Protezione .NET
Supporto dei certificati nelle applicazioni con .NET Framework 2.0
Dominick Baier

In questo articolo verranno discussi i seguenti argomenti:	In questo articolo verranno utilizzate le seguenti tecnologie:
• L'archivio certificati di Windows • Classi di certificati in .NET • Convalida, SSL, servizi Web e firma del codice • Firmare e crittografare i dati	Framework 2.0

Sommario

I certificati sono molto utilizzati in Microsoft® .NET Framework, ad esempio per la protezione delle comunicazioni, per la firma del codice e per i criteri di protezione. .NET Framework 2.0 ha introdotto un supporto rinnovato per i certificati e ha aggiunto uno spazio dei nomi completamente nuovo per le operazioni di crittografia con certificati, compatibili con gli standard. In questo articolo sanno discussi il background dei certificati e l'archivio certificati di Windows®. Verrà illustrato come lavorare con le API di certificato e come queste ultime sono utilizzate da Framework per implementare le funzionalità di protezione.
Un "certificato" è un file codificato ASN.1 (Abstract Syntax Notation One) che contiene una chiave pubblica e altre informazioni relative alla chiave e al proprietario. Inoltre, un certificato ha un periodo di validità ed è firmato con un'altra chiave (l'autorità di certificazione) che è utilizzata per fornire una garanzia di autenticità di quegli attributi e, cosa ancora più importante, della chiave pubblica stessa. È possibile pensare ad ASN.1 come a una specie di XML binario. Come XML, ha regole di codifica, tipi forti e tag; in genere sono valori binari che spesso non corrispondono a nessun carattere stampabile.
Per far sì che un tale file sia intercambiabile tra i sistemi, è necessario un formato standard. Questo è X.509 (attualmente alla versione 3), che è descritto in RFC 3280 (tools.ietf.org/html/rfc3280). X.509 non determina il tipo di chiave incorporata nel certificato, ma l'algoritmo RSA è attualmente l'algoritmo crittografico asimmetrico più comunemente utilizzato.
Iniziamo dalla storia dell'algoritmo. Il nome RSA è l'acronimo dei cognomi dei tre inventori di quest'algoritmo: Ron Rivest, Adi Shamir e Len Adleman. I tre hanno fondato un'azienda, RSA Security, che ha pubblicato diversi documenti sugli standard Public Key Cryptography Standards (PKCS), i quali descrivono diversi aspetti della crittografia.
Uno dei più noti di questi documenti, PKCS #7, definisce un formato binario per i dati firmati e crittografati, che viene detto Cryptographic Message Syntax (CMS). Attualmente CMS è utilizzato in molti protocolli di protezione assai diffusi, tra cui Secure Sockets Layer (SSL) e Secure Multipurpose Internet Mail Extensions (S/MIME). Poiché è uno standard, è anche il formato comunemente usato quando c'è la necessità che le applicazioni scambino dati firmati e crittografati tra parti diverse. I documenti di PKCS sono disponibili sul sito Web di RSA Laboratories (www.rsasecurity.com/rsalabs/node.asp?id=2124).

Come ottenere un certificato

Ci sono diversi modi acquisire un certificato. Quando i file vengono scambiati, i certificati di solito appaiono in uno di questi due formati. I file con estensione cer sono file firmati ASN.1 nel formato X.509v3, contengono una chiave pubblica e le informazioni aggiuntive citate in precedenza. Sono queste le informazioni comunicate ai partner commerciali o agli amici in modo che sia loro possibile utilizzare la chiave pubblica per codificare i dati destinati all'utente.
Esistono anche file con estensione pfx (Personal Information Exchange). Un file pfx contiene un certificato e la corrispondente chiave privata (il formato è descritto nello standard PKCS #12). Tali file sono estremamente riservati e sono generalmente utilizzati per importare coppie di chiavi su un server o per scopi di backup. Quando si esportano coppie di chiavi, Windows permette di crittografare il file pfx con una password; sarà necessario fornire nuovamente la password al momento di importare la coppia di chiavi.

È possibile anche generare i propri certificati, con modalità che dipendono di solito dall'utilizzo che se ne farà. Per i normali scenari Internet, quando non si conoscono gli interlocutori, tipicamente si richiede un certificato da un'autorità di certificazione commerciale (CA). Quest'approccio ha il vantaggio che queste autorità di certificazione sono note e ritenute attendibili da Windows e da qualunque altro sistema operativo (e browser) che supporta i certificati e SSL. Di conseguenza, non è necessario uno scambio di chiave CA.
Per gli scenari B2B Intranet, è possibile utilizzare una autorità di certificazione interna. I Servizi di certificazione sono inclusi in Windows 2000 e Windows Server® 2003. Abbinata a Active Directory®, questa funzionalità permette di distribuire facilmente i certificati in una organizzazione. Di seguito si descrive come richiedere i certificati da una autorità di certificazione privata.
A volte durante lo sviluppo potrebbe capitare una situazione in cui gli approcci descritti in precedenza non siano applicabili. Ad esempio, se si ha bisogno di un certificato rapidamente per esigenze di test, è possibile utilizzare makecert.exe. Contenuto in .NET Framework SDK, questo strumento genera certificati e coppie di chiavi. Esiste anche uno strumento simile, chiamato selfssl.exe, in IIS Resource Kit; è specializzato per creare coppie di chiavi SSL e può anche configurare IIS con tale coppia di chiavi in un solo passaggio.

Archivio certificati di Windows
I certificati ed le loro corrispondenti chiavi private possono essere memorizzate su una varietà di periferiche, come dischi rigidi, smart card e chiavi USB. Windows fornisce un livello astrazione, chiamato archivio certificati, per unificare l'accesso ai certificati indipendentemente da dove sono memorizzati. Se la periferica hardware ha un provider del servizio di crittografia (CSP) supportato da Windows, è possibile accedere ai dati memorizzati su di essa utilizzando l'API Archivio certificati.
L'archivio certificati è nascosto in profondità nel profilo utente. Questo consente l'uso delle ACL sulle chiavi per un account specifico. Ogni archivio è suddiviso in contenitori. Ad esempio, c'è un contenitore chiamato Personale dove si memorizzano i certificati personali (quelli che hanno una chiave privata associata). Il contenitore Autorità di certificazione principale attendibile contiene tutti i certificati delle autorità di certificazione considerate attendibili. Il contenitore Altri contatti contiene i certificati delle persone con cui si comunica in modo protetto, e così via. Il modo più semplice per accedere all'archivio certificati è eseguire certmgr.msc.
Esiste anche un archivio a livello di computer, che è utilizzato dagli account di computer Windows (NETWORK, LOCAL SERVICE e LOCAL SYSTEM) o se si desidera condividere i certificati o le chiavi tra gli account. Le applicazioni ASP.NET utilizzano sempre l'archivio a livello di computer; per le applicazioni desktop, si installano generalmente certificati nell'archivio utente.
Soltanto gli amministratori possono gestire gli archivi di computer e di servizio account. A questo scopo, occorre avviare Microsoft Management Console (mmc.exe) e aggiungere lo snap-in Certificati, che consente di scegliere l'archivio da gestire. **Nella figura 1** viene mostrata una schermata dello snap-in di MMC.

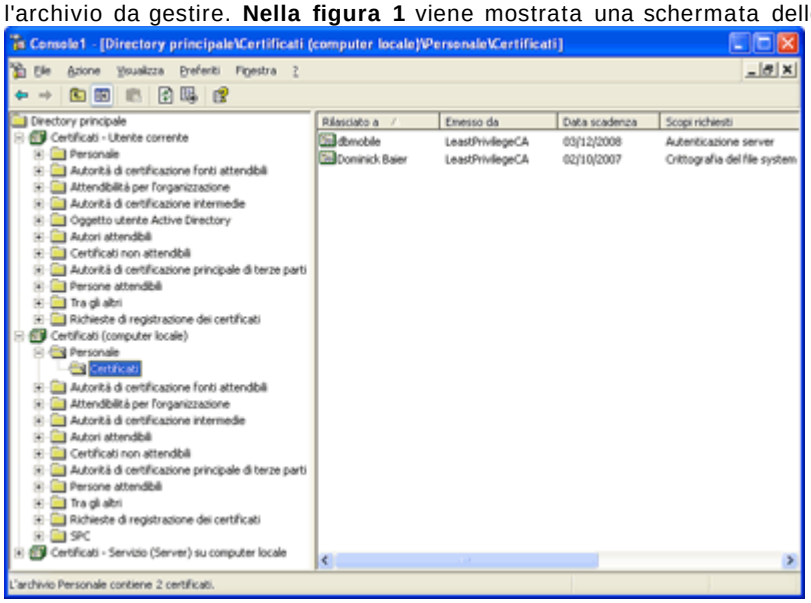

Figura 1 **Snap-in MMC Certificati** (Fare clic sull'immagine per ingrandirla)
Oltre a consentire di importare, esportare e cercare i certificati, lo snap-in permette di richiedere i certificati da una autorità di certificazione interna dell'organizzazione. Basta fare clic col pulsante destro del mouse sul contenitore personale e scegliere Tutte le attività | Richiedi certificato. Il computer locale genera poi una coppia di chiavi di RSA e invia la parte di chiave alla autorità di certificazione per la firma. Windows aggiunge il certificato firmato all'archivio certificati e la corrispondente chiave privata in un contenitore chiavi. Il certificato viene collegato al contenitore chiavi tramite un attributo di archiviazione.
L'accesso ai contenitori di chiavi private è rigidamente regolato da ACL e limitato all'account corrispondente o al sistema locale. Questo è un problema quando si desidera accedere a chiavi contenute nel profilo del computer da ASP.NET o dall'account di altri utenti. Uno strumento da utilizzare per modificare le ACL del file contenitore è disponibile all'indirizzo www.leastprivilege.com/HowToGetToThePrivateKeyFileFromACertificate.aspx .
Le autorità di certificazione commerciali e di Windows dispongono anche di interfacce Web per richiedere certificati. In questi casi, un controllo ActiveX® in Internet Explorer ® genera le chiavi e le importa nell'archivio dell'utente corrente. Come regola generale, per rendere un certificato accessibile a un utente o un servizio,

esistono due possibilità: importarlo nel proprio archivio oppure richiederlo dopo avere eseguito l'accesso come quell'utente.

Utilizzo dei certificati
I certificati sono utilizzati ampiamente in .NET Framework. A un certo livello, tutta la funzionalità si basa sulla classe X509Certificate dello spazio dei nomi System.Security.X509Certificates. Dopo un esame attento, si trova anche una classe di certificati che finisce con un 2. Ciò dipende dal fatto che .NET Framework 1.*x* ha ricevuto una rappresentazione dei certificati X.509 denominata X509Certificate. Questa classe aveva funzionalità limitate e nessun supporto per le operazioni di crittografia. Nella versione 2.0, è stata aggiunta una nuova classe denominata X509Certificate2, la quale deriva da X509Certificate e aggiunge molte funzionalità. È possibile scegliere tra le due versioni secondo necessità, ma è preferibile utilizzare la versione più recente.

Accedere ai certificati
È possibile recuperare i certificati direttamente dal file system, nonostante sia meglio farlo dall'archivio certificati. Per creare un'istanza X509Certificate2 da un file cer, basta passare il nome file al costruttore:

Copia codice

```
X509Certificate2 cert1 = new X509Certificate2("alice.cer");
```

È anche possibile caricare certificati dai file pfx. Come si è visto in precedenza, i file pfx possono essere protetti da password e la password deve essere fornita come SecureString. SecureString crittografa la password internamente e tenta di minimizzarne l'esposizione nella memoria, nei file di paging e nei dettagli di arresto anomalo. Per questa ragione, è possibile aggiungere solo un carattere (tipo valore) alla volta alla stringa. Il codice nella **figura 2**, che disattiva l'eco console e restituisce una SecureString, è utile se si vuole richiedere agli utenti una password dalla console.

Figure 2 Richiedere una password alla console

Copia codice

```
private SecureString GetSecureStringFromConsole()
{
    SecureString password = new SecureString();

    Console.Write("Enter Password: ");
    while (true)
    {
        ConsoleKeyInfo cki = Console.ReadKey(true);

        if (cki.Key == ConsoleKey.Enter) break;
        else if (cki.Key == ConsoleKey.Escape)
        {
            password.Dispose();
            return null;
        }
        else if (cki.Key == ConsoleKey.Backspace)
        {
            if (password.Length != 0)
                password.RemoveAt(password.Length - 1);
        }
        else password.AppendChar(cki.KeyChar);
    }

    return password;
}
```

Nell'articolo "Credential Management with the .NET Framework 2.0" (disponibile in inglese all'indirizzo msdn.microsoft.com/library/en-us/dnnetsec/html/credmgmt.asp), Kenny Kerr ha incluso il codice per convertire il risultato del normale dialogo di credenziali Windows in una SecureString. Indipendentemente da come è stata ottenuta, la SecureString può essere passata poi al costruttore X509Certificate2 per caricare il file pfx, nel modo seguente:

Copia codice

```
X509Certificate2 cert2 = new X509Certificate2("alice.pfx", password);
```

Per accedere all'archivio certificati di Windows, si utilizza la classe X509Store. Nel costruttore si fornisce la posizione dell'archivio (l'utente corrente o il computer) e il nome archivio. Per specificare il contenitore che si vuole aprire, è possibile utilizzare sia una stringa sia l'enumerazione StoreName. Tenere conto che i nomi interni non sempre corrispondono a quelli che si trovano nello snap-in MMC. Il contenitore Personale viene mappato al nome My, mentre Altri contatti diventano AddressBook.
Una volta che si dispone di una istanza X509Store valida, è possibile cercare, recuperare, eliminare e aggiungere certificati. Con l'eccezione degli scenari di distribuzione, probabilmente si utilizzerà molto più spesso la funzionalità di ricerca. È possibile cercare certificati in base a svariati criteri, compreso il nome del soggetto, il numero di serie, l'identificazione personale, l'autorità di certificazione e il periodo di validità. Se si recuperano regolarmente certificati nelle proprie applicazioni dall'archivio, si dovrebbe utilizzare una proprietà unica, ad esempio l'identificatore chiave del soggetto. Anche l'identificazione personale è unica, ma occorre tener presente che questo è un valore hash SHA-1 del certificato e cambia se, per esempio, il certificato viene rinnovato. Il codice nella **figura 3** mostra un modo generico di cercare i certificati.

Figure 3 Cercare i certificati

Copia codice

```
static void Main(string[] args)
{
    // search for the subject key id
    X509Certificate2 cert = FindCertificate(
      StoreLocation.CurrentUser, StoreName.My,
      X509FindType.FindBySubjectKeyIdentifier,
      "21f2bf447298e83056a69eb02ebe9085ed97f10a");
}

static X509Certificate2 FindCertificate(
    StoreLocation location, StoreName name,
    X509FindType findType, string findValue)
{
    X509Store store = new X509Store(name, location);
    try
    {
        // create and open store for read-only access
        store.Open(OpenFlags.ReadOnly);

        // search store
        X509Certificate2Collection col = store.Certificates.Find(
          findType, findValue, true);

        // return first certificate found
        return col[0];
```

```
    }
        // always close the store
    finally { store.Close(); }
}
```

Quando si dispone di un'istanza X509 Certificate2, è possibile esaminare le varie proprietà del certificato (come il nome del soggetto, le date di scadenza, l'autorità di certificazione e il nome descrittivo). La proprietà HasPrivateKey fa capire se esiste una chiave privata associata. Le proprietà PrivateKey e PublicKey restituiscono le corrispondenti chiavi come istanza RSACryptoServiceProvider.
Per importare un certificato, si chiama il metodo Add nell'istanza X509Store. Quando si specifica un nome archivio che non esiste nel costruttore dell'archivio, viene creato un nuovo contenitore. Ecco come importare un certificato da un file chiamato alice.cer in un nuovo contenitore chiamato Test:

Copia codice

```
static void ImportCert()
{
    X509Certificate2 cert = new X509Certificate2("alice.cer");
    X509Store store = new X509Store("Test", StoreLocation.CurrentUser);
    try
    {
        store.Open(OpenFlags.ReadWrite);
        store.Add(cert);
    }
    finally { store.Close(); }
}
```

Visualizzare i dettagli di certificato e la selezione certificato
Windows offre due finestre di dialogo standard per lavorare con i certificati: una visualizza i dettagli del certificato (proprietà e percorso di certificazione) e una consente agli utenti di scegliere un certificato da un elenco. È possibile accedere a queste finestre di dialogo dai due metodi statici della classe X509Certificate2UI: SelectFromCollection e DisplayCertificate.
Per mostrare un elenco di certificati occorre popolare un X509Certificate2Collection e trasferirlo a SelectFromCollection. È molto comune permettere a un utente di scegliere uno dei certificati personali in archivio. A tale scopo, basta passare alla proprietà Certificates di un archivio X509Store aperto. È possibile controllare anche il titolo della finestra di dialogo, un messaggio e se le selezioni multiple sono consentite. Il metodo DisplayCertificate mostra lo stesso dialogo che si vede quando si fa doppio clic su un file cer in Esplora risorse. Nella **figura 4** viene mostrata la finestra di dialogo utilizzata per scegliere un certificato e nella **figura 5** si fornisce il codice corrispondente.
Figure 5 Codice per scegliere un certificato

Copia codice

```
private static X509Certificate2 PickCertificate(
  StoreLocation location, StoreName name)
{
    X509Store store = new X509Store(name, location);
    try
    {
        store.Open(OpenFlags.ReadOnly);

        // pick a certificate from the store
        X509Certificate2 cert =
```

```
            X509Certificate2UI.SelectFromCollection(
                store.Certificates, "Caption",
                "Message", X509SelectionFlag.SingleSelection)[0];

        // show certificate details dialog
        X509Certificate2UI.DisplayCertificate(cert);
        return cert;
    }
    finally { store.Close(); }
}
```

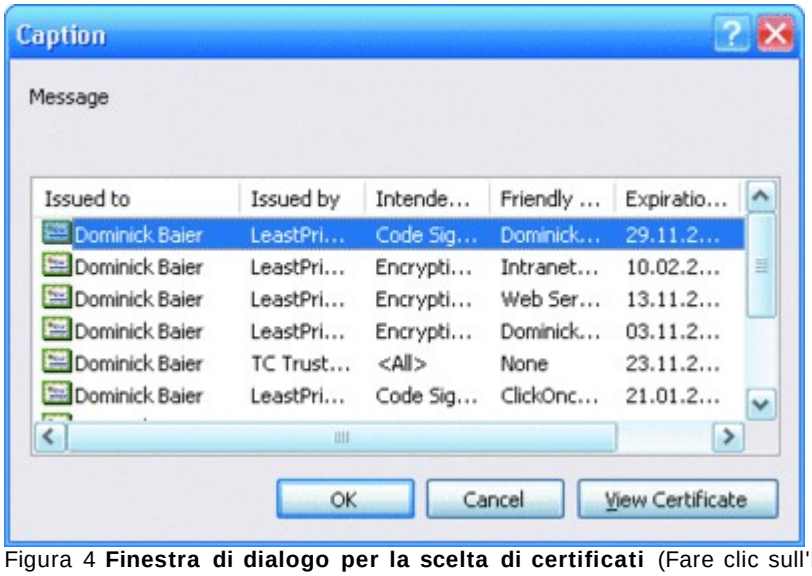

Figura 4 **Finestra di dialogo per la scelta di certificati** (Fare clic sull'immagine per ingrandirla)

Convalida dei certificati
Ci sono alcuni criteri da considerare quando si convalida un certificato, soprattutto l'autorità di certificazione (generalmente, si considera attendibile solo il certificato emesso da una autorità di certificazione disponibile nell'elenco locale) e la validità attuale (i certificati possono cessare di essere validi quando scadono o sono revocati dall'autorità di certificazione di emissione). È possibile utilizzare la classe X509Chain per verificare queste proprietà. Utilizzando questa classe, è possibile specificare un criterio per la verifica della validità. Ad esempio, è possibile richiedere un'autorità di certificazione radice disponibile nell'elenco locale o specificare di controllare gli elenchi di revoche locali o online. Per controllare i certificati che sono stati utilizzati per firmare i dati, è importante controllare se il certificato era valido al momento della firma. A tale scopo, X509Chain permette di modificare il momento di verifica.
Dopo aver definito i criteri, chiamare il metodo Build per ottenere le informazioni sul risultato della convalida della proprietà ChainStatus. Se ci sono errori di convalida multipli, è possibile iterare la procedura sull'insieme ChainElement per avere più dettagli. Nella **figura 6** viene mostrato come eseguire una convalida rigorosa di un certificato e dell'autorità di certificazione confrontando gli elenchi delle revoche, fuori linea e in linea.
⊞ Figure 6 Convalida ristretta del certificato

Copia codice

```
static void ValidateCert(X509Certificate2 cert)
{
    X509Chain chain = new X509Chain();

    // check entire chain for revocation
    chain.ChainPolicy.RevocationFlag = X509RevocationFlag.EntireChain;
```

```
    // check online and offline revocation lists
    chain.ChainPolicy.RevocationMode =
        X509RevocationMode.Online | X509RevocationMode.Offline;

    // timeout for online revocation list
    chain.ChainPolicy.UrlRetrievalTimeout = new TimeSpan(0, 0, 30);

    // no exceptions, check all properties
    chain.ChainPolicy.VerificationFlags = X509VerificationFlags.NoFlag;

    // modify time of verification
    //chain.ChainPolicy.VerificationTime = new DateTime(1999, 1, 1);

    chain.Build(cert);

    if (chain.ChainStatus.Length != 0)
        Console.WriteLine(chain.ChainStatus[0].Status);
}
```

Supporto SSL
Il protocollo di autenticazione SSL si basa sui certificati. Il supporto per SSL in .NET Framework consiste di due parti. Il caso speciale (ma più utilizzato) di SSL su HTTP è implementato dalla classe HttpWebRequest (utilizzata anche per i client proxy del servizio Web). Per abilitare SSL, non si deve fare nulla di speciale tranne specificare un URL che utilizza il protocollo https .
Quando ci si connette a un endpoint protetto SSL il certificato server è convalidato sul client. Se la convalida non riesce, per impostazione predefinita la connessione viene immediatamente chiusa. È possibile modificare questa impostazione fornendo una richiamata a una classe denominata ServicePointManager. Quando lo stack client HTTP esegue la convalida del certificato, dapprima controlla se è stata fornita una richiamata e, nel caso, esegue il relativo codice. Per associare la richiamata, occorre fornire un delegato del tipo RemoteCertificateValidationCallback:

Copia codice

```
// override default certificate policy
// (for example, for testing purposes)
ServicePointManager.ServerCertificateValidationCallback =
    new RemoteCertificateValidationCallback(VerifyServerCertificate);
```

Nella richiamata, si ottiene il certificato server, un codice di errore e il passaggio di un oggetto di catena. È possibile quindi fare la verifica e restituire il valore true o false. Può essere utile disattivare uno di questi controlli se, per esempio, il certificato è scaduto durante lo sviluppo o il test. D'altra parte, questo permette anche di implementare i criteri di convalida più rigorosi di quelli previsti per impostazione predefinita. Nella **figura 7** si fornisce un esempio di richiamata di convalida.
Figure 7 Richiamata di convalida

Copia codice

```
private bool VerifyServerCertificate(
    object sender, X509Certificate certificate,
    X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
    if (sslPolicyErrors == SslPolicyErrors.None) return true;
```

```
    foreach (X509ChainStatus s in chain.ChainStatus)
    {
        // allows expired certificates
        if (string.Equals(s.Status.ToString(), "NotTimeValid",
            StringComparison.OrdinalIgnoreCase))
                return true;
    }

    return false;
}
```

SSL supporta anche l'autenticazione client tramite certificato. Se il sito Web o il servizio cui si vuole accedere impone un certificato client, sia il proxy client del servizio Web sia HttpWebRequest forniscono una proprietà ClientCertificates di tipo X509Certicate:

Copia codice

```
  proxy.Url =
    "https://server/app/service.asmx";
  proxy.ClientCertificates.Add(
    PickCertificate(...));
```

Inoltre, in .NET Framework 2.0 viene introdotta una nuova classe denominata SslStream. Questa classe lascia il livello SSL sopra a qualunque flusso, non solo HTTP, e consente di abilitare per SSL un protocollo personalizzato basato su socket. SslStream utilizza il supporto dei certificati standard .NET in vari modi, per esempio utilizzando il meccanismo di richiamata di convalida già discusso:

Copia codice

```
public SslStream(Stream innerStream, bool leaveInnerStreamOpen,
    RemoteCertificateValidationCallback ValidationCallback) {...}
```

Per iniziare un'autenticazione SSL con SslStream, X509Certificate viene passato al metodo AuthenticateAsServer:

Copia codice

```
ssl.AuthenticateAsServer(PickCertificate(...));
```

Protezione Servizio web

Lo standard Web Services Security specifica l'autenticazione di client e server e la comunicazione protetta tramite i certificati. I toolkit come Web Services Enhancements (WSE) per .NET Framework e le tecnologie come Windows Communication Foundation supportano totalmente questo approccio. Si tratta ancora una volta di fornire un certificato in codice o tramite configurazione. Il frammento di codice seguente mostra come aggiungere un certificato client a un proxy del servizio Web utilizzando WSE3:

Copia codice

```
X509SecurityToken token = new X509SecurityToken(PickCertificate(...));
proxy.RequestSoapContext.Security.Tokens.Add(token);
```

Windows Communication Foundation consente di fornire un riferimento a un archivio certificati in un file di configurazione (vedere la **figura 8**). Come si vede, tutti gli attributi di configurazione eseguono il mapping direttamente agli enumeratori utilizzati in precedenza nel codice.

⊞ Figure 8 Fornire riferimenti del certificato in WCF

Copia codice

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
```

```
      <behavior name="ServiceBehavior">
        <serviceCredentials>
          <serviceCertificate storeLocation="LocalMachine"
            storeName="My" x509FindType="FindBySubjectKeyIdentifier"
            findValue="1a7b..." />
        </serviceCredentials>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

Criteri di protezione e firma del codice
I certificati sono utilizzati anche nella firma codice Authenticode®. Firmando un codice binario, è possibile aggiungere informazioni sull'autore e assicurarsi che il file firmato possa essere convalidato in modo affidabile dopo che è stato firmato. È possibile utilizzare lo strumento signtool.exe di .NET Framework SDK per firmare i file exe e dll. Successivamente, è possibile verificare la firma e vedere il certificato utilizzando la finestra di dialogo proprietà in Esplora risorse. Notare che, se sono utilizzati sia Authenticode che la firma con nome sicuro, quest'ultima deve essere applicata per prima. Inoltre, gli assembly firmati Authenticode possono subire ritardi in fase di caricamento, il che si traduce in un tempo di avvio dell'applicazione superiore se è il punto di ingresso dell'eseguibile che è stato firmato.
I file firmati possono essere utilizzati anche per i criteri di protezione. Utilizzando criteri di restrizione software, è possibile limitare l'esecuzione di eseguibili non gestiti basandosi sulle firme o l'assenza di firme (vedere microsoft.com/technet/prodtechnol/winxppro/maintain/rstrplcy.mspx). Il criterio di protezione accesso al codice (CAS) di .NET Framework supporta i gruppi di codice basati sul certificato dell'autore.
Per creare un criterio CAS, utilizzare mscorcfg.msc per creare un nuovo gruppo di codice basato su una condizione di appartenenza dell'autore. È possibile assegnare poi un insieme di autorizzazioni a tutte le applicazioni firmate da quell'autore (vedere la **figura 9**).

Figura 9 **Assegnazione delle autorizzazioni a un autore** (Fare clic sull'immagine per ingrandirla)

Manifesti ClickOnce
Un'altra tecnologia che utilizza i certificati per le informazioni dell'autore è ClickOnce. Quando si pubblica un'applicazione ClickOnce, si deve firmare il manifesto di distribuzione e applicazione. In tal modo si aggiungono

le informazioni dell'autore all'applicazione e si assicura che le informazioni riservate nei manifesti (come le dipendenze dai criteri di protezione e applicazione) non possano essere modificate senza invalidare la firma. ClickOnce rende le informazioni dell'autore disponibili ai client durante l'installazione, così da permettere decisioni oculate sull'attendibilità dell'applicazione. In funzione del certificato (e del suo risultato di convalida) anche il programma di installazione di ClickOnce utilizza diverse indicazioni visive. Nella **figura 10** viene mostrata la finestra di dialogo di firma del manifesto Visual Studio®.

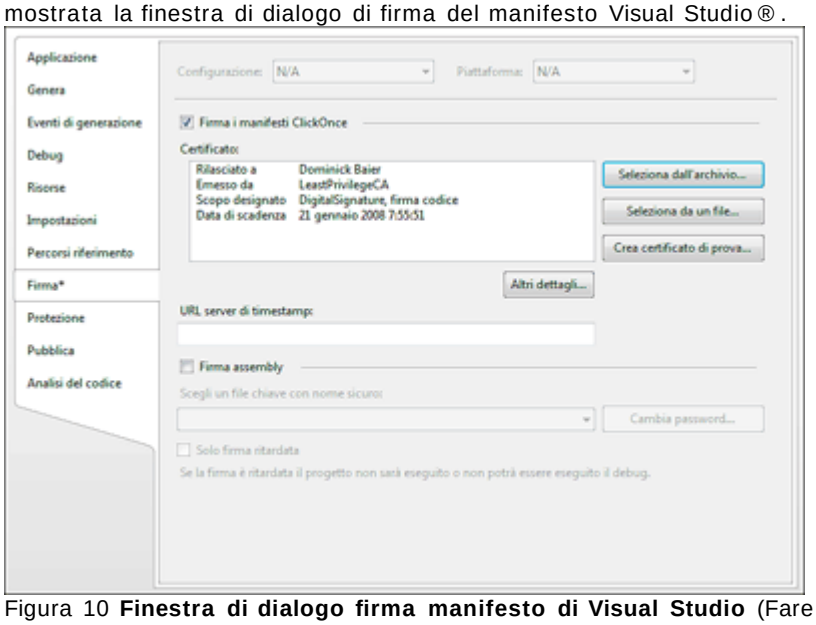

Figura 10 **Finestra di dialogo firma manifesto di Visual Studio** (Fare clic sull'immagine per ingrandirla)

Firma e crittografia dei dati

Fino ad ora, ci siamo soffermati sulle API fondamentali relative ai certificati e su come siano utilizzate dalle altre tecnologie. Restano da descrivere le operazioni di crittografia, nella fattispecie, la crittografia e la firma dei dati con i certificati e la nuova implementazione PKCS#7 che si trova in .NET Framework 2.0.

Il processo di protezione dei dati è sempre diviso in due fasi. Dapprima occorre firmare i dati per renderli non modificabili. Poi occorre crittografare i dati per proteggerli dalla divulgazione. Prima di eseguire qualsiasi operazione di crittografia con le classi PKCS#7, occorre inserire i dati in un oggetto ContentInfo, che rappresenta una struttura di dati CMS. Poi è possibile trasformare i dati in dati firmati o crittografati, rappresentati rispettivamente dalle classi SignedCms ed EnvelopedCms.

Dal punto di vista tecnico, una firma digitale è un hash dei dati che viene poi crittografato con la propria chiave privata. Questo significa che è necessario un certificato con una chiave privata associata o un file pfx. Basandosi su tale certificato, è possibile creare un oggetto CmsSigner, che rappresenta il firmatario dei dati. La classe SignedCms calcola a sua volta la firma e restituisce una matrice di byte PKCS#7, conforme a CMS. Nella **figura 11** viene mostrato il codice corrispondente. La matrice di byte codificata contiene i dati, la firma e il certificato utilizzato per firmare i dati.

Figure 11 Ricavare matrici di byte conformi CMS

Copia codice

```
byte[] Sign(byte[] data, X509Certificate2 signingCert)
{
    // create ContentInfo
    ContentInfo content = new ContentInfo(data);

    // SignedCms represents signed data
    SignedCms signedMessage = new SignedCms(content);

    // create a signer
    CmsSigner signer = new CmsSigner(signingCert);

    // sign the data
    signedMessage.ComputeSignature(signer);
```

```
    // create PKCS #7 byte array
    byte[] signedBytes = signedMessage.Encode();

    // return signed data
    return signedBytes;
}
```

Ciò può non essere di importanza critica se si firmano grandi quantità di dati, ma se l'ammontare di dati è piccolo, sorgono alcuni problemi. Per esempio, firmando una matrice di 10 byte con una chiave pubblica da 2 KB, si ottiene una matrice di circa 2.400 byte. Occorre tenerlo a mente se, ad esempio, si desidera memorizzare i dati firmati in un database. Un approccio alternativo è quello di utilizzare una cosiddetta firma detached. Questo permette di rimuovere i dati dalla firma e di memorizzarli separatamente. Si potrebbe, ad esempio, combinare dapprima blocchi multipli di dati e firmarli poi tutti assieme. Per creare una firma detached si deve passare un ulteriore true al costruttore SignedCms, come illustrato nella **figura 12**.

Figure 12 Creare una firma detached

Copia codice

```
byte[] SignDetached(byte[] data, X509Certificate2 signingCert)
{
    // create ContentInfo
    ContentInfo content = new ContentInfo(data);

    // pass true to the constructor to indicate
    // we want to sign detached
    SignedCms signedMessage = new SignedCms(content, true);

    // these steps are the same
    CmsSigner signer = new CmsSigner(signingCert);
    signedMessage.ComputeSignature(signer);
    byte[] signedBytes = signedMessage.Encode();

    // return only the signature (not the data)
    return signedBytes;
}
```

Una volta firmati i dati, è possibile codificarli. Saranno necessarie le chiavi pubbliche dei destinatari che dovrebbero essere in grado di decrittografare i dati. Questi solitamente si ottengono dall'archivio Contatti (o da un file cer se non si vuole utilizzare l'archivio certificati). Questa volta la classe IEnvelopedCms fa tutto il lavoro. Specificare le chiavi pubbliche utilizzate per la crittografia in un elenco CmsRecipientCollection, che viene passato al metodo Encrypt. Come con SignedCms, il metodo Encode crea la matrice di byte PKCS#7, conforme a CMS (vedere la **figura 13**).

Figure 13 Metodo Encode

Copia codice

```
byte[] Encrypt(byte[] data, X509Certificate2 encryptingCert)
{
    // create ContentInfo
    ContentInfo plainContent = new ContentInfo(data);
```

```
    // EnvelopedCms represents encrypted data
    EnvelopedCms encryptedData = new EnvelopedCms(plainContent);

    // add a recipient
    CmsRecipient recipient = new CmsRecipient(encryptingCert);

    // encrypt data with public key of recipient
    encryptedData.Encrypt(recipient);

    // create PKCS #7 byte array
    byte[] encryptedBytes = encryptedMessage.Encode();

    // return encrypted data
    return encryptedBytes;
}
```

Internamente, EnvelopedCms genera una chiave di sessione casuale con cui i dati vengono crittografati simmetricamente. La chiave di sessione viene quindi crittografata con la chiave pubblica di ogni destinatario. In tal modo, non occorre una versione crittografata separata dei dati per ciascun destinatario. Vengono incorporate, inoltre, alcune informazioni aggiuntive, che consentono al destinatario di trovare la chiave privata corrispondente per la decrittografia nel proprio archivio certificati.

Decrittografare i dati e verificare le firme
In ricezione, l'intero processo è invertito. Ciò significa che dapprima vengono decrittografati i dati quindi vengono convalidati la firma e il certificato di firma. Nel codice, occorre chiamare dapprima il metodo Decode delle classi SignedCms e EnvelopedCms per deserializzare la matrice di byte CMS restituendola a una rappresentazione di oggetto. Poi è possibile chiamare, rispettivamente, Decrypt e CheckSignature.
Per vedere se la chiave di sessione può essere decrittografata, il processo cerca nel pacchetto crittografato una corrispondere chiave privata nell'archivio certificati. Successivamente, la chiave di sessione viene utilizzata per decrittografare i dati reali. È possibile fornire anche un elenco di certificati aggiuntivi che dovrebbero essere considerati durante la decrittografia qualora la chiave privata non sia memorizzata nell'archivio certificati:

Copia codice

```
static byte[] Decrypt(byte[] data)
{
    // create EnvelopedCms
    EnvelopedCms encryptedMessage = new EnvelopedCms();

    // deserialize PKCS#7 byte array
    encryptedMessage.Decode(data);

    // decryt data
    encryptedMessage.Decrypt();

    // return plain text data
    return encryptedMessage.ContentInfo.Content;
```

```
}
```

La verifica dei dati è un processo in due fasi. Prima ci si assicura che la firma è valida, il che significa che i dati non sono stati manomessi. Poi si controlla certificato firma. Il metodo CheckSignature della classe SignedCms permette di eseguire entrambi i passaggi in una volta sola. In questo caso, il certificato è convalidato rispetto ai criteri di sistema predefiniti. Se si vuole un maggiore controllo sul processo, è possibile fare una propria verifica utilizzando un oggetto X509Chain e un codice come quello della **figura 6**. Nella **figura 14** viene mostrato il codice utilizzato per controllare e rimuovere una firma, mentre nella **figura 15** si fornisce il codice utilizzato per la convalida di firma detached.

Figure 15 Convalida di una firma detached

Copia codice

```
static bool VerifyDetached(byte[] data, byte[] signature)
{
    ContentInfo content = new ContentInfo(data);

    // pass true for detached
    SignedCms signedMessage = new SignedCms(content, true);

    // deserialize signature
    signedMessage.Decode(signature);

    try
    {
        // check if signature matches data
        // the certificate is also checked
        signedMessage.CheckSignature(false);
        return true;
    }
    catch { return false; }
}
```

Figure 14 Verificare e rimuovere una firma

Copia codice

```
byte[] VerifyAndRemoveSignature(byte[] data)
{
    // create SignedCms
    SignedCms signedMessage = new SignedCms();

    // deserialize PKCS #7 byte array
    signedMessage.Decode(data);

    // check signature
    // false checks signature and certificate
    // true only checks signature
```

```
    signedMessage.CheckSignature(false);

    // access signature certificates (if needed)
    foreach (SignerInfo signer in signedMessage.SignerInfos)
    {
        Console.WriteLine("Subject: {0}",
          signer.Certificate.Subject);
    }

    // return plain data without signature
    return signedMessage.ContentInfo.Content;
}
```

Riassumendo
Quando si lavora con criteri di protezione o protocolli di comunicazione, si incontrano i certificati in molte situazioni diverse. In questo articolo è stata descritta la API fondamentale utilizzata per recuperare e ricercare i certificati e si è mostrato come utilizzarli per la crittografia e le firme digitali. Sono stati forniti alcuni esempi di servizi di applicazione di alto livello che richiedono la comprensione dell'archivio certificati di Windows e il rapporto tra chiavi pubbliche e private.
Il codice sorgente per questo articolo, disponibile per il download dal sito Web *MSDN® Magazine*, contiene un'applicazione Windows Form che supporta la firma e la crittografia dei file. Nel codice sono utilizzate molte delle tecniche discusse nell'articolo, come la scelta dei certificati da diversi archivi e la protezione e la verifica dei dati tramite la crittografia e le firme.

Dominick Baier è consulente per la protezione e lavora in Germania. Collabora con le aziende nell'ambito della sicurezza di progetti e architettura, sviluppo di contenuti, test di penetrazione e la verifica del codice. È anche responsabile dei corsi sulla sicurezza in DevelopMentor, Developer Security MVP e autore di "Developing More-Secure Microsoft ASP.NET 2.0 Applications", Microsoft Press, 2006 (in inglese). Il suo blog si trova all'indirizzo www.leastprivilege.com.