Chapter
11

# Kerberizing Applications Using Security Support Provider Interface

In this chapter, we examine the Security Support Provider Interface (SSPI), the common interface to request security services in Windows 2000, and show how to use SSPI to kerberize applications. We also define some important security concepts and explain how Windows 2000 provides security services to applications.

## SSPI and Windows 2000 Security Architecture

The **Security Support Provider Interface** (**SSPI**) is an application programming interface (API) that applications should use to request security services from security service providers. In Windows 2000, a **security support provider**—also called a **security provider**—is a dynamic link library that supports the SSPI specification and provides one or more security support packages. Each **security support package** (**SSP**)—also called **security package**—supports one specific security protocol and implements the SSPI functions according to that security protocol. Kerberos, NTLM, and SSL are examples of security packages that are provided by the default security provider in Windows 2000.

SSPI defines the following interface categories:

- **Credential management interfaces** to access credentials, such as passwords, tickets, or to free accessed credentials.
- **Context management interfaces** to create and to use security contexts. Security contexts, created by both sides of a communication link, provide the information needed by message-support interfaces to carry out secure communication.
- **Message support interfaces** to provide communication integrity and privacy.
- **Package management interfaces** to get information about the packages supported by the security provider.

In a typical SSPI scenario, applications first load a security provider DLL and use the package management functions to find information about the security package they want to use. Then, the applications initialize the desired security context through the context management functions and, finally, use the message support functions to sign and to encrypt messages or to verify signatures and to decrypt encrypted messages. By providing these common interfaces, SSPI hides the details of how to use a specific package from the application and also allows an application to use services provided by any security package without changing the interface to use such services.

It is important to know that SSPI does not provide any interfaces for sending and receiving messages. Distributed applications should use other interfaces, such as WinSock, to transfer messages constructed by the SSPI functions.

Figure 11-1 shows how SSPI fits into the overall security architecture of Windows 2000. SSPI is the abstraction layer between applications and security protocols supported by Windows 2000. Applications can use the services of a security package in various ways: either directly or through higher-level protocols or APIs, such as RPC, DCOM, WinSock, or WinInet. Nevertheless, one way or another, access to the security services is always through the SSPI layer.

## SSPI Functions

Table 11-1 shows the SSPI functions supported by the credential management, context management, message support, and package management interface categories, respectively. The `InitSecurityInterface` function returns a pointer to a function table structure that contains pointers to all the SSPI functions. After loading a security

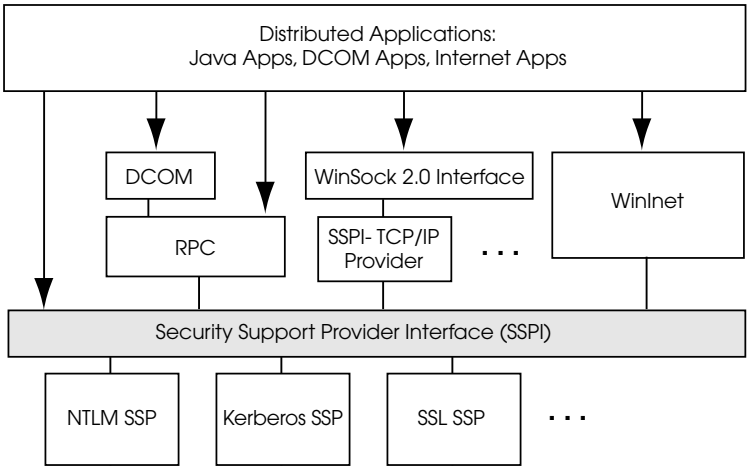


**Figure 11-1** SSPI and the overall Windows 2000 distributed security architecture

**Table 11-1** SSPI Functions

| *Function* | *Description* |
|---|---|
| *Credential Management* | |
| `AcquireCredentialsHandle` | Acquires a handle to the credentials of the specified principal. Refer to Chapter 1 for the definition of a security principal. |
| `FreeCredentialsHandle` | Releases a credential handle and associated resources. |
| `QueryCredentialAttributes` | Queries credential attributes. |
| *Context Management* | |
| `InitializeSecurityContext` | Initiates a security context by generating a security token that must be passed to the server. The application that uses this function is called an SSPI client. |
| `AcceptSecurityContext` | Initiates a security context using the security token received from the client. The application that uses this function is called an SSPI server. |
| `DeleteSecurityContext` | Frees a security context and associated resources. |
| `QueryContextAttributes` | Queries security context attributes for information needed when signing or encrypting messages, and so on. |
| `QuerySecurityContextToken` | Obtains Windows 2000 access token associated with a client security context for direct use. |
| `ApplyControlToken` | Applies a control token to an existing security context. A control token can be used, for example, to gracefully shut down a secure connection in SSL. SSPI applications using Kerberos would not need this function. |
| `CompleteAuthToken` | Completes an authentication token. Used in some protocols, such as DCE RPC, in which the security context needs to be revised after the application has updated some message fields. Not used in Kerberos. |
| `FreeContextBuffer` | Frees a memory buffer allocated by the security provider. |
| `ImpersonateSecurityContext` | Impersonates the client's security context. |
| `RevertSecurityContext` | Stops impersonation. |
| `ExportSecurityContext` | Exports a security context into a buffer for later use. |

(*continued*)

**Table 11-1** SSPI Functions (*cont.*)

| *Function* | *Description* |
|---|---|
| `ImportSecurityContext` | Imports an exported security context into the current process. |
| *Message Support* | |
| `DecryptMessage` | Decrypts a message created by a call to `EncryptMessage`. |
| `EncryptMessage` | Encrypts a message. |
| `MakeSignature` | Signs a message. |
| `VerifySignature` | Verifies a signature created by a call to `MakeSignature`. |
| *Package Management* | |
| `InitSecurityInterface` | Returns a pointer to the SSPI function table. |
| `EnumerateSecurityPackages` | Lists all available security packages and their attributes. |
| `QuerySecurityPackageInfo` | Queries an individual security package for its attributes. |

provider DLL, SSPI clients bind to the implementation of the SSPI functions at runtime by calling the `InitSecurityInterface` function. All the SSPI functions are called by using this function table. If a function is not implemented, its function pointer is set to `NULL` in the function table. SSPI clients should make sure that a function is implemented by comparing the function pointer to `NULL` before calling the function.

## Using SSPI

To use SSPI, distributed applications go through the following three major steps:

1. Initialize the SSPI.
2. Authenticate the connection.
3. Exchange secure messages over the authenticated connection, using security services, such as integrity, privacy, and so on.

The rest of this section gives an overview of what happens in each of these three steps on each side of the communication link. More detailed information comes

later, when we go through our SSPI sample project on how to kerberize a distributed application.

### Initializing the SSPI

Table 11-2 summarizes what needs to be done to initialize a security support package. Both the client and the server go through the initialization step in the same way and should choose the same package in order to be able to establish an authenticated connection and exchange messages.

Windows 2000 supports a security negotiation package that automatically selects a security package acceptable to both ends of communication. The *negotiate* package attempts to use the Kerberos package first and, if security negotiation fails, tries the NTLIM package next.

### Authenticating the Connection

Establishing an authenticated connection is more complicated and differs for the client and the server. Here, the term client means SSPI client, which is that side of the communication that starts the process of building the security context by calling the `InitializeSecurityContext` function. The term server means SSPI server, which is the side that waits to receive a security token from the SSPI client and calls the `AcceptSecurityContext` function. An SSPI client could be the server part of a distributed application. Tables 11-3 and 11-4 summarize the steps to initialize a security support package on the client and server side, respectively.

When `AcquireCredentialsHandle` is called, the name of the chosen security package is passed to obtain the correct type of credential. Each security package requires a different type of credential, and the specifics of this operation are abstracted by SSPI.

**Table 11-2** Initializing the SSPI for Client and Server

| *Step* | *Description* |
|---|---|
| Load the security support provider. | Use the `LoadLibrary` function to load the DLL that implements the package. Load `secur32.dll` for the default security provider. |
| Get a pointer to the package function table. | Use the `GetProcAddress` function to obtain the address of the `InitSecurityInterface` function. Get the security provider's function table by calling the `InitSecurityInterface` function. |
| Find the desired security package | Call the `EnumerateSecurityPackages` function to get information about security packages and choose one. |

**Table 11-3** Authentication Steps for the Client

| *Step* | *Description* |
|---|---|
| Get a handle to the credentials to use for authentication. | Use the `AcquireCredentialsHandle` function to get a handle to the client's credentials. |
| Authenticate and build the security context. | Call the `InitializeSecurityContext` function as many times as needed to authenticate the connection and to build the security context. After each call, send any output from this function to the server. If you need to call this function again, wait to receive information from the server, and pass the received information to this function. The return value of this function tells you whether you need to call this function again. |
| Check the security context. | Make sure that the final attributes of the security context, returned in the final call to `InitializeSecurityContext`, are sufficient. If not, you may want to deny the connection. |

In their calls to `InitializeSecurityContext` and `AcceptSecurityContext`, the client and the server, respectively, can specify the type of authentication and security environment needed. At the end of the authentication step, each side should check and make sure that the established environment has the required security attributes and close the communication channel if that is not the case. Every call to these functions may have some output data that must be sent to the other. The return values of these functions specify whether they need to be called again. If so, the client or the server has to wait to receive security data from the other side and pass it to the function in the next call.

When Kerberos is used, a three-leg authentication is carried out, regardless of the client's choice for mutual authentication. This means that there will be two calls to the `InitializeSecurityContext` function and one call to the `AcceptSecurityContext` function. In the first call to `InitializeSecurityContext`, the client communicates with the Key Distribution Center (KDC) to obtain a session ticket for the specified server, as specified by the Ticket Granting Service (TGS) Exchange protocol, if such a ticket is not already in the ticket cache. When it receives the session ticket—the output from the `InitializeSecurityContext` function—the server validates the ticket by calling the `AcceptSecurityContext` function, as specified by the Client/Server (CS) Exchange

**Table 11-4** Authentication Steps for the Server

| *Step* | *Description* |
|---|---|
| Get a handle to the credentials to use for authentication. | Use the `AcquireCredentialsHandle` function to get a handle to the server's credentials. |
| Authenticate and build the security context | Call the `AcceptSecurityContext` function as many times as needed to authenticate the connection and to build the security context. Pass the information received from the client to this function every time you call this function. After each call, send the output from this function to the client and wait to receive information from the client if there is a need. The return value of this function tells you whether you need to call this function again. |
| Check the security context. | Make sure that the final attributes of the security context, returned in the final call to `AcceptSecurityContext`, are sufficient. If not, you may want to deny the connection. |

protocol. The output of the `AcceptSecurityContext` function needs to be sent to the client so that it can complete the construction of its security token. If mutual authentication is requested, this output also contains the authenticator message—the `KRB_AP_REP` message—that the client validates in its second call to the `InitializeSecurityContext` function.

As mentioned earlier, it is up to the applications to exchange the data returned by the SSPI functions. Applications should define an application-level protocol and message formats for such exchanges.

### *Exchanging Messages*

Having established an authenticated connection, the client and the server can proceed with exchanging secure messages by signing or encrypting messages, as shown in Table 11-5. To be able to use message signing and encryption, appropriate flags should be passed to the `InitializeSecurityContext` or the `AcceptSecurityContext` functions so that the appropriate security context is created during the authentication step. Both the client and the server can request such services and could end the connection if the requested services are denied.

**Security context attributes** specify the capabilities of the security context. Both sides of the communication can ask for security attributes when using the

InitializeSecurityContext and the AcceptSecurityContext functions, by combining the values specified in Table 11-6. When using attributes, be aware that

- Not all security attributes are available in a security package.
- Some attributes, such as IDENTIFY, can be requested only by a specific side of the connection.
- Some attributes, such as CONFIDENTIALITY, can be requested by either side of the communication.
- Some attributes, such as CONNECTION, are honored only when requested by both sides of the connection.

The most important security attributes supported in Kerberos are listed in Table 11-6. For a complete list of context attributes, refer to [MICR00].

To make sure that the established context attributes are acceptable, both the client and the server should check the arguments in the InitializeSecurity Context and the AcceptSecurityContext functions, which specify the established attributes. If no context attributes are requested, the security package will use its default context attributes. In Kerberos, for example, the default security context allows only client authentication and impersonation. To be able to do mutual authentication, message signing, or encryption, the related context attribute should be requested.

The preceding security attributes should be prefixed with ISC_REQ_ and ASC_REQ_ before being used in the InitializeSecurityContext and the AcceptSecurityContext functions, respectively. For example, to request mutual authentication, the client should specify ISC_REQ_MUTUAL_AUTH with the InitializeSecurityContext function. The server can request confidentiality by using ASC_REQ_CONFIDENTIALITY with the

**Table 11-5** Message Exchange Steps for Client and Server

| *Step* | *Description* |
|---|---|
| Sign a message before sending it. | Use the MakeSignature function to sign. |
| Verify a signature on a signed message before accepting it. | Use the VerifySignature function to verify the signature. |
| Encrypt a message before sending it. | Use the EncryptMessage function to encrypt a message. |
| Decrypt an encrypted message. | Use the DecryptMessage function to decrypt an encrypted message. |

**Table 11-6** Security Context Attributes in Kerberos

| *Attribute* | *Description* |
|---|---|
| IDENTIFY | The server can identify but cannot impersonate or delegate the client. This attribute is not supported by Kerberos and is available only in the SSPI specification. This attribute can be set only by the client side. |
| DELEGATE | The server can build a new security context impersonating the client, and that context will be accepted by other remote servers as the client's context. When neither DELEGATE nor IDENTIFY is used, impersonation is assumed by default. This attribute can be set only by the client side. |
| MUTUAL_AUTH | The communicating parties must authenticate their identities to each other. In Kerberos, without MUTUAL_AUTH, the client authenticates to the server; with MUTUAL_AUTH, the server must authenticate to the client. This flag can be set only by the client. |
| REPLAY_DETECT | Security package checks for replayed packets and notifies the caller if a packet has been replayed. The use of this flag implies all the conditions specified by the INTEGRITY flag. |
| SEQUENCE_DETECT | Out-of-order packets could be detected through the message support functions. Use of this flag implies all the conditions specified by the INTEGRITY flag. |
| CONFIDENTIALITY | Data can be encrypted using the message support functions. Use this flag if you need to encrypt and to decrypt the data. |
| INTEGRITY | Integrity can be verified, but no sequencing or replay detection is enabled. Use this flag if you need to sign the data or to verify a signature. |
| CONNECTION | Connection-oriented context. Provides support for connection-based distributed applications. |
| DATAGRAM | Connectionless context. Provides support for datagram-based and DCE-style RPC-based distributed applications. See SSPI documentation [MICR00] for more information. |

AcceptSecurityContext function. The established context attributes are prefixed by ISC_RET_ and ASC_RET.

## Impersonation and Delegation

Impersonation is an important concept in client/server communication. A server application may run under a security context that has more privileges than the

client requesting the service. Before servicing the request, such a server should make sure that the client has sufficient access rights for the requested operation.

One approach to client authorization is for the server to keep and to use its own per client authorization information to determine whether an operation can be performed by the client. This approach is difficult to implement and to manage. The second approach is to have the operating system determine whether a client is authorized to perform an operation, using a mechanism called impersonation.

**Impersonation** is the temporary change of the security context to use the security context of another principal. When a server impersonates a client before accessing a resource, the operating system checks access rights under the new—client—security context automatically. Moving the task of checking access rights to the operating system simplifies server applications quite a bit. There is no need to maintain separate per client authorization data and write application code to enforce access rights.

In order for a server to impersonate a client in SSPI, the client should specify the impersonation level that the server is allowed to use when servicing the request. Windows 2000 has the following four impersonation levels:

- **Anonymous**—the server is not allowed to find any information on the client's security context.
- **Identification**—the server can only authenticate the client but cannot use the client's security context for access checks.
- **Impersonation**—the server can authenticate the client and use the client's security context for local access checks directly. The server can also pass the context to another server on the same machine.
- **Delegation**—the server can authenticate the client and use the client's security context for local access checks (direct or via other servers). The server can also pass on the context to a remote server to request service on behalf of the client.

An SSPI client should use `ISC_REQ_IDENTITY` to limit the server to do only identification and should use `ISC_REQ_DELEGATE` to allow for delegation. A server's request for delegation is never honored by the security package, as this attribute is controlled by the client. If neither `ISC_REQ_IDENTITY` nor `ISC_REQ_DELEGATE` is used, impersonation is granted by default.

A server that wants to use delegation should use `SECPKG_CRED_BOTH` in its call to the `AcquireCredentialsHandle` function. To do delegation, the server should impersonate the client after the authentication and go through the authentication step with the new server as a client. This is done by calling the `InitializeSecurityContext` function under the client's identity and going through the authentication loop, as explained

earlier. If delegation is allowed in the new connection, the second server can also impersonate the original client and delegate the credentials exactly as the first server did. There is no limit to the number of delegations in such a scenario.

For delegation to work, you need to make sure that all of the following conditions are true:

- Client account can be delegated.
- Service log-on account can delegate.
- Service host computer is trusted for delegation.

To make sure that the client account can be delegated, unselect the Account is sensitive and cannot be delegated option. To verify that the service log-on account can delegate, select the Account is trusted for delegation option. Both of these options are available in the properties of the account object in Active Directory. To make sure that the computer on which the service is running is trusted for delegation, select the Computer is trusted for delegation option in the properties of the computer object in Active Directory. Note that the last two settings should be true for all the services and computers that participate in delegation.

A security package that supports impersonation sets the `SECPKG_FLAG_IMPERSONATION` flag in the `fCapabilities` field of `SecPkgInfo`. A security package that supports delegation sets the `SECPKG_FLAG_DELEGATION` flag in the `fCapabilities` field of `SecPkgInfo`. These capabilities can be obtained by using the package management functions. Delegation is only supported by the Kerberos package.

## Sample Project: Using SSPI to Kerberize Applications

In this SSPI sample project, we will define a few C++ classes to hide complexities of using SSPI and Kerberos to kerberize applications. As you will see, the classes are designed to facilitate the authentication and secure messaging in a socket-based client/server application. In our sample project, we follow the steps we outlined earlier on how to use SSPI. Note that all the code listings show parts of the C++ classes. While reading the code, keep in mind that error handling is often omitted for brevity. The complete project can be found on the companion CD-ROM.

Let's first look at the `CKerberos` top-level class. The constructor for this class loads the `secur32.dll` library, the default security provider, and then gets a pointer to the security provider function table, finds the Kerberos package, and uses the maximum token size of the package to initialize buffers used later during the authentication step. In Kerberos, `SSP_NAME` and `KERB_PACKAGE` are defined as `secur32.dll` and `kerberos`, respectively, and `SECURITY_ENTRYPOINT` is set to `InitSecurityInterface`.

**Listing 11-1** CKerberos Class

```
//--------------------------------------------------------------------
// CKerberos - Constructor to load Kerberos package and
// initialize vars
//--------------------------------------------------------------------
CKerberos::CKerberos()
{
   FARPROC           pInit;
   SECURITY_STATUS   ss;
   PSecPkgInfo       pkgInfo;

   m_pInBuf = NULL;
   m_pOutBuf = NULL;
   m_hLib = NULL;
   // Load and initialize the default ssp
   //
   m_hLib = LoadLibrary (SSP_NAME);
   pInit = GetProcAddress (m_hLib, SECURITY_ENTRYPOINT);
   m_pFuncTbl = (PSecurityFunctionTable) pInit ();
   // Query for Kerberos package
   //
   ss = m_pFuncTbl->QuerySecurityPackageInfo (KERB_PACKAGE,
 &pkgInfo);
   if (!SEC_SUCCESS(ss)){
      throw CKerberosErr ("Couldn't query the package. Package may not
exist", ss);
   }

   // Initialize vars
   //
   m_cbMaxToken = pkgInfo->cbMaxToken;
   m_pFuncTbl->FreeContextBuffer (pkgInfo);
```

```
    m_pInBuf = (PBYTE) malloc (m_cbMaxToken);
    m_pOutBuf = (PBYTE) malloc (m_cbMaxToken);
    m_pkgName = KERB_PACKAGE;
    m_fHaveCtxtHandle = false;
    m_fHaveCredHandle = false;
    m_reqCtxtAttrs = 0;
}
```

We derive two classes from `CKerberos`: `CKerberosClient` and `CKerberosServer`. After loading the default security provider and setting up variables through the `CKerberos` constructor, each class's constructor calls the `AcquireCredentialsHandle` function to get a handle to the logged-on user's credentials. The first argument to this function is `NULL` because we want to use the credentials of the current logged-on user. We have to pass the name of the package we want to use—second argument—and the usage type to get the correct type of credentials. `CKerberosClient` uses `SEKPKG_CRED_OUTBOUND` for the credential usage, whereas `CKerberosServer` uses `SEKPKG_CRED_BOTH` to be able to do delegation if needed later. `NULL` is passed for arguments that are not used by the Kerberos package or that we are not interested in.

**Listing 11-2** `CKerberosClient` Class

```
//--------------------------------------------------------------------
// CKerberosClient - Constructor to get credential handles
//--------------------------------------------------------------------
CKerberosClient::CKerberosClient(LONG ctxtAttrs)
{
    SECURITY_STATUS ss;

    ss = m_pFuncTbl->AcquireCredentialsHandle (
              NULL,   // Use current principal
              m_pkgName, // Set to "kerberos"
              SECPKG_CRED_OUTBOUND,
              NULL,
              NULL,
              NULL,
              NULL,
```

```
            &m_hCred, // Handle to the credentials
            &m_credLifetime);

   if (SEC_E_OK != ss){
         throw CKerberosErr ("AcquireCredentialsHandle failed",
                  ss);
   }
   m_fHaveCredHandle = true;
   m_reqCtxtAttrs = ctxtAttrs;
}
```

Both constructors of these derived classes take an argument that specifies the requested context attributes. The caller applications should use the appropriate context attributes flags when using these classes.

**Listing 11-3** KerberosServer Class

```
//--------------------------------------------------------------
// CKerberosServer - Constructor to get credential handles
//--------------------------------------------------------------
CKerberosServer::CKerberosServer(LONG ctxtAttrs)
{
   SECURITY_STATUS ss;

   ss = m_pFuncTbl->AcquireCredentialsHandle (
            NULL,    // Use current user
            m_pkgName, // Set to "kerberos"
            SECPKG_CRED_BOTH,
            NULL,
            NULL,
            NULL,
            NULL,
            &m_hCred, // Handle to the credentials
            &m_credLifetime
            );
```

```
   if (SEC_E_OK != ss){
         throw CKerberosErr ("AcquireCredentialsHandle failed",
               ss);
   }

   m_reqCtxtAttrs = ctxtAttrs;
   m_fHaveCredHandle = true;
}
```

The `Authenticate` method of each class goes through the process of building the security token and sending it to the other side until the authentication is completed or failed. The security token is built by calling `InitializeSecurityContext` when in the `CKerberosClient::Authenticate` method. In the first call to `InitializeSecurityContext`, we do not use any input buffer, as there is no information received from the server to pass to this function yet. The client keeps calling this function until the return code specifies that there is no need to do so anymore. In subsequent calls to the function, we pass the security token we receive from the server in `inSecBuffDesc`. The security package uses the content of this buffer to build the security context and to authenticate the server if needed. The output buffer in `outSecBuffDesc` is the security token, under construction, we send to the server so that it can authenticate the client. By default, in Kerberos, the server authenticates the client. By requesting mutual authentication, a client is asking Kerberos to authenticate the server too. It is up to the client to terminate a connection if mutual authentication never took place, as shown in the following listing.

**Listing 11-4** Client-Side Authentication

```
//--------------------------------------------------------------------
// Authenticate - Authenticate the connection (client side)
//                Target is the SPN of the service or the
//                "domain\\username" of the service account
//--------------------------------------------------------------------
void CKerberosClient::Authenticate(SOCKET s, SEC_CHAR *target)
{
   SECURITY_STATUS  ss;
   SecBufferDesc          outSecBufDesc;
```

```
SecBuffer        outSecBuf;
SecBufferDesc          inSecBufDesc;
SecBuffer        inSecBuf;
BOOL                   done = false;
DWORD                        cbIn;

// prepare output buffer
//
outSecBufDesc.ulVersion = 0;
outSecBufDesc.cBuffers = 1;
outSecBufDesc.pBuffers = &outSecBuf;
outSecBuf.cbBuffer = m_cbMaxToken;
outSecBuf.BufferType = SECBUFFER_TOKEN;
outSecBuf.pvBuffer = m_pOutBuf;

while (!done){
      ss = m_pFuncTbl->InitializeSecurityContext (
                     &m_hCred,
                     m_fHaveCtxtHandle ? &m_hCtxt : NULL,
                     target,
                     // context requirements
                     m_reqCtxtAttrs,
                     0,      // reserved1
                     SECURITY_NATIVE_DREP,
                     m_fHaveCtxtHandle ? &inSecBufDesc : NULL,
                     0,
                     &m_hCtxt,
                     &outSecBufDesc,
                     &m_ctxtAttr,
                     &m_ctxtLifetime
                     );
       if (!SEC_SUCCESS (ss)){
            throw CKerberosErr ("InitializeSecurityContext
                              failed", ss);
```

```
            }
            m_fHaveCtxtHandle = TRUE;
            done = !((SEC_I_CONTINUE_NEEDED == ss) ||
            (SEC_I_COMPLETE_AND_CONTINUE == ss));

            // Send to the server if we have anything to send
            //
            if (outSecBuf.cbBuffer){
               SendMsg (s, m_pOutBuf, outSecBuf.cbBuffer);
            }
            if (!done){
                RecvMsg (s, m_pInBuf, m_cbMaxToken, &cbIn);
            }
            // Prepare input buffer for next round
            //
            inSecBufDesc.ulVersion = 0;
            inSecBufDesc.cBuffers = 1;
            inSecBufDesc.pBuffers = &inSecBuf;
            inSecBuf.cbBuffer = cbIn;
            inSecBuf.BufferType = SECBUFFER_TOKEN;
            inSecBuf.pvBuffer = m_pInBuf;
            // Reset the size on output buffer. We reuse it.
            //
            outSecBuf.cbBuffer = m_cbMaxToken;
         }
      }
      // Check if we did mutual attribute if requested.
      // CheckCtxtAttr throws an exception if the attribute
      // is not set
      if (m_reqCtxtAttrs & ISC_RET_MUTUAL_AUTH){
            CheckCtxtAttr (ISC_RET_MUTUAL_AUTH);
      }
   }
```

As far as the server part is concerned, after a request is received from the client for authentication, the server starts the process by setting up the security buffers and calling the `AcceptSecurityContext` function. This is very similar to the client's `Authenticate` method. Note that both the client and the server `Authenticate` method take a socket argument as the first argument. The socket connection should be set up before using these methods by the caller applications. We use blocking sockets for simplicity.

We have defined an application-level protocol to send and to receive messages between the client and the server in our `SendMsg` and `RecvMsg` functions. In our simple protocol, which is suitable for TCP connections, we always send the size of the message first. See the sample code on the CD-ROM for more information on these functions and others not shown here.

**Listing 11-5** Server-Side Authentication

```
//--------------------------------------------------------------------
// Authenticate - Authenticate the connection (server side)
//--------------------------------------------------------------------
void CKerberosServer::Authenticate(SOCKET s)
{
   SECURITY_STATUS  ss;
   SecBufferDesc          outSecBufDesc;
   SecBuffer        outSecBuf;
   SecBufferDesc          inSecBufDesc;
   SecBuffer        inSecBuf;
   BOOL                   done = false;
   DWORD                  cbIn;

   // Prepare output buffer
   //
   outSecBufDesc.ulVersion = 0;
   outSecBufDesc.cBuffers = 1;
   outSecBufDesc.pBuffers = &outSecBuf;
   outSecBuf.cbBuffer = m_cbMaxToken;
   outSecBuf.BufferType = SECBUFFER_TOKEN;
   outSecBuf.pvBuffer = m_pOutBuf;
```

```
while (!done){
     // Get the security buffer from the client
     //
     RecvMsg (s, m_pInBuf, m_cbMaxToken, &cbIn);
     // prepare input buffer for second round
     //
     inSecBufDesc.ulVersion = 0;
     inSecBufDesc.cBuffers = 1;
     inSecBufDesc.pBuffers = &inSecBuf;
     inSecBuf.cbBuffer = cbIn;
     inSecBuf.BufferType = SECBUFFER_TOKEN;
     inSecBuf.pvBuffer = m_pInBuf;
    // Reset the size on output buffer. We reuse it.
    //
     outSecBuf.cbBuffer = m_cbMaxToken;

     ss = m_pFuncTbl->AcceptSecurityContext (
            &m_hCred,
            m_fHaveCtxtHandle ? &m_hCtxt : NULL,
            &inSecBufDesc,
            m_reqCtxtAttrs,
            SECURITY_NATIVE_DREP,
            &m_hCtxt,
            &outSecBufDesc,
            &m_ctxtAttr,
            &m_ctxtLifetime
            );
      if (!SEC_SUCCESS (ss)){
            throw CKerberosErr ("AcceptSecurityContext failed",
                        ss);
      }
      m_fHaveCtxtHandle = TRUE;
      done = !((SEC_I_CONTINUE_NEEDED == ss) ||
              (SEC_I_COMPLETE_AND_CONTINUE == ss));
```

```
            // Send to the client if we have anything to send
            //
            if (outSecBuf.cbBuffer){
               SendMsg (s, m_pOutBuf, outSecBuf.cbBuffer);
            }
         }
      }
```

After establishing a secure connection, the client and the server can start exchanging secure messages. The `CKerberos` class defines four methods for secure messaging: `SendSignedMsg`, `RecvSignedMsg`, `SendEncryptedMsg`, and `RecvEncryptedMsg`. These methods should be used only if appropriate flags are passed to the constructors; otherwise, exceptions are raised.

To sign a message, two security buffers need to be allocated: one to keep the message and one for the signature itself. The size of the buffer needed to keep the signature is specified in the `SECPKG_ATTR_SIZES` structure, which we query by using the `QuerySecurityAttributes` function.

**Listing 11-6** Sending a Signed Message

```
//--------------------------------------------------------------
// SendSignedMsg - Send a signed message. First send the message
// and then the signature.
//--------------------------------------------------------------
void CKerberos::SendSignedMsg (SOCKET s, PBYTE pBuf,
DWORD cbBuf)
{
   SECURITY_STATUS      ss;
   SecPkgContext_Sizes  ctxtSizes;
   SecBufferDesc        secBufDesc;
   SecBuffer            secBufs[2];

   ss = m_pFuncTbl->QueryContextAttributes(
               &m_hCtxt,
               SECPKG_ATTR_SIZES,
               &ctxtSizes);
```

```
if (ctxtSizes.cbMaxSignature == 0){
      throw CKerberosErr("Message signing not supported");
}
secBufDesc.cBuffers = 2;
secBufDesc.pBuffers = secBufs;
secBufDesc.ulVersion = SECBUFFER_VERSION;
secBufs[0].BufferType = SECBUFFER_DATA;
secBufs[0].cbBuffer = cbBuf;
secBufs[0].pvBuffer = pBuf;

// Build a security buffer for the message signature.
//
secBufs[1].BufferType = SECBUFFER_TOKEN;
secBufs[1].cbBuffer = ctxtSizes.cbMaxSignature;
secBufs[1].pvBuffer = (void *)malloc (
                                ctxtSizes.cbMaxSignature);
// Sign the message
//
ss = m_pFuncTbl->MakeSignature(
                 &m_hCtxt,
                0,     // No quality of service in Kerberos
                &secBufDesc,
                0);    // We don't use sequence numbers

if (!SEC_SUCCESS(ss)){
          throw CKerberosErr ("MakeSignature failed" , ss);
}
// Send the message first
//
SendMsg(s, (BYTE *)secBufs[0].pvBuffer,
                          secBufs[0].cbBuffer);
                        .
// Send the signature next. That is our
```

```
   // protocol
   SendMsg(s, (BYTE *)secBufs[1].pvBuffer,
                              secBufs[1].cbBuffer);
   free (secBufs[1].pvBuffer);
}
```

The security buffer that keeps the message has the SECBUFFER_DATA type. The second buffer, which keeps the signature, is of SECBUFFER_TOKEN type and should have a size equal to cbMaxSignature. After setting up the buffers, we sign the message. Our protocol is to send the message size first and then the message itself, in the same order in which we receive them in the RecvSignedMsg function. Note that no quality of service is supported by Kerberos.

If you request out-of-sequence packet detection in any of the constructors, you can pass a sequence number to the MakeSignature call. If you do so, the security package will keep this information in the signature and will report an error if you receive an out-of-sequence packet when verifying the signature. The security package does this by comparing the sequence number in the signature with the one you provide in the VerfiySignature function. The application is responsible for maintaining the sequence numbers.

To verify the signature, after receiving the signature and the message, we allocate two security buffers and initialize the SECBUFFER_TOKEN with the received signature. The SECBUFFER_DATA keeps the message. The security buffers are put in the security buffer description and passed to the VerifySignature function.

**Listing 11-7** Receiving a Signed Message

```
//--------------------------------------------------------------------
// RecvSignedMsg - Receive a signed message sent by calling
//                 SendSignedMsg.
//--------------------------------------------------------------------
PBYTE CKerberos::RecvSignedMsg (SOCKET s, DWORD *cbBuf)
{
   PBYTE             pBuf;
   PBYTE             sigBuf;
   DWORD             sigBufSize;
   SECURITY_STATUS   ss;
   SecBufferDesc     secBufDesc;
```

```
SecBuffer         secBuf[2];
DWORD             cbRead;
ULONG             fQOP;

// Receive the message size first
//
RecvBytes(s, (PBYTE) cbBuf, sizeof (*cbBuf), &cbRead);

pBuf = (PBYTE) malloc (*cbBuf);
// Receive the message
//
RecvBytes(s, pBuf, *cbBuf, &cbRead);
// Receive the signature size first
//
RecvBytes(s, (PBYTE) &sigBufSize,
                sizeof (sigBufSize),&cbRead);
sigBuf = (PBYTE) malloc (sigBufSize);
// Receive the signature
//
RecvBytes(s, sigBuf, sigBufSize, &cbRead);
// Build the input buffer descriptor
//
secBufDesc.cBuffers = 2;
secBufDesc.pBuffers = secBuf;
secBufDesc.ulVersion = SECBUFFER_VERSION;
// Build the security buffer for message
//
secBuf[0].BufferType = SECBUFFER_DATA;
secBuf[0].cbBuffer = *cbBuf;
secBuf[0].pvBuffer = pBuf;
// Build the security buffer for signature
//
secBuf[1].BufferType = SECBUFFER_TOKEN;
```

```
   secBuf[1].cbBuffer = sigBufSize;
   secBuf[1].pvBuffer = sigBuf;
   // Verify the signature
   //
   ss = m_pFuncTbl->VerifySignature(&m_hCtxt,
                            &secBufDesc,
                        0,  // no sequence number used
                        &fQOP);


   if(SEC_SUCCESS(ss)){
      // OK. Return the message
       return pBuf;
   }
   else if(ss == SEC_E_MESSAGE_ALTERED){
           throw CKerberosErr("The message was tampered with");
   }
   else if (ss == SEC_E_OUT_OF_SEQUENCE ){
            throw CKerberosErr("The message is out of sequence.");
   }
   else{
            throw CKerberosErr("VerifySignature failed with unknown
            error");
   }
}
```

To read the message and the signature, we first read the size of the message and the signature, allocate enough memory, and then read the message and the signature. We also read the message first and then the signature because that is the order in which the `SendSignedMsg` function sends them.

Message encryption and decryption work in a similar way. The difference is in the way we set up the security buffers. To encrypt a message, we first query the security context for the encryption-related size information. We then use this information to set up three security buffers: one to keep the trailer, one to keep the message to be encrypted, and one for the padding. The types of these buffers are `SECBUFFER_`

TOKEN, SECBUFFER_DATA, and SECBUFFER_PADDING, respectively. The message is encrypted in place. We then collect all the data and send it to the other side.

**Listing 11-8** Sending an Encrypted Message

```
//-------------------------------------------------------------
// SendEncryptedMsg - Encrypt a message and send the encrypted
//                          message
//-------------------------------------------------------------
void CKerberos::SendEncryptedMsg (SOCKET s, PBYTE pBuf,
             DWORD cbBuf)
{
   SECURITY_STATUS      ss;
   SecBuffer            inSecBuf, outSecBuf;
   SecBuffer            secBufs[3];
   SecBufferDesc        secBufDesc;
   SecPkgContext_Sizes  sizes;

   ss = m_pFuncTbl->QueryContextAttributes(
             &m_hCtxt,
             SECPKG_ATTR_SIZES,
             &sizes);

   if(SEC_E_OK != ss){
             throw CKerberosErr("Error reading SECPKG_ATTR_SIZES",
                      ss);
   }

   // Prepare buffers to encrypt the message
   //
   secBufDesc.cBuffers = 3;
   secBufDesc.pBuffers = secBufs;
   secBufDesc.ulVersion = SECBUFFER_VERSION;
   inSecBuf.pvBuffer = pBuf;
   inSecBuf.cbBuffer = cbBuf;
```

```
secBufs[0].cbBuffer = sizes.cbSecurityTrailer;
secBufs[0].BufferType = SECBUFFER_TOKEN;
secBufs[0].pvBuffer = malloc(sizes.cbSecurityTrailer);
secBufs[1].BufferType = SECBUFFER_DATA;
secBufs[1].cbBuffer = inSecBuf.cbBuffer;
secBufs[1].pvBuffer = malloc(secBufs[1].cbBuffer);

memcpy(secBufs[1].pvBuffer, inSecBuf.pvBuffer,
           inSecBuf.cbBuffer);

secBufs[2].BufferType = SECBUFFER_PADDING;
secBufs[2].cbBuffer = sizes.cbBlockSize;
secBufs[2].pvBuffer = malloc(secBufs[2].cbBuffer);

// Encrypt the message
//
ss = m_pFuncTbl->EncryptMessage(&m_hCtxt, 0, &secBufDesc, 0);

if (ss != SEC_E_OK){
   throw CKerberosErr(" EncryptMessage failed", ss);
}
// Create the mesage to send
//
outSecBuf.cbBuffer = secBufs[0].cbBuffer +
secBufs[1].cbBuffer + secBufs[2].cbBuffer;
outSecBuf.pvBuffer = malloc(outSecBuf.cbBuffer);

memcpy(outSecBuf.pvBuffer, secBufs[0].pvBuffer,
   secBufs[0].cbBuffer);
memcpy((PUCHAR) outSecBuf.pvBuffer + (int)
   secBufs[0].cbBuffer, secBufs[1].pvBuffer,
   secBufs[1].cbBuffer);
memcpy((PUCHAR) outSecBuf.pvBuffer + secBufs[0].cbBuffer +
secBufs[1].cbBuffer,
```

```
            secBufs[2].pvBuffer,
            secBufs[2].cbBuffer);

   free (secBufs[0].pvBuffer);
   free (secBufs[1].pvBuffer);
   free (secBufs[2].pvBuffer);
   // Send everything to the server
   //
   SendMsg(s, (PBYTE) outSecBuf.pvBuffer, outSecBuf.cbBuffer);
   free(outSecBuf.pvBuffer);
 }
```

To decrypt a message, we first receive the encrypted message. Because we don't know the size of the message, we first read the size information, allocate enough memory, and then receive the encrypted message. To do the decryption, we need to set up two security buffers this time: one to keep the encrypted message and one to hold the result of the decryption process. The types of these buffers are SECBUFFER_STREAM and SECBUFFER_DATA, respectively. We then call the decryption function.

**Listing 11-9** Receiving an Encrypted Message

```
//--------------------------------------------------------------
// RecvEncryptedMsg - Receive an encrypted message sent by
// SendEncryptedMsg
//--------------------------------------------------------------
PBYTE CKerberos::RecvEncryptedMsg (SOCKET s, DWORD *cbBuf)
{
   SECURITY_STATUS      ss;
   SecBuffer            secBufs[2];
   SecBufferDesc        secBufDesc;
   PBYTE                pBuf;
   DWORD                cbRead;
   ULONG                qop;

   // Receive the encrypted message size first
   //
```

```
   RecvBytes(s, (PBYTE) cbBuf, sizeof (*cbBuf), &cbRead);
   pBuf = (PBYTE) malloc (*cbBuf);
   // Recieve the encrypted message
   //
   RecvBytes(s, pBuf, *cbBuf, &cbRead);
   // Build the security buffers for decryption
   //
   secBufDesc.cBuffers = 2;
   secBufDesc.pBuffers = secBufs;
   secBufDesc.ulVersion = SECBUFFER_VERSION;
   // Keep the encrypted message here
   //
   secBufs[0].BufferType = SECBUFFER_STREAM;
   secBufs[0].pvBuffer = pBuf;
   secBufs[0].cbBuffer = *cbBuf;
   // Buffer to keep the decrypted message
   secBufs[1].BufferType = SECBUFFER_DATA;
   secBufs[1].cbBuffer = 0;
   secBufs[1].pvBuffer = NULL;

   ss = m_pFuncTbl->DecryptMessage(&m_hCtxt,
                                   &secBufDesc,
                                   0,  // no sequence number
                                   &qop);
   if (ss != SEC_E_OK){
      free (pBuf);
      throw CKerberosErr("DecryptMessage failed", ss);
   }
   *cbBuf = secBufs[1].cbBuffer;
   return ((PBYTE) (secBufs[1].pvBuffer));
}
```

The `CKerberosServer` class has other methods to impersonate the client, to stop impersonation, to display security context attributes, and so on. Refer to the source

code on the companion CD-ROM for more information and for the example code on how to use these classes.

## Summary

The Security Support Provider Interface (SSPI) is an application programming interface (API) that applications should use to request security services from security service providers. In Windows 2000, a security support provider is a dynamic link library that supports the SSPI specification and that provides one or more security support packages. Each security support package (SSP) supports one specific security protocol and implements the SSPI functions according to that security protocol.

Applications can use the services of a security package in various ways. Applications can use the SSPI functions directly or through higher-level protocols or APIs, such as RPC, DCOM, WinSock, or WinInet. Nevertheless, one way or another, access to the security services is always through the SSPI layer.

SSPI defines four interface categories for credential management, context management, message exchange, and package management. Note that SSPI does not provide any interfaces for sending and receiving messages.

## References

[MICR00] Microsoft Corporation, "The Security Support Provider Interface," January 2000. (http://www.microsoft.com/TechNet/win2000/win2ksrv/technote/sspi2k.asp)