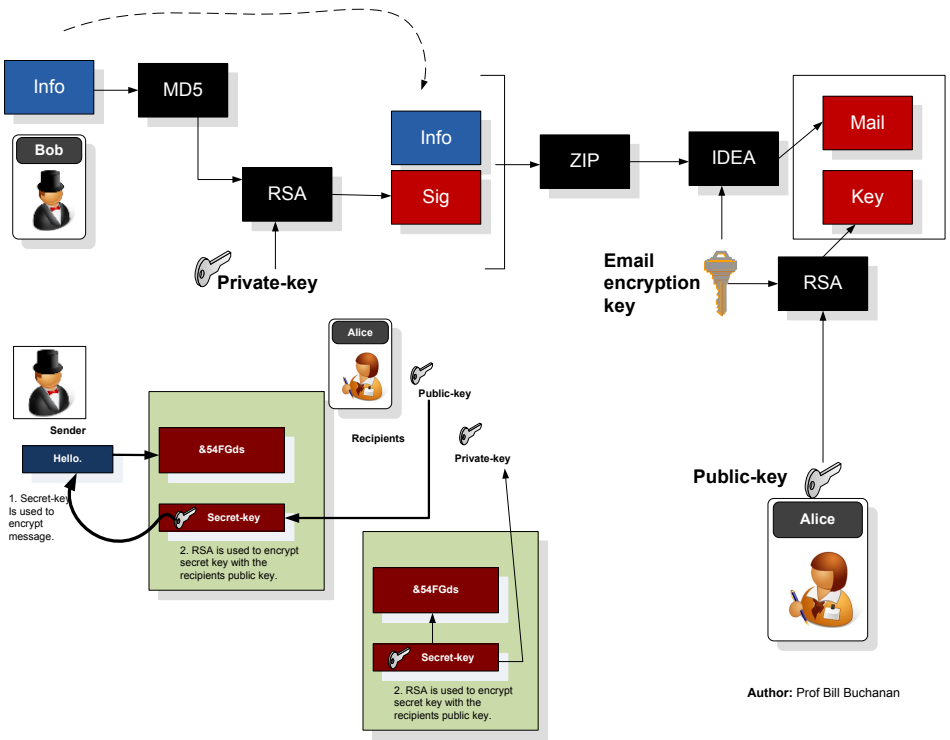


Figure 4.28 PGP

4.11 Tutorial

4.10.1 If Bob sends an email to Alice, which key does he use to keep the message secret:

(a) Bob's public key (b) Bob's private key
(c) Alice's public key (d) Alice's private key

4.10.2 If Bob sends an email to Alice, which key does he use to authenticate himself:

(a) Bob's public key (b) Bob's private key
(c) Alice's public key (d) Alice's private key

4.10.3 Which of the following is asymmetric encryption:

(a) RSA (b) DES
(c) AES (d) IDEA

4.10.4 Which of the following is symmetric encryption:

(a) RSA (b) DES
(c) AES (d) IDEA

4.10.4 Which of the following cannot be reversed with a decryption key:

(a) RSA (b) DES
(c) 3DES (d) MD5

4.10.5 Which of the following is an example of an MD5 hash signature:

(a) #54301
(b) d41d8cd98f00b204e9800998ecf8427e
(c) Sales-PC
(d) 00-ff-11-22-55-a1

4.10.6 Which of the following is not part of Bob's distributable digital certificate:

(a) Bob's public key
(b) Bob's private key
(c) The issuer
(d) Date of validity

4.10.9 The following is a digital certificate (http://buchananweb.co.uk/cert.zip). Download the file, and import it:

```
-----BEGIN CERTIFICATE-----
MIIDVzCCAwGgAwIBAgIKT39uTwAAAABHcDANBgkqhkiG9w0BAQUFADBgMQswCQYD
VQQGEwJHQjERMA8GA1UEChMIQXNjZXJ0aWExJjAkBgNVBAsTHUNsYXNzIDEgQ2Vy
dGlmaWNhdGUgQXV0aG9yaXR5MRYwFAYDVQQDEw1Bc2NlcnRpYSBDQSAxMB4XDTA3
MDExMTIwMzAyN1oXDTA4MDExMTIwNDAyN1owgZ8xJjAkBgkqhkiG9w0BCQEWF3cu
YnVjaGFuYW5AbmFwaWVyLmFjLnVrMQswCQYDVQQGEwJVSzEQMA4GA1UECBMHTG90
aGlhbjESMBAGA1UEBxMJRWRpbmJ1cmdoMRowGAYDVQQKExFOYXBpZXIgVW5pdmVy
c2l0eTELMAkGA1UECxMCSVQxGTAXBgNVBAMTEFdpbGxpYW0gQnVjaGFuYW4wgZ8w
DQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALB5YlMu1nZwqZO/C87/evlLhUXVw65U
BToYUJFpnp84caJZW8yzRpZ8iUgHfrPrO74dv+SecBu7qHlVfo8pMKe+a91i6AQ2
Zh9mffe0ndp9NzoHqt8dEn9hL8uq2bs8Oysn7h7ulRoE6TYOcSUDw08CYFKabfdg
0hrC9kyCG59hAgMBAAGjggEXMIIBEzAdBgNVHQ4EFgQUnWTXEQNjqwEJwWSHxnT/
W3SbhGkwYwYDVR0jBFwwWoAUlP5Zh0V700k6CorvRMWB9ifVkBmhP6Q9MDsxCzAJ
```

```
BgNVBAYTAkdCMREwDwYDVQQKEwhBc2NlcnRpYTEZMBcGA1UEAxMQQXNjZXJ0aWEg
Um9vdCBDQYIBDTBNBgNVHR8ERjBEMEKgQKA+hjxodHRwOi8vd3d3LmFzY2VydGlh
LmNvbS9PbmxpbmVDQS9jcmxzL0FzY2VydGlhQ0ExL2NsYXNzMS5jcmwwPgYIKwYB
BQUHAQEEMjAwMC4GCCsGAQUFBzAChiJodHRwOi8vb2NzcC5nbG9iYWx0cnVzdGZp
bmRlci5jb20vMA0GCSqGSIb3DQEBBQUAA0EAcB/Fg47QYK0u91aiG95mSKuCd9ND
mERu3MKKSIwy+Sx4LikwJEA0D2/8WcYL5LQ7q6y4tnRkQQBXQ1MvWFFGcw==
-----END CERTIFICATE-----
```

Determine the following:

Issued to:

Issued by:

Date of issue:

Signature algorithm:

Public-key type:

Subject:

4.10.9 Access the following e-commerce sites, and get to a place which produces an HTTPS connection (such as to purchase something or with a login). Determine the details of their certificates (the first one has already been completed):

Site	Issued to:	Issued by:	Date of issue/expiry	Signature algorithm:	Public key
amazon.co.uk	*amazon.co.uk*	*Verisign*	*23/01/08 to 22/01/09*	*SHA1RSA*	*RSA (1024 bits)*
amazon.com					
napier.ac.uk					
ebay.com					
maplin.co.uk					
paypal.com					

4.12 Software Tutorial

4.11.1 Implement a program for the MD5, SHA, SHA (256-bit), SHA (384-bit), SHA (512-bit) and complete the following table (with just the first five hex characters):

Text	**MD5**	**SHA**	**SHA (256)**	**SHA (384)**	**SHA (512)**
apple					
Apple					
apples					
This is it.					
This is it					

How many characters does each of the types have?

An outline of the code is:

```
using System;
using XCrypt; // Program uses XCrypt library from
//http://www.codeproject.com/csharp/xcrypt.asp
namespace Hash
{
   class Hash
   {
      static void Main(string[] args)
      {
         XCryptEngine xe = new XCryptEngine();
         xe.InitializeEngine(XCryptEngine.AlgorithmType.MD5);
         //      xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA);
         //      xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA256);
         //      xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA384);
         //      xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA512);
         Console.WriteLine("Enter string to hash:");
         string inText = Console.ReadLine();

         string hashText = xe.Encrypt(inText);

         Console.WriteLine("Input: {0}\r\nHash: {1}", inText,hashText);
         Console.ReadLine();
      }
   }
}
```

📖 **Web link:** http://buchananweb.co.uk/srcSecurity/tut5_1.zip

4.11.2 An alternative approach is to use the standard .NET classes for encryption. The following shows an example.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Security.Cryptography;

namespace ConsoleApplication1
{
    class Hash
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a message: ");
            string message= Console.ReadLine();
            System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
            MD5 md5 = new MD5CryptoServiceProvider();
            SHA1 sha1 = new SHA1CryptoServiceProvider();
            byte[] messageBytes = encoding.GetBytes(message);
            byte[] hashmessage = md5.ComputeHash(messageBytes);
            string stringMD5 = ByteToString(hashmessage);
            hashmessage = sha1.ComputeHash(messageBytes);
            string stringSHA1 = ByteToString(hashmessage);
            Console.WriteLine("MD5: {0}\r\nSHA-1: {1}", stringMD5, stringSHA1);
            Console.ReadLine();
        }
        public static string ByteToString(byte[] buff)
        {
            string sbinary = "";
            for (int i = 0; i < buff.Length; i++)
            {
                sbinary += buff[i].ToString("X2"); // hex format
            }
            return (sbinary);
```

```
        }
    }
}
```

This gives a sample run of:

```
Enter a message: hello
MD5: 5D41402ABC4B2A76B9719D911017C592
SHA-1: AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D
```

🕮 **Web link:** http://buchananweb.co.uk/hash.zip

Run the program, and prove its output. Next add 256-bit, 386-bit and 512-bit, using:

```
SHA256Managed sha256 = new SHA256Managed();
SHA384Managed sha384 = new SHA384Managed();
SHA512Managed sha512 = new SHA512Managed();
```

Thus show that that the SHA values for "hello" is:

SHA-256: 2CF24DBA5FB0A30E26E83B2AC5B9E29E1…2938B9824
SHA-384: 59E1748777448C69DE6B800D7A33BBFB9…DE828684F
SHA-512: 9B71D224BD62F3785D96D46AD3EA3D733…3BCDEC043

4.11.2 Export a certificate from your system, and update the program in Section 4.6 so that it displays the expiration date, the format, and the hash code, such as:

```
Serial Number: C0DD5E19983C6F575EFE454E7E66AD02
Effective Date: 08/11/1994 16:00:00
Expiration Date: 07/01/2010 15:59:59
Name: C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authoriy
Public key:
308185027E0092CE7AC1AE833E5AAA898357AC2501760CADAE8E2C37CEEB357864503E5844051C9BF8
F08E28A8208D216863755E9B12102AD7668819A05A24BC94B256622566C88078F781596D8407657013
71763E9B774CE35089569848B91DA7291A132E4A11599C1E15D549542C7336982B197399C6D706748E
5DD2DD6C81E7B0203010001
Public key algorithm: 1.2.840.113549.1.1.1
Issuer: C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authoity
Hash code: 1147389233
Format: X509
```

🕮 **Web link:** http://buchananweb.co.uk/srcSecurity/tut5_2.zip

4.11.3 For SHA1 HMAC, in Section 5.7, prove that the HMAC signature for a key of "fred" and a key of "apple" is:

bfca635df0a2faf671d14120a56010a543384818

4.11.4 Modify the program in Section 5.7 so that it accepts the message and key from the command prompt, and shows the resultant HMAC SHA-1 code, such as:

Enter message: **This is a message**

Enter key: **fred**
HMAC-SHA-1 signature: `19DCA8DA4499F49A8E1940FF7A6A937281369DBC`

Thus show that the key of "fred" produces a different output that "Fred.

4.11.5 The following are an HMAC MD5 and an HMAC SHA-1 signature. Show that the MD5 signature has 128 bits, and that the SHA one has 160 bits.

HMAC MD5 signature: `7c187710d7cd3c73c0135b1d34617d46`
HMAC-SHA-1 signature: `bfca635df0a2faf671d14120a56010a543384818`

4.11.6 A message of (ignore the inverted commas as this is not part of the message):

```
"This is the end of the world, do not panic!"
```

was sent and the HMAC result was:

`7BF135C0B795DB32E7E8533012E831C32C058871`

Which is the following is the secret key (use your own code or use http://buchanaweb.co.uk/hmac.aspx):

A `bert`
B `berty`
C `fred`
D `freddy`

4.11.7 The code in the following has only HMAC-MD5 and HMAC-SHA1. Update it with HMAC-SHA256 (`HMACSHA256`), HMAC-SHA-384 (`HMACSHA384`), HMAC-SHA-512 (`HMACSHA512`), and HMACRIPEMD160 (`HMACRIPEMD160`):

http:// buchananweb.co.uk/hmac2.zip

4.13 On-line Exercises

The on-line exercise for this chapter are at:

http://buchananweb.co.uk/auth.html

4.14 Web Page Exercises

Implement following Web pages using Visual Studio:

4.13.1 http://buchananweb.co.uk/security03.aspx [MD5/SHA-1]
4.13.2 http://buchananweb.co.uk/security03a.aspx [MD5/SHA-1 to Base-64]
4.13.3 http://buchananweb.co.uk/security03b.aspx [MD5/SHA-1 with salt]

4.13.3 http://buchananweb.co.uk/security01.aspx [HMAC]

4.15 NetworkSims exercises

Complete:

Complete: PIX_SNPA Challenge I11-30

4.16 Chapter Lecture

View the lecture at:

http://buchananweb.co.uk/security00.aspx

and select **Principles to Authentication [Link]**.

4.17 Reference

[1] This code is based around the Xcrypt libraries provided at http://www.codeproject.com/csharp/xcrypt.asp.