Simplify enterprise Java authentication with single sign-on

Design secure client/server Java applications that use GSS-API and Kerberos tickets to implement SSO

Faheem Khan

As you add more and more password-protected applications to your organization's computing environment, you add authentication complexity that will burden both developers and users. Most enterprise application integration projects include single sign-on (SSO) functionality, which allows users to log in once to use a range of different applications. In this article, consultant Faheem Khan shows you how to implement SSO on the Java platform. Security is a tricky subject, but you'll see how you can use the GSS-API and Kerberos standards to abstract away some of the difficulty, and implement an SSO-based architecture with relative ease.

Does your enterprise run many coexisting Java applications, each requiring authentication in order to access enterprise resources? If so, you'll probably want to implement single sign-on (SSO) security functionality to make authentication less intrusive for your users. In this article, you'll learn how to implement SSO using Kerberos and the Java Generic Security Services API (GSS-API). First we'll cover what SSO means and illustrate its potential applications. Then we'll explore the sequence of message exchanges that occurs to implement Kerberos-based SSO. Next, we'll briefly introduce the Java GSS-API and the architecture of a typical Java application that accomplishes SSO using GSS. Finally, we'll put the pieces together and provide working code examples to demonstrate how Java developers can implement SSO with GSS Kerberos tickets.

What is single sign-on?

Fundamentally, single sign-on authentication means the sharing of authentication data. For instance, many employees of a warehousing company might need to access enterprise resources (database tables, for example) in order to fulfill their job requirements, with different employees needing different resources depending on their job function. An accounts manager may need to access only the accounts-related database tables, while a sales manager may need to access sales-related database tables. In contrast, the CEO of the company may need to access any table in the enterprise’s database.
Obviously, this enterprise needs an authentication mechanism in place that can determine which employee is trying to access a particular resource. Once the enterprise authentication module knows the identity of the employee, an authorization module within the enterprise implementation can check whether the authenticated user has the necessary privileges to access the resource.

Let's suppose that employees use their usernames and passwords for authentication. The enterprise's authentication module would thus have a database of usernames and passwords. Each incoming request for authentication would be accompanied by a username-password pair, which the authentication module would compare against the pairs in its internal database.

Now, our warehousing company may have several applications running within its scope. Different applications form different modules of the same enterprise. Each application is complete in itself, which means that it has its own user base, as well as several different tiers, including the back-end database, business logic, and a GUI for its users. Enterprise application integration (EAI) is a popular name for projects involving the integration of such independent applications into an enterprise.

One common factor usually marks the authentication process in EAI projects: users of a particular application need to access another application within the enterprise. For example, a sales manager who is using the sales database may need to access the inventory database in order to check the availability of a particular component. How can we enable this type of cross-application authentication?

We have two choices:

- We could duplicate the username and password databases on both applications, thus effectively allowing both applications to process authentication requests for all the employees of our warehousing company. This also means that a user will authenticate separately on the two applications -- in other words, he will enter his username and password while accessing either of the two applications, and the application will perform all the authentication steps. This means that we are not only duplicating the username and password database: we are also duplicating the authentication process overhead. The amount of redundancy in this solution should be very obvious.
- The second choice is to enable sharing of authentication data through single sign-on between the sales and inventory applications. If a user is authenticated at one application, his authentication information is transferred to the second. The second application accepts the authentication information as such without going through all the authentication steps. There is no redundancy in this solution. The only requirement is that the two applications trust each other so that each application accepts authentication data coming from the other.
Normally, SSO is implemented as a separate authentication module. All applications that need to authenticate users rely on the SSO-based authentication module to check the identity of their users. Based on this authentication information, different applications then enforce their own authorization policies.

Hopefully, our warehousing company example has illustrated what SSO looks like from the user’s perspective. The next question, naturally, is: How do we implement SSO? There are several ways in which we can do so. In the next section we'll discuss Kerberos, which offers various security services, including SSO.

**Using Kerberos for SSO**

Kerberos is an Internet Engineering Task Force (IETF) standard that defines a typical key exchange mechanism. Applications can use the Kerberos service to authenticate their users and exchange cryptographic keys with them. Kerberos is based on the idea of tickets. A ticket is just a data structure that wraps a cryptographic key, along with some other bits of information. A key distribution center (KDC) distributes Kerberos tickets to authenticated users. A KDC issues two types of tickets:

- A master ticket, also known as the ticket granting ticket (TGT)
- A service ticket

A KDC first issues a TGT to a client. The client can then request several service tickets against his or her TGT. To explain how TGTs and service tickets work, let's consider the following key exchange scenario:

1. A client sends a message to the KDC requesting the issuance of a TGT. The request is in plain text form (without any encryption), and includes the username of the client, but does not include his password.
2. The KDC issues a TGT to the client. The TGT contains a session key in encrypted form. To encrypt the session key, the KDC uses a key derived from the client's password. This means that only the client can decrypt the TGT and fetch the session key. Therefore, although a client application does not need to know the password to request a TGT, it does require the password to process or use the TGT.
3. The client decrypts the TGT and extracts the session key from it. The client then authors a request for a service ticket. A service ticket is valid only for communication between two parties -- that is, between the client and the other entity (a server, say) with whom the client wants to communicate. No one else can use the service ticket. Therefore, while requesting the service ticket, the client specifies the name of the server with which he plans to use that service ticket. The server should be already registered with the KDC.
4. The KDC authors a service ticket for the server. This ticket contains the client's authentication data and a new cryptographic key, called a sub-session key. The KDC encrypts the service ticket with the secret key of the server (the secret key is a shared secret between the KDC and the server). This means that only the server can decrypt the service ticket.
5. The KDC authors a message and wraps the service ticket inside of it. The KDC also copies the sub-session key inside the message. Notice that the sub-session key is now contained in the message twice: once directly in the message and again inside the service ticket.
6. The KDC encrypts the complete message with the session key from steps 2 and 3. Thus, only the client can decrypt the message and extract the sub-session key as well as the service ticket. But the client cannot decrypt the service ticket -- only the server can. Therefore, no one else can use the service ticket for any purpose. The KDC then sends the message to the client.

7. The client decrypts the message received from the KDC and fetches the sub-session key inside the message as well as the service ticket. It sends the service ticket to the server.

8. The server receives the service ticket and decrypts it to fetch the authentication data of the requesting client as well as the sub-session key. The server then acknowledges the client's request, and a new secure session is established between the client and the server. Both client and server now possess the same sub-session key, which they can use for secure communication with each other.

The client can repeat steps 3 through 8 for another server application. This means that our Kerberos service can be used to share authentication data, and that the same client (which represents a single user) can authenticate with different applications. This effectively enables SSO.

The process we've just described illustrates how any application can use Kerberos for authentication. Now we'll look at how you can specifically use the Java platform to take advantage of Kerberos's functionality.

**The Java Generic Security Services API**

The IETF has defined the Generic Security Services API (GSS-API) as a high-level security API that provides features like confidentiality, message integrity, and authentication. GSS-API can easily work in client-server environments. The IETF has defined GSS-API in a language-independent manner.

The Java GSS-API is the Java language-specific form of the GSS-API. Sun has developed the Java GSS-API through the Java Community Process and has also provided a reference implementation, which comes bundled with version 1.4 of the JDK. See Related topics for a link to the official RFC for GSS-API.

### Setting up a KDC

The Kerberos implementation in GSS does not require a particular KDC implementation. Therefore, all code based on GSS will work with any GSS-compliant KDC. See Related topics for a list of popular KDC implementations.

To test the code in this article, I used Microsoft's KDC implementation, which comes as a service in Windows 2000 Server (and later editions of Windows-based servers). The file j-gss-sso.zip, which contains all the code for this article, also contains the file Setup.txt, which outlines the steps for setting up the Microsoft KDC.

The Microsoft KDC accepts all Windows users as users of the service. This means that the KDC can issue a TGT for any user registered with the Windows network.

This article discusses only the Java GSS-API; therefore, for the sake of simplicity, we will refer to the Java GSS-API as **GSS** in the rest of this article.
GSS aims to provide a high-level abstraction layer on top of different low-level security services. Kerberos is one of the technologies that you can use under the GSS abstraction, though there are many others, like the Simple Public Key Mechanism (SPKM; see Related topics), which we won't discuss in this article. The GSS layer of abstraction allows programmers to develop secure applications without worrying about what mechanism will work on a lower level to provide security services. In this article, we'll focus on the use of Kerberos as the low-level security mechanism working under GSS to achieve SSO.

In the remainder of this section, we'll offer a very simple introduction to GSS; the next section will demonstrate the API in action. We will often use the term communicating entities or peers in the forthcoming discussion. An entity or a peer is an application that communicates using GSS. A requesting peer will act as a client and request a new session to securely communicate with a serving peer. The serving peer will act as a server and accept or reject the request.

GSS works on the idea of a GSSName, a GSSCredential, and a GSSContext. A GSSName identifies you as an individual, like the username that you enter while checking your e-mail. A GSSCredential is something that you present to prove your identity, like the password that you enter while checking your e-mail. A Kerberos service ticket is another example of a GSSCredential. A GSSContext is like a secure session that encapsulates security-related information, such as cryptographic keys. GSS-based applications will use a GSSContext to securely communicate with each other.

Now let's put to work the concepts that we've learned so far.

**GSS client and server applications**

In this section, we will demonstrate an actual implementation of a GSS-based secure Java applications. We will develop two reusable JavaBeans components. One bean acts as a requesting client and requests the initiation of a new GSS session (a GSS session is referred to as a GSS context). The other bean will act as a server, listen for requests, and accept the incoming request from the client; it then establishes the secure context and communicates with the client.

As we are discussing and demonstrating SSO in this article, we'll simplify our application by having the KDC used by the application hosted by a third party. Both the client and server trust this KDC and accept authentication data coming from it. Thus, you'll need a Kerberos implementation running in order to use the sample code in this article, but you can use any GSS-compliant KDC you'd like. (If you're using a recent Windows server platform, you already have access to a suitable Kerberos implementation; see the sidebar entitled "Setting up a KDC" for more details.)

**A JAAS authentication client**

Once you have a KDC service running, you can go ahead and request the KDC to issue a TGT. A requesting GSS client (the one who wants to establish a secure GSS context with a remote serving peer) will ask the KDC to issue a TGT.

However, we have a small problem here. GSS does not contain any method to fetch the username-password pair from a user. Therefore, GSS applications have to rely on other non-GSS mechanisms for login information. We are going to use the Java Authentication and Authorization
Service (JAAS) to allow the requesting client to provide a username and password and acquire the TGT.

Have a look at Listing 1, which shows a class named GSSClient. This class represents the functionality of a GSS client that wants to establish a secure session with a remote GSS server. The GSSClient constructor takes a number of parameters:

1. The name of the client peer
2. The password of the client peer
3. The name of the remote serving peer
4. The address of the remote serving peer
5. The port of the server
6. The Kerberos realm or domain (the domain in which the KDC is running; see Setup.txt for details of how to specify the realm for Microsoft's KDC)
7. The address of the KDC
8. The location path and name of a login configuration file (we'll explain this shortly)
9. The name of a client configuration (which specifies the authentication mechanism that the client wants to use)

Note: We will be referencing Listing 1 repeatedly throughout our discussion of the client, so you may want to keep the window open for guidance.

Note that the main() method simulates a simple application. We have included this method in Listing 1 only to demonstrate the operation of this class by running it from the command line. The main() method reads the parameter values from the command line in the sequence shown above and calls the GSSClient() constructor, passing the parameter values to the constructor.

The GSSClient() constructor stores the parameters coming from the command line in different fields and also sets three system properties itself. The java.security.krb5.realm system property specifies the KDC realm; the java.security.krb5.kdc property specifies the address of the KDC server; and the java.security.auth.login.config property specifies the location path and name of a login configuration file. The GSS framework will use these properties internally.

After instantiating a GSSClient object, the main() method calls the GSSClient.login() method. This method instantiates a LoginContext object, which is part of the JAAS framework. The LoginContext constructor takes two parameters. The first, confName, carries the ninth command-line parameter value; the second is an object of a class named BeanCallBackHandler. Let's look at the use of these two parameters in more detail.

confName and the configuration file
confName carries the name of a JAAS configuration. The ninth command-line parameter (the name of a client configuration) specifies the JAAS configuration that the client wants to use for authentication. The JAAS configuration file, specified by the eighth command-line parameter, contains one or more client configurations, out of which the client can use any one.
A JAAS configuration specifies the mechanism that will be used for authentication. The concept of configuration files allows Java applications to choose an authentication mechanism independent of authentication logic.

JAAS configurations are stored as .conf files. The sample JAAS configuration file in Listing 2 has two configurations. The gssclient configuration looks like the this:

```
GSSClient {
    com.sun.security.auth.module.Krb5LoginModule required;
};
```

This JAAS configuration specifies the name of the Java class com.sun.security.auth.module.Krb5LoginModule. This class is the Kerberos login module in JAAS, and the gssclient configuration uses it for login. Thus, specifying this configuration means that we will use Kerberos as our mechanism for authentication.

### Listing 2. JAAS login configuration for GSSClient and GSSServer

```java
/****
Login.conf
****/
GSSClient{
    com.sun.security.auth.module.Krb5LoginModule required;
};
GSSServer{
    com.sun.security.auth.module.Krb5LoginModule required
    storeKey=true;
};
```

The details of JAAS authentication logic are beyond the scope of this article. If you want to learn more about it, check out the link to the tutorial, "Java security, Part 2: Authentication and authorization," in the Related topics section.

**BeanCallBackHandler**

Now, coming back to our discussion of the client in Listing 1, the second parameter that we have passed along with the LoginContext constructor method invocation call (an instance of the BeanCallBackHandler class) specifies the object that will handle callback during the authentication process. Callback allows Java applications to interact with JAAS implementations during the authentication process. You will normally use callback functionality to pass a username and password to the Kerberos authentication module. Notice that we have passed the username and password in the BeanCallBackHandler constructor call.

Now look at Listing 3, which shows our BeanCallBackHandler class. This class implements an interface named callbackHandler, which is part of the JAAS framework and contains just one method, named handle().

### Listing 3. Handling callback from the JAAS framework

```java
/****
BeanCallbackHandler.java
****/
```
import java.io.*;
import java.security.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;

public class BeanCallbackHandler implements CallbackHandler {
    // Store username and password.
    String name = null;
    String password = null;

    public BeanCallbackHandler(String name, String password)
    {
        this.name = name;
        this.password = password;
    }

    public void handle (Callback[] callbacks) throws
    UnsupportedCallbackException, IOException
    {
        for(int i=0; i<callbacks.length; i++) {
            Callback callBack = callbacks[i];

            // Handles username callback.
            if (callBack instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback)callBack;
                nameCallback.setName(name);
            }

            // Handles password callback.
            else if (callBack instanceof PasswordCallback) {
                PasswordCallback passwordCallback =
                (PasswordCallback)callBack;
                passwordCallback.setPassword(password.toCharArray());
            } else {
                throw new UnsupportedCallbackException(callBack,
                "Call back not supported");
            }
        }
    }
}

The handle() method of the CallbackHandler object will automatically receive control during authentication.

The JAAS framework passes an array of Callback objects to the handle() method of the CallbackHandler instance that we pass as the second parameter value of the LoginContext constructor. The array of Callback objects contains different types of Callback objects; however, we are only interested in two types: the NameCallback and PasswordCallback, both of which extend the basic Callback class.

The NameCallback object is used to provide the username to the JAAS framework, while the PasswordCallback carries the password during authentication. Inside the handle() method in Listing 3, we have simply called the setName() method of the NameCallback object and the setPassword() method of the PasswordCallback object.
To summarize the above discussion on the two parameters that go along with the `LoginContext` constructor call, we can say that we have supplied all the information to the `LoginContext` object that is required for user authentication. So the next step in Listing 1 should be to call the `LoginContext.login()` method, which will perform the actual login process.

If something goes wrong during the authentication process -- if the `password` turns out to be incorrect, for instance -- a `javax.security.auth.login.LoginException` will be thrown. If there's no exception as a result of the login method call, then we can assume that the authentication was successful.

Because we are using Kerberos login, the JAAS framework internally manages all communication with the KDC and fetches the Kerberos TGT, thus hiding all technical details under the high-level, easy-to-use JAAS interface.

Successful authentication will result in loading the Kerberos TGT in the `LoginContext` object. You can call the `getSubject()` method of the `LoginContext` class, which returns an instance of a class named `Subject`. The `Subject` instance wraps the TGT.

We will use this `Subject` class to perform the action for which we have logged in: establishing a secure GSS context. Let's see how to invoke the required action after successful authentication.

The `Subject` class contains a static method named `doAs()`, which takes two parameters. Look at the `Subject.doAs()` method call in Listing 1. The first parameter value is the `Subject` instance that we just obtained after successful authentication. The `doAs()` method will use this authenticated `Subject` to make the authorization decision -- that is, it will use it to judge whether this `Subject` (which we have already authenticated) is authorized to invoke the given operation.

We have used the `Subject` class only to fetch the TGT. Therefore, we have not specified any authorization policy for our `GSSClient`. Any user can run this client. The `GSSClient` doesn't require any security authorization to execute the given operation. However, Web clients such as applets run under a strict security context, which requires security authorization to execute successfully. Setup.txt contains brief instructions on writing an authorization policy for an applet-based GSS client, and we'll look at such a client in more detail later in this article.

The second parameter value of the `doAs()` method call, this, specifies the `GSSClient` object (Listing 1) that we are discussing. This parameter expects an object that exposes a `PrivilegedAction` interface. Notice that our `GSSClient` implements that `PrivilegedAction` interface. We have combined all client-side code in one class, but you can have a separate class that implements the `PrivilegedAction` interface, if you'd like. If you choose to do so, you will instantiate that object and pass it as the second parameter value with the `doAs()` method call.

The `PrivilegedAction` interface contains just one method, named `run()`. If the authorization policy allows the `Subject` to access the `run()` method of the `GSSClient` class, that method will receive control in a separate thread of execution. When the `run()` method receives control, the security context (the TGT, essentially) goes along with it. Our GSS logic will automatically use that security context and fetch the TGT.
Designing a GSS client
We have to perform the following steps in the `run()` method of Listing 1 to establish a GSS session:

1. Instantiate a `GSSManager` object. Notice in Listing 1 that we have called the `getInstance()` static method of the `GSSManager` class, which returns a `GSSManager` object. The `GSSManager` object will at least support the Kerberos mechanism and perhaps some other mechanisms as well. The `GSSManager` class contains methods that a GSS application can call to specify other security mechanisms. But because our focus is on the Kerberos mechanism only, we can simply call the `getInstance()` method to use the Kerberos mechanism. We will not go into the details of other mechanisms. Thus, after instantiating a `GSSManager` object in Listing 1, we have instantiated an `Oid` (Object ID) object named `kerberos`, which identifies the Kerberos mechanism. We will pass on this object wherever we need to say, in essence, "We want to use Kerberos as the underlying technology below the GSS layer."

2. Create a `GSSName` object, which represents a GSS entity. During two-way communication, you can think of a GSS entity as a communicating peer. Therefore, we will actually create two `GSSName` objects: one for the requesting client peer (`clientPeerName`) and another for the remote peer (`remotePeerName`).

3. Create a set of credentials. GSS is a generic security mechanism, so it has to rely on the underlying technology to create these credentials. Because we are using Kerberos, the Kerberos ticket is the actual credential. To get a `GSSCredential` object, we use the `createCredential()` method of the `GSSManager` class. The `createCredential()` method returns an object that exposes the `GSSCredential` interface.

4. Create a secure GSS context, which will be used to establish secure communication between the two communicating peers. The `createContext()` method of the `GSSManager` creates and returns an instance of the `GSSContext` interface. The `GSSContext` object wraps the actual secure context that the GSS client wants to establish with a remote peer using the Kerberos service. Notice in Listing 1 that we have passed the `GSSName` and `GSSCredential` objects that we create in steps 2 and 3 to the `createContext()` method call.

Now we have the `GSSContext` object that wraps the secure context, but the context itself is not yet established. After obtaining a `GSSContext` object, a requesting peer will call its `requestConf()` method. This method causes the application to request confidentiality and data integrity, so any data that the application wants to send to the remote server will be in encrypted form.

After calling the `requestConf()` method in Listing 1, we have declared an array of bytes named `byteToken` and instantiated it to a size of zero bytes. The `byteToken` array will hold the data bytes that the GSS client will send to and receive from the server.

We now call the `initSecContext()` method of the `GSSContext` interface repeatedly in a `while` loop. This method performs the actual exchange of bytes to establish a secure context between the requesting and serving peers.

The `while(!peerContext.isEstablished())` block in Listing 1 performs the actual two-way communication between the GSS client and the server. This block exits only when the secure context is established (or an exception occurs in case something goes wrong).
When the `while` loop is executed for the first time, the `byteToken` array will have no data. The `peerContext.isEstablished()` method will return `false`, as the security context is not yet established.

The first thing that we do inside the `while` loop is to pass on the `byteToken` array to the `initSecContext()` method. This method performs two jobs: it generates the bytes that the client sends to the server, and it also accepts bytes coming from the server. This exchange of bytes carries on until the GSS context is established.

Naturally, when we call the `initSecContext()` method for the first time, we have no bytes to pass on to the method. So we pass on the empty `byteToken` array to the method for the first call. The `initSecContext()` method returns some bytes, which we have stored in the same `byteToken` array. Next, we write the `byteToken` (that we got from the `initSecContext()` method call) to the output stream and flush the stream so that the `byteToken` array is sent to the remote server.

Now we expect that the remote server will send something in response to the bytes that we have sent to it. So, we read the bytes from the input stream, store the bytes in the same `byteToken` array, and then pass the `byteToken` array back to the `initSecContext()` method. This method again returns an array of bytes, which we send to the remote server. This exchange of bytes carries on inside the `while` loop until the secure context is established and the `peerContext.isEstablished()` method returns `true`.

The establishment of a secure context means that the appropriate Kerberos keys have now been made available on both client and server. Both parties can use these keys for secure communication. When a secure context is established, we simply return the `GSSContext` object.

**Using the GSS context**

We will now see how to use the `GSSContext` object in the `main()` method of our `GSSClient` class. After calling the `login()` method, we check to see if the `GSSContext` that the `login()` method has returned is null. If it isn’t, we are sure a secure context is available for secure communication with the remote server.

The `main()` method then checks to see if the remote server has honored the client’s request for confidentiality and message integrity (recall that the client made this request before starting to establish the GSS context). The `getConfState()` method of the `GSSContext` class returns `true` if the server has honored our request.

The `main()` method can now send whatever data it wants to the remote server. We have written a `sendMessage()` helper method to send data to the server. Notice in this method that the `wrap()` method of the `GSSContext` class takes a byte array of data and returns a byte array. The `sendMessage()` method supplies plain text data to the `wrap()` method and gets an encrypted byte array in response. We can now send the encrypted byte array to the remote server by writing it on the output stream.
The `sendMessage()` method then listens for incoming data from the server. When we receive some data from the server, we can pass on the incoming data to the `unwrap()` method of the `GSSContext` class, which returns the plain text form of the data.

Our `main()` and `sendMessage()` methods are very simple examples of application-specific logic for data exchange over a secure GSS sessions. You can build your own application logic using the same concept.

### A GSS server application

We have seen how our GSS client application works. Now let's build a server to interact with it.

Have a look at Listing 4, which shows the code for a `gssserver` class. (Again, you'll want to keep this listing window open as we discuss the server.) The `startServer()` method of `gssServer` performs the same functions that we have already discussed while explaining the `login()` method of the `GSSClient` class.

Now, inside the `run()` method of the `gssServer` class, we have created a `GSSManager` and a `GSSName` object representing the server. So far, there's hardly any difference between the client- and server-side code. But look a bit closer and notice that the server creates only one `GSSName` object, which represents the server. After creating this `GSSName`, the server calls the `createCredential()` method to load its credentials in a `GSSCredential` object.

The next step is to call the `createContext()` object to create a new GSS context. This call to the `createContext()` method is different from the `createContext()` call that we made in the GSS client application. This time, the `createContext()` method takes just one parameter: the server's credential. This means that this server-side context is not between two parties. It is like an open-ended connection, where only one of the two communicating parties is decided.

Next, we create input and output streams for communication, and then enter into a `while` loop that will keep on looping until a secure context is established with a requesting client. Inside this loop, we wait for the requesting client to send a connection establishment request. When we receive some data on the input stream inside the `while` loop, we read the data into a byte array and provide the byte array token to the `acceptSecContext()` method of the `GSSContext` class. The `acceptSecContext()` method returns some data bytes to us, which we send back on the output stream.

The `initSecContext()` and `acceptSecContext()` methods of the `GSSContext` class work in combination. We demonstrated the use of the `initSecContext()` method while discussing our GSS client application. The `initSecContext()` method produces the initial bytes that a GSS client sends to a GSS server application. The `acceptSecContext()` method accepts these incoming bytes and generates its own byte array, which we send back to the client. This exchange of bytes continues until a secure GSS context is established.

Therefore, GSS handles all communication as byte array tokens. You can use any type of transport service to convey the array of bytes from the client to the server and back. GSS is indifferent to the transport facility you use for data transmission.
The process of establishing a secure session concludes with the authentication of the requesting client. You can call the `getSrcName()` method of the `GSSContext` class to fetch the `GSSName` of the authenticated client. On the other hand, the `getTargName()` method of the `GSSContext` class returns the `GSSName` of the server that accepted the remote client's request.

After the `while (!context.isEstablished())` loop returns in Listing 4, the `run()` method waits for communication from the client. It keeps on listening for incoming data, and when it receives any incoming byte string, it hands that string over to the `unwrap()` method of the `gsscontext` class. The `unwrap()` method will return the plain text form of the message from the client.

Just for the sake of demonstration, in our sample code we pass a message answer string back to the requesting client. Application developers can implement their own application logic to send their application-specific data back to the requesting client.

We have demonstrated the basic forms of the GSS client and server-side application development. You can put our sample applications to use in several different scenarios. For example, you could use our GSS client as part of a JSP or a JFC application. One interesting scenario would be the use of our GSS client as part of a Java applet running inside a browser, which we'll cover in some detail next.

**GSS in a browser**

Have a look at Listing 5, which demonstrates how an applet can use the GSS client in Listing 1 to establish secure communication with the GSS server in Listing 4.

The applet will run in an HTML page such as the one shown in Listing 6:

**Listing 6. An HTML page that uses a GSS applet**

```html
<!-- E-Commerce Login.html
-->

<HTML>
<HEAD>
<TITLE>E-Commerce Login... </TITLE>
</HEAD>
<BODY>
<p align="center">
<table bgcolor="Gray">
<tr>
<td align="center">
<b>E-Commerce Site Login Page </b>
</td>
</tr>
</table>
</td>
</tr>
<tr>
<td>
<Applet 
CODE="GSSClientApplet.class"
archive="GSSClientApplet.jar"
name="GSSClientApplet"
width="500" height="280">
</Applet>
</td>
</tr>
</body>
</HTML>
```
Let's assume that this applet is running on the main page of an e-commerce Web site. Figure 1 shows what it would look like in action. There are two text entry fields, three buttons, and a text area. Each of the three buttons corresponds to the server-side implementation of a partner of the e-commerce Web site.

**Figure 1. An HTML page showing GSS applet usage**

![E-Commerce Site Login Page](image)

A customer of the e-commerce Web site can authenticate himself with any of the site's partners. The customer will enter his username and password in the text fields of the applet, and press the Login button corresponding to the partner site with which he wants to authenticate. The button event handlers simply provide the required parameters to the `GSSClient` constructor. The rest of the work is our GSS client's job, which we have already explained.

**Summary**

We have discussed single sign-on, Kerberos, and GSS in this article. Applications can use SSO to share authentication data with each other. Kerberos offers a mechanism for key management and exchange. GSS is a high-level security API that works on top of different security services.

As you've seen here, you can use GSS and Kerberos to build SSO solutions for the Java platform. While the concepts may seem complex at first, the layers of abstraction afforded by GSS should allow you to implement SSO fairly easily in your Java platform-based enterprise. I hope that you can use the code outlined here as a jumping-off point for your own projects.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-gss-sso.zip</td>
<td>28KB</td>
</tr>
</tbody>
</table>
Related topics

- Read RFC 1510, which officially defines Kerberos.
- RFC 1508 and RFC 1964 define GSS.
- Visit the GSS page at Sun's Web site.
- Check out this KDC implementation from MIT.
- Heimdal provide free Kerberos implementations.
- "Java security, Part 2: Authentication and authorization," by Brad Rubin (developerWorks, July 2002), is an excellent tutorial on JAAS.
- Refer to RFC 2025 for more details of SPKM.

© Copyright IBM Corporation 2003
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)