Security

Introduction

THIS DOCUMENT COVERS

- Introduction

Cryptography Overview

Cryptographic algorithms are either symmetric or public key (asymmetric). With symmetric key cryptography the same key is used for encryption and decryption. With public key cryptography there are two keys: the public key and the private key. One key is used for encryption and the other is used for decryption. In some algorithms, such as RSA, either the public or private key can be used for encryption. We can use cryptography to carry out the following tasks.

- Secure Communication
- Digitally Sign Messages
- Authentication

Secure Communication

We can use cryptography to ensure the communication between two counterparties remains secure.

Public Key Cryptography

Alice wants to send a secure message to Bob. This can be achieved using public key cryptography (asymmetric cryptography) as follows.

1. Bob gives Alice his public key p^B
2. Alice Encrypt the message using the public key $c = E(m, p^B)$
3. Alice gives the encrypted message c to Bob.
4. Bob decrypts the message using his secret key $m = D(c\,, s^B)$

The means by which Alice gives her public key to Bob is important. Bob needs to be sure that the person giving him Alice's key is really Alice to ensure imposters cannot send him messages. We will come back to this point again later.

Hybrid

In practice, because public key cryptography is significantly slower than symmetric cryptography, public key cryptography is only used to exchange keys. Subsequent exchanges can then use symmetric cryptography. Such hybrid systems work as follows.

1. Bob sends Alice his public key p^B
2. Alice generates a random session key k and encrypts it using Bobs public key. $k' = E(k, p^B)$
3. Alice sends the encrypted key k' to Bob
4. Bob decrypts the session key using his private key $k = D(k', s^B)$
5. Both parties then communicate using symmetric cryptography and the session key.

Digitally Signatures and Authentication

Another use of cryptography is to enable a counterparty to digitally sign their messages such that recipients can be sure the messages came from them. This works as follows.

Alice takes the message m, that she wants to send, and adds some extra information x (such as her name)

$$m' = m + x$$

Now Alice takes the resulting document and encrypts it with her private key giving the following.

$$\sigma = E(m', s^A)$$

Alice sends a packet with both the encrypted and unencrypted versions to Bob.

$$(\sigma, m')$$

Bob uses Alice's public key to decrypt σ. If it matches m' then it must have been sent by Alice. Furthermore, it cannot have been tampered with.

This system has the following properties.

- **Authentic:** If Alice's public key decrypts a message it must have come from Alice
- **Unforgeable:** Only Alice knows her private key
- **Unalterable:** If the signed document is tampered with it cannot be decrypted with Alice's public key.

> **NOTE:**
>
> As in the previous section on secure communication Bob must be sure the public key he thinks is Alice's is really Alice's public key. The next section discusses how we can achieve this.

Chains of trust

In the previous section we noted that the authentication mechanism works only if Bob is sure the public key that he has does in fact belong to Alice and not to some imposter. One way they could achieve this is to meet in person to enable Alice to give Bob her key. This is not feasible in general practice.

Now consider what happens if rather than meet Alice, Bob instead meets Charlie and Charlie gives Bob his public key. Charlie can effectively vouch for Alice. Charlie encrypts Alice's public key using his private key. When Bob receives it he decrypts it with Charlies public key. At this stage he knows that Alice is who she says she is. This is how certificate chains work. Operating systems and browsers come with pre-installed lists of root certificates which can be used to verify any other certificates issues by those root certificates in a chain of trust.

Certificates

On windows, X509 digital certificates are used to.

- Authenticate servers.
- Authenticate clients.
- Encrypt messages.
- Digitally sign messages.

Each certificate has a set of fields and a public key. The fields contain information such as the subject to which the certificate was issued and the issuer. Each such field is a claim. The claims can be categorised into those that relating to identity and those relating to rights.

We now show how we can use PowerShell to create two certificate files: one with a public/private key pair protected by password and another with just the public key.

Creating Certificates (PowerShell)

This section is based on the link

https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/how-to-create-temporary-certificates-for-use-during-development

CREATE CERTIFICATE WITH PUBLIC/PRIVATE KEY IN CERTIFICATE STORE.

The following creates a self-signed certificate with a subject of **MySubject.**

```
$rootcert = New-SelfSignedCertificate -CertStoreLocation
Cert:\CurrentUser\My -DnsName "MySubject" -TextExtension
@("2.5.29.19={text}CA=true") -KeyUsage CertSign,CrlSign,DigitalSignature
```

CREATE LOOKUP PATH TO CERTIFICATE IN STORE.

The certificate we just created is stored in the certificate store (more later). The key in the store is the certificates thumbprint. This is a hash of the certificates contents that can be used to refer to it. We build a path to the certificate in the store.

```
[String]$rootCertPath = Join-Path -Path 'cert:\CurrentUser\My\' -ChildPath
"$($rootcert.Thumbprint)"
```

CREATE PRIVATE/PUBLIC KEY CERTIFICATE FILE

We export the certificate to a PFX file (with **`.pfx`** extension) Because this file contains the private key, we need to password protect it

```
[System.Security.SecureString]$rootcertPassword = ConvertTo-SecureString -
String "Passw0rd" -Force -AsPlainText

Export-PfxCertificate -Cert $rootCertPath -FilePath 'PubPrivateCert.pfx' -
Password $rootcertPassword
```

CREATE PUBLIC KEY CERTIFICATE FILE

Now we export a certificate with just the public key and call it

```
Export-Certificate -Cert $rootCertPath -FilePath 'Public.crt'
```

We now have two files in the folder `C:\Users\rps\WebSec`

```
Public.crt
'PubPrivateCert.pfx'
```

We will use these in the next sections.

Using Certificates for Authentication

The following code uses the certificates we created in the previous sections to carry out authentication in DotNet Core.

```
// Load the X509 certificate and extract is public and private keys.
X509Certificate2 certificate = new X509Certificate2(
    fileName: @"C:\Users\rps\WebSec\PubPrivateCert.pfx",
    password: "Passw0rd");

RSA rsaPublicKey = certificate.GetRSAPublicKey();
RSA rsaPrivateKey = certificate.GetRSAPrivateKey();

// Take a piece of data that we want to sign and convert it
// to a byte array
string message = "Hello World";
byte[] messageBytes = System.Text.Encoding.UTF8.GetBytes(message);

// Sign the message bytes using the private key
byte[] signedMessageBytes = rsaPrivateKey.SignData(
    messageBytes, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);

// Verify the message bytes using the signed bytes.
// and the public key
// As the messageBytes are untampered
bool result = rsaPublicKey.VerifyData(
    messageBytes,
    signedMessageBytes,
    HashAlgorithmName.SHA256,
    RSASignaturePadding.Pkcs1);

// Tamper with the bytes
messageBytes[0] = 45;

// Now the verification will fail
bool result2 = rsaPublicKey.VerifyData(
    messageBytes,
    signedMessageBytes,
    HashAlgorithmName.SHA256,
    RSASignaturePadding.Pkcs1);
```

Using Certificates for Encryption

The following example shows how to use the certificates we generated for encryption

```
// Load the X509 certificate and extract is public and private keys.
X509Certificate2 certificate = new X509Certificate2(
    fileName:@"C:\Users\rps\WebSec\PubPrivateCert.pfx",
    password:"Passw0rd");

RSA rsaPublicKey = certificate.GetRSAPublicKey();
RSA rsaPrivateKey = certificate.GetRSAPrivateKey();

// Take a piece of data that we want to sign and convert it
// to a byte array
string message = "Hello World";
byte[] messageBytes = System.Text.Encoding.UTF8.GetBytes(message);

// Encrupt the message bytes using the public key
byte[] encryptedBytes =
rsaPublicKey.Encrypt(messageBytes,RSAEncryptionPadding.Pkcs1);

// Try and look at the encrypted bytes and we get garbage.
string resBytes = System.Text.Encoding.UTF8.GetString(encryptedBytes);

// Now decrypt the encripted bytes using the private key
byte[] decryptedBytes =
rsaPrivateKey.Decrypt(encryptedBytes,RSAEncryptionPadding.Pkcs1);
string resBytes2 = System.Text.Encoding.UTF8.GetString(decryptedBytes);
```

Using Certificates And JWT Tokens

```
// Load the X509 certificate that contains both the public and private
keys,
// protected by password
X509Certificate2 certificate = new X509Certificate2(
    fileName:@"C:\Users\rps\WebSec\PubPrivateCert.pfx",
    password:"Passw0rd");

// Load the certificate that only has a public key
X509Certificate2 publicCertificate = new X509Certificate2(
fileName: @"C:\Users\rps\WebSec\Public.crt");


// Create a key from the public/private certificate
// and use it to create some signing credentials
X509SecurityKey privateKey = new X509SecurityKey(certificate);
SigningCredentials signingCredentials =
    new SigningCredentials(privateKey,SecurityAlgorithms.RsaSha256);

// Create a set of claims.
var claims = new List<Claim>()
{
    new Claim("application", "one"),
    new Claim("application", "two")
};

// Create a token that signs the claims
JwtSecurityToken token = new JwtSecurityToken(
    issuer: "RandomIssuer",
    audience: "RandomAudience",
    expires: DateTime.Now.AddDays(1),
    signingCredentials: signingCredentials,
    claims: claims);

// Create a token handler and use it to create a token string
    var securityTokenHandler = new JwtSecurityTokenHandler();
string tokenString =securityTokenHandler.WriteToken(token);
tokenString.Dump();

// Two validate the token we just need the public certificate
var validationKey = new X509SecurityKey(publicCertificate);
```

```
// The validation parameters
var tokenValidationParameters = new TokenValidationParameters
{
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidAudience = "RandomAudience",
    ValidIssuer = "RandomIssuer",
    IssuerSigningKey = validationKey,
    ValidateLifetime = true,
    ClockSkew = TimeSpan.FromSeconds(30)
};

// Validate the claim and get back the claims
var claimsPrinciple = securityTokenHandler.ValidateToken(
    tokenString,
    tokenValidationParameters,
    out SecurityToken outToken);

outToken.Dump();

// Tamper with the token throws exception
tokenString = tokenString + "more";
var claimsPrinciple2 = securityTokenHandler.ValidateToken(
    tokenString,
    tokenValidationParameters,
    out SecurityToken outToken2);
```

X509 Certificates

Overview

Most certificates these days are X509 v3 certificates. The PKIX variants of these are the ones used by browsers for HTTPS and TLS between clients and servers. X509 is an abstract format and needs an encoding to describe the binary layout of the certificate. The most common encoding is DER (Distinguished Encoding Rule).

A standard DER encoded certificate is binary and has the `.der` extension. Because binary is tricky to read, certificates can be packaged as PEM (Privacy Enhanced Email) files. PEM has a header and footer with a base 64 encoding in between. Such files will have a `.pem` extension.

x509 files can take an envelope format. An envelope takes one or more certificates and other stuff. Extensions are `.p7b, .p7c` for java and `.pfx, .p12` for Microsoft. Typically, these are encoded as raw DER.

Certificate Stores

Certificates are contained in stores. At the highest level there are two stores.

- **Local Machine:** Certificates accessed by machine processes such as ASP.NET.
- **Current User:** Certificates that authenticate a user to a service.

As a rule of thumb if a service is hosted as a windows service it will use the local machine store. If the service or client runs under a user account, then use the current user store.

Each of these can is divided into the following two sub-stores.

- **Trusted Root Certification Authorities:** Used to create a chain of certificates, which can be traced back to a certification authority. The local computer implicitly trusts any certificate placed in this store.
- **Personal: "**This store is used for certificates associated with a user of a computer. Typically, this store is used for certificates issued by one of the certification authority certificates found in the Trusted Root Certification Authorities store. Alternatively, a certificate found here may be self-issued and trusted by an application"

Certificate Chains

> *"Certificates are created in a hierarchy where each individual certificate is linked to the CA that issued the certificate. This link is to the CA's certificate. The CA's certificate then links to the CA that issued the original CA's certificate. This process is repeated up until the Root CA's certificate is reached. The Root CA's certificate is inherently trusted."*

Service and Client Certificates

Service certificates primarily authenticate the server to clients. A client will check the value of the subject field against the URI used to contact the service. The intended purpose must have the correct value such as "Server Authentication".

Hashing and Digital Signatures

Public Key algorithms are inefficient for large documents. To save time, one-way hash functions are used. Alice signs a hash of the document, rather than the document itself.

1. Alice hashes the document $h = H(m)$
2. Alice encrypts the hash using her private key $h' = E(h, k_i^a)$
3. Alice sends the encrypted hash and the document to Bob
4. Bob decrypts the hash using Alice's public key $h = E(h', k_u^a)$
5. Bob hashes the document m and compares it with h

All digital signature algorithms are public key based. They use secret information to sign documents and public information to verify the signatures. We denote the signing process with private key K as

$$S(m, k_i)$$

And verifying with public key as

$$V(m, k_u)$$

The bit string that we add to the document when signed is called the digital signature The whole process that convinces the receiver of the identity of the sender is known as authentication.

Digital Signatures and Encryption

We can combine public key cryptography and digital signatures to carry out authentication and privacy.

1. Alice signs the message using her private key $S(m, k_i^a)$
2. Alice encrypts the signed message using Bobs public key $E(S(m, k_i^a), k_u^b)$
3. Bob decrypts the encrypted signed message using his private key $D\left(E\left(S(m, k_i^a), k_u^b\right), k_i^b\right) = S(m, k_i^a)$
4. Bob verifies with Alice's public key $V(S(m, k_i^a), k_u^a) = m$

Protocols

Key Exchange

Assume we have a Key Distribution Center. Both Alice and Bob have a private key on the KDC and want to communicate

1. Alice tells KDC she wants to communicate with Bob.
2. KDC generates a session key sends two copies to Alice; one encrypted with Alice's private key and the other encrypted with Bobs private key
3. Alice decrypts the session key encrypted with her own private key and sends the one encrypted with bobs private key to Bob
4. Bob decrypts the session key encrypted with his private key
5. This communicate using the session key

One Way Function

A function with the property that while it is easy to calculate $f(x)$ given x, it is impossible to calculate x given $f(x)$