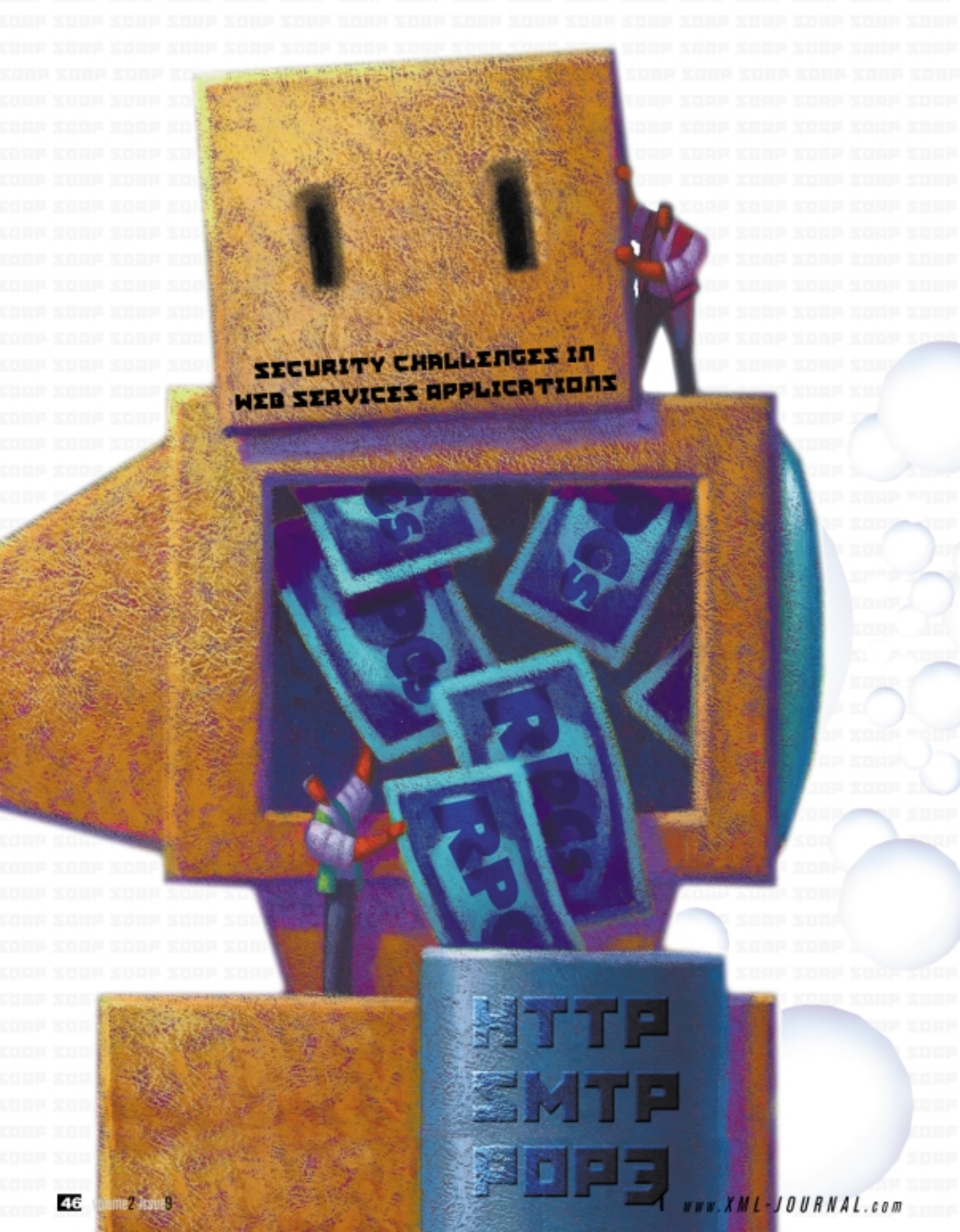
SECURITY CHALLENGES IN
WEB SERVICES APPLICATIONS
HTTP
SMTP
POP3

WEB SERVICES SECURITY

USING PLUGGABLE PROVIDERS FOR REALM-BASED AUTHENTICATION AND AUTHORIZATION

THE WEB SERVICES MODEL IS RAPIDLY GROWING AS AN OPTION FOR DEPLOYING REUSABLE, COMPONENT-BASED FUNCTIONALITY OVER THE INTERNET. EXPOSING THESE SERVICES IS THE SIMPLE OBJECT ACCESS PROTOCOL, WHICH DEFINES A LIGHTWEIGHT MECHANISM FOR INVOKING THESE SERVICES OR TRANSACTIONS.

The SOAP specification defines a set of rules for using XML to represent such Remote Procedure Calls (RPCs). These RPCs can ride on various transport protocols, such as HTTP, SMTP, and POP3. However, the current SOAP specification (version 1.1) doesn't define all the features developers might expect to find in a traditional distributed object protocol, such as security. This article introduces you to the pluggable provider for Apache SOAP, and how a provider can be implemented to enable user-level security. (Source code for this article can be found on the ***XML-J*** Web site, www.sys-con.com/xml/sourcec.cfm.)

Why Do We Need User-Level Security for Web Services?

At first glance it may seem that SOAP is rather incomplete – why create a protocol that intentionally omits one of the prime facets of distributed computing? Simply, SOAP really is incomplete. But this is by design. Security mechanisms are native to the underlying transport protocol used by SOAP. Because the SOAP specification doesn't currently address issues related to security, we can leverage the built-in security features of the protocols that are used to deliver SOAP messages.

Since most Web services will rely on the HTTP protocol, we can use any of its various security implementations. Web servers, firewalls, and the like can use the information provided by HTTP to validate usernames and passwords (authentication), and grant or deny their access to resources based on the caller's identity (authorization). With SOAP on top of HTTP, we can pass username and password information in the HTTP header. The HTTP 1.1 Specification defines several challenge–response authentication mechanisms, which can be used by a server to challenge a client request and by a client to provide authentication information. The general framework for access authentication, and the specification of "basic" and "digest" authentication, are defined in "HTTP Authentication: Basic and Digest Access Authentication." Ultimately, a future SOAP specification should provide support for many of the available standard, transport-layer security mechanisms, such as basic, digest, and cryptographic messages over SSL.

The server that's invoking the Web service should perform authentication and authorization, as opposed to relying on the Web server or other hardware. In this fashion we'll have a robust framework through which we expose our services. These issues are especially important if you deal with sensitive information, or wish to charge a fee for your Web services.

WRITTEN BY MARK A. RICHMAN

Apache SOAP and the Pluggable Provider

The Apache SOAP project implements a SOAP Web Services framework using Java. A provider is the bridge between the SOAP engine and the service being invoked. The provider is responsible for:

- Locating, loading, and invoking the service
- Converting the result from the service into a SOAP envelope
- Implementing specialized functionality outside the scope of the SOAP specification

Here, our specialized functionality will be security. By default the SOAP engine will use the RPCJavaProvider for RPC services, and the MsgJavaProvider for Message services. The RPCJavaProvider simply loads the appropriate class, invokes the desired method, and converts any result into a SOAP envelope. Our pluggable provider will use the JDBC interface to authenticate and authorize users against the SOAP service before invoking it.

The pluggable provider demonstrated in this article will implement authentication and authorization through a JDBC interface to a database of user profiles and Web services. The user profiles contain username/password pairs for authentication, and a mapping of usernames to Web services for authorization. The use of a SQL data source will be a familiar paradigm for Java developers who already make use of database-driven security.

Setting Up the Server

Both a servlet engine and Web server are required to host the Apache SOAP framework. To operate correctly this framework requires the Java 2 Platform, Standard Edition (J2SE), which includes JDK 1.3. The Apache SOAP distribution includes installation instructions for various Web servers, including Apache Tomcat, BEA WebLogic Server, IBM WebSphere, and Allaire JRun.

For this article we stick with freely available, open-source products. The examples in this article run unmodified on Apache Tomcat 3.2.1. You'll also need Apache Xerces (Java) version 1.2 or higher. Apache SOAP requires that xerces.jar is placed at the front of your classpath. This will resolve conflicts with other XML parsers you may have, including the parser that comes with Tomcat. Apache SOAP requires DOM Level 2 namespaces. You'll also need JavaMail (mail.jar) and JavaBeans Activation Framework (activation.jar) in your classpath.

Some samples that come with Apache SOAP, such as the Calculator sample, require the Bean Scripting Framework and JavaScript. If you'd like to try these as well, place bsf.jar and js.jar in your classpath. Many servers let you drop these archives into a /lib folder without an explicit reference in a configuration file. Of course, you need to add soap.jar as well. In a nutshell, here's how to hook it all up.

Some systems require you to set the TOMCAT_HOME environment variable. On Windows 2000 I set mine to C:\jakarta-tomcat-3.2.1.

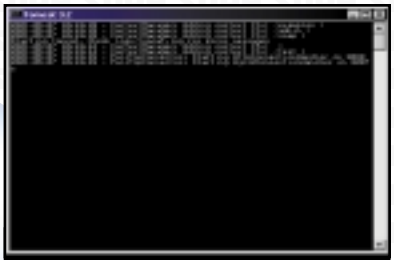

FIGURE 1 Apache Tomcat console (Windows 2000)

FIGURE 2 Apache Tomcat default home page

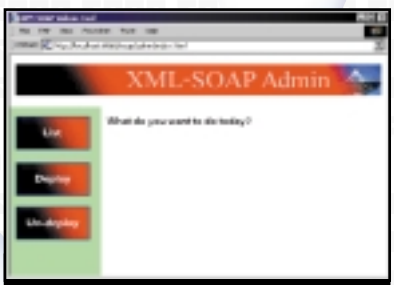

FIGURE 3 Apache SOAP Web admin

FIGURE 4 Apache SOAP Web service deployment

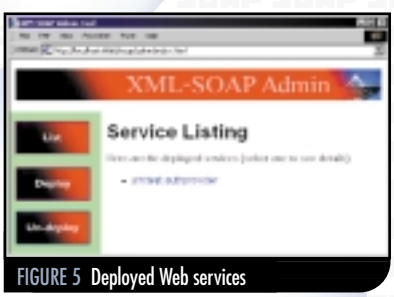

FIGURE 5 Deployed Web services

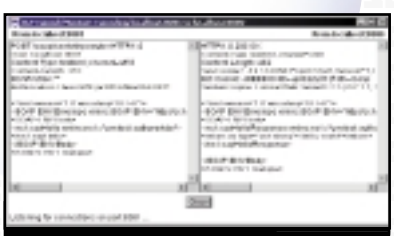

FIGURE 6 TcpTunnelGui

Set Tomcat's classpath with xerces.jar at the front. In bin/tomcat.bat find the :setClasspath section and modify it to look like this:

```
:setClasspath
set CP=<path to Xerces-J>\lib\xerces.jar;%TOMCAT_HOME%\classes
```

On UNIX systems find the export CLASSPATH statement in bin/tomcat.sh, and add this line right before it:

```
CLASSPATH=<path to Xerces-J>/lib/xerces.jar:${CLASSPATH}
```

Next, make sure Tomcat is running on port 8080. Verify that server.xml contains a section like this:

```
<Connector className="org.apache.
  tomcat.service.PoolTcpConnector">
    <Parameter name="handler"
value="org.apache.tomcat.service.
  http.HttpConnectionHandler"/>
    <Parameter name="port" value="8080"/>
</Connector>
```

Finally, add a server context for Apache SOAP. Inside the <ContextManager> tag add the following section:

```
<Context path="/soap"
    docBase="webapps/soap"
    crossContext="true"
    debug="0"
    reloadable="true"
    trusted="false" >
</Context>
```

If you set up everything correctly, running bin/startup should display the console shown in Figure 1.

Point your browser to http://localhost:8080/ to verify that everything is up and running (see Figure 2).

Naturally, running bin/shutdown will stop Tomcat.

Setting Up the Database

For authentication and authorization we'll use a SQL database. Specifically, I've chosen MySQL. This type of database-driven authentication should be familiar if you've set up any type of security on a Web site. Our schema is relatively simple: a table to store our principals and credentials (usernames and passwords), a table to store our Web service identifiers (URNs), and a table to map the principals and Web services. We use a many-to-many relationship between principals and Web services since we want to relate many users to many Web services.

To decrease the duplication of data, we'll create a table called principal_webservice_map that consists of nothing but primary keys from the two other tables. Here we adhere to the Fourth Normal Form (4NF). This form is often overlooked, but it's important when dealing with many-to-many relations. This means that any given relation may not contain

more than one multivalued attribute. This concept is beyond the scope of this article, so I leave it as an exercise for the reader to discover its implications.

For the DDL example we'll use MySQL's syntax. This DDL can easily be ported to any SQL database with minor changes. To create our database using MySQL, start up the server (mysqld) and execute the following command from the command prompt:

```
C:\mysql\bin>mysql < createdb.sql
```

The DDL for this example is shown in Listing 1.

The next step in setting up our database is to configure our JDBC data source. We can use either a native driver or the JDBC–ODBC bridge driver. I've used the bridge driver (sun.jdbc.odbc-.JdbcOdbcDriver) to make our provider a little more portable. Naturally, you would want to use a native or Type 4 driver in an optimal solution. When using a Type 4 driver, remember to add it to the classpath of your server. In the Windows 2000 control panel, I've created a system DSN called *authprovider* using the MySQL ODBC driver.

That's all there is to setting up the SQL database. Let's move on to writing our pluggable provider's implementation.

Writing a Pluggable Provider

All Apache SOAP providers need to implement the org.apache.soap.util.-Provider interface. We implement the provider using two methods: locate() and invoke(). Let's cover these steps at a high level initially. First, the RPCRouterServlet loads the provider as specified in the deployment descriptor for your service (see Listing 2). We'll cover deployment later in this article.

The locate() method will be called in order to allow the provider a chance to verify that the service exists and is available to process the request. If an error occurs, this method should throw a SOAPException. After a successful call to locate it, the SOAP engine will then call invoke() to actually call the service. You may notice that the invoke method doesn't have any parameters explicitly pertaining to the Web service. These parameters are extracted by using the SOAP envelope that's passed to the RPCRouter-Servlet via HTTP.

The invoke method is also responsible for converting any response from the service into a SOAP envelope and placing it in the res parameter (a SOAPContext). This response is then sent back to the client via HTTP.

Now, let's walk through our pluggable provider's implementation.

In our locate() method we need to grab the HTTP Basic Authentication information from the headers. This can be obtained from the HttpServletRequest object:

```
HttpServletRequest req = (HttpServlet-
Request)reqContext.getProperty-
(Constants.BAG_HTTPSERVLETREQUEST);
```

HTTP encodes the authentication header using Base64 encoding. This string is of the form:

```
Authorization: Basic username:password
```

where username:password has been encoded in Base64. Now let's extract the authentication information:

```
String authorization =
  req.getHeader("authorization");
authorization = authorization.substring
  (authorization.indexOf(" "));
byte[] bytes = Base64.decode(authorization);
String decoded = new String(bytes);
int i = decoded.indexOf(":");
String username = decoded.substring(0,i);
String password = decoded.substring
  (i+1,decoded.length());
```

Next, we need to connect to our database so we can perform the authentication. I've created a separate class called AuthProviderDB to manage the database connection and abstract out the authentication and authorization queries. We'll use this class's methods, isAuthenticated() and isAuthorized(), to evaluate the caller's credentials. Our first query validates the user's password. If this function returns true we perform our second query, which evaluates the user's authorization to the Web service requested (see Listing 3).

Notice I use a PreparedStatement in the second function. The database query can be done either way; I just wanted to demonstrate both techniques. Usually, the latter is preferable, since many database servers can precompile or cache this type of query, improving performance. If this function returns true, we can grant the caller access to the Web service. Next, the AuthProvider class resolves the Web service, referred to as the *target object*. Apache SOAP's Service Manager performs this lookup.

```
ServletConfig  config  = servlet.get
  ServletConfig();
ServletContext context = config.get-
ServletContext();
ServiceManager serviceManager =
    ServerHTTPUtils.getServiceManager-
FromContext(context);
...
```

```
Object targetObject = ServerHTTPUtils.getTargetObject(serviceManager,
    dd, targetObjectURI, servlet, session, context);
```

Once our target object has been located, we can invoke the service:

```
public void invoke(SOAPContext reqContext, SOAPContext resContext)
throws SOAPException {
    ...
    Response resp = RPCRouter.invoke(dd, call, targetObject, resContext);
    Envelope env = resp.buildEnvelope();
    StringWriter  sw = new StringWriter();
    env.marshall(sw, call.getSOAPMappingRegistry(), resContext);
    resContext.setRootPart(sw.toString(), Constants.HEADERVAL_
      CONTENT_TYPE_UTF8);
    ...
}
```

Here, the Provider class invokes the Web service as identified by the target object and builds a SOAP Response envelope to encapsulate our return value, encoded in SOAP's XML envelope format.

Writing the Web Service

Coding the actual Web service is the simple part of this exercise. Here, we'll write a Web service that exposes one method, sayHello(), that does just what you may have guessed, says "Hello, world!". This is implemented as a method that returns a string object. Listing 4 provides the code for the implementation class.

Once compiled, you can add HelloWorld.class to your classpath, either under the com/empsoft/soap/services/ path, or in a JAR file containing this package structure. I prefer to use a JAR file so I can package my Web service, pluggable provider, and database classes together. We create our JAR file as follows:

```
jar cvf AuthProvider.jar com/empsoft/soap/
  services/*.class com/empsoft/soap/
  providers/*.class
```

The resulting JAR file will be structured as such:

```
META-INF/MANIFEST.MF
com/empsoft/soap/providers/AuthProvider.class
com/empsoft/soap/providers/AuthProviderDB.class
com/empsoft/soap/services/HelloWorld.class
```

Dropping this JAR file into Tomcat's /lib folder automatically adds it to the classpath on startup.

Deploying Web Services Using Pluggable Providers

Web services can be deployed via the command line or through the Apache SOAP Web Admin. Make sure your Web server is running and launch http://localhost:8080/soap/ from your browser. Click on "Run the admin client" when you see the welcome screen. You should then see the screen shown in Figure 3.

This screen shows the three options to list, deploy, and undeploy your Web services. Since we don't have any Web services deployed yet, go ahead and click "Deploy". You'll be presented with the screen shown in Figure 4.

Let's walk through each property in the Deploy screen, as explained in the Apache SOAP documentation:

- ***ID:*** Uniquely identifies the service to clients. It must be unique among the deployed services, and be encoded as a URI. We commonly use the format "urn:UniqueServiceID". It corresponds to the target object ID in the terminology of the SOAP specification. I chose urn:test-auth-provider, but you can pick any locally unique string you like.
- ***Scope:*** Defines the lifetime of the object serving the invocation request. This corresponds to the scope attribute of the <jsp:useBean> tag in JSP. This tag can have one of the following values:
  - –***Page:*** Indicates that the object is to be available until the target JSP page (in this case the rpcrouter.jsp) sends a response back or the request is forwarded to another page (if you're using the standard deployment mechanism, this is unlikely to happen). Since we don't need a new instance of the HelloWorld service created for every SOAP request, we'll use the "Application" scope.
  - –***Request:*** The object is available for the duration of the request, regardless of forwarding.
  - –***Session:*** The object is available for the duration of the session.
  - –***Application:*** Any page within the application may access the object. In particular, successive service invocations belonging to different sessions will share the same instance of the object. Note that the value of this attribute can have important security implications. The page and request scopes ensure the isolation of successive calls. On the other extreme, application scope implies that all service objects are shared among different users of the SOAP server. Since we're serving the same static content from our Web service to every client, it's safe to choose this option.
- ***Method list:*** Defines the names of the method that can be invoked on this service object. We have one method, sayHello.
- ***Provider type:*** Indicates whether the service is implemented using Java or a scripting language. We're obviously using Java, and our provider's full class name is com.empsoft.soap.services.HelloWorld.
- ***For Java services, Provider class:*** Fully specified class name of the target object servicing the request. Our pluggable provider is called com.empsoft.soap.providers.AuthProvider.

We can skip the rest of the options for the purposes of this article. Once you've entered the fields, scroll down and click the "Deploy" button at the bottom of the window. You should see a screen that indicates that the service has been deployed. If you click on the List button, you'll see the URN of your Web service listed. Click on its link and you should see the information shown in Figure 5.

When deploying a Web service from the command prompt, we use a deployment descriptor in XML format. The pluggable provider is simply placed as the full classname in the deployment descriptor's type attribute (see Listing 5).

Use the deployment descriptor in Listing 5 to fully describe the service. To deploy the Web service from the command line use the following:

```
java org.apache.soap.server.ServiceManagerClient http://local-
host:8080/soap/servlet/rpcrouter deploy DeploymentDescriptor.xml
```

The ServiceManagerClient is a SOAP client that's provided with the distribution that communicates with the Service Manager. Your Web service will be configured and registered by the Service Manager automatically.

Writing the SOAP Client

Invoking a Web service using the Apache SOAP client API is relatively straightforward. We invoke services using the Call object (org.apache.soap.rpc.Call), which is configured with the endpoint's URL, the target object's URI, the method name, and any parameters. The URL we bind to will be driven from the command line for flexibility. In this example the URL will be http://localhost:8080/soap/servlet/rpcrouter. This URL is bound to org.apache.soap.server.http.RPCRouterServlet on the server (see /webapps/soap/WEB-INF/web.xml). We also construct a SOAPHTTPConnection object to capture the HTTP basic authentication information.

The URI of the method call element is used as the object ID on the remote side. Since our sayHello service takes no parameters, our params object is empty. We can now call the invoke() method of our call object, which returns a response object. If any exceptions were thrown

on the server, the fault object will contain those details. The resp object will encapsulate the SOAP response. Calling the getValue() will display the string "Hello, world!" (see Listing 6).

To execute the AuthProviderClient, run the following at the command prompt:

```
java -cp %CLASSPATH% AuthProviderClient
http://localhost:8080/soap/servlet
/rpcrouter
```

If all goes well, the client will be authenticated against the test-auth-provider Web service. You should also see some informational text in the Web server's console. Now, test out what happens when you use the wrong password, or delete the reference in the principal_web-service_map table in the database. There's a Web service called urn:soap-unauthorized in the database for which no user has permissions. Try using this as the parameter for call.setTargetObjectURI(). Bad username/password combinations generate the fault string "Bad password," and authorization failures generate a "User not authorized" fault. I'll leave this as an exercise for the reader to explore the strength of this mechanism.

Using TcpTunnelGui

Apache SOAP also includes a trace utility that can be used to monitor all SOAP communications over HTTP between the client and the server. The TcpTunnelGui application listens on a given port, forwarding all traffic to the destination host and port (see Figure 6). When using the TcpTunnelGui make sure you point your SOAP client to the listening port of this tool, rather than that of the server. This provides a useful tool for debugging your SOAP applications.

Configure the AuthProviderClient at the command prompt to point to port 8081, instead of port 8080. To launch the TcpTunnelGui application, execute the following at the command prompt:

```
java -cp %CLASSPATH%
org.apache.soap.util.net.TcpTunnelGui 8081
localhost 8080
```

As you can see, the HTTP Basic Authentication information as well as the server's response is displayed in the headers.

Conclusion

Web services is a powerful new way to implement distributed applications. However, being in its infancy, the technology presents some challenges. Once security is formally addressed by the SOAP specification, acceptance of Web services will grow more rapidly. Until then, complete security solutions will remain homegrown. Issues such as security and privacy will become increasingly important as Web services applications are distributed over the public internet. In future articles, we'll explore other Web services topics.

Author Bio

Mark A. Richman has over 10 years' experience as an independent consultant and software developer. He specializes in large-scale distributed Web applications. Mark holds a BS in computer science.

MARK@EMPSOFT.COM

Listing 1 DDL for MySQL (createdb.sql)

```
CREATE DATABASE IF NOT EXISTS authprovider;

USE authprovider;

CREATE TABLE principal (
  id int(11) NOT NULL auto_increment,
  username varchar(32) NOT NULL default '',
  password varchar(32) NOT NULL default '',
  PRIMARY KEY (id),
  UNIQUE KEY username (username)
) TYPE=MyISAM;

INSERT INTO principal VALUES (1,'markr','Jsdh8qSD');
INSERT INTO principal VALUES (2,'johns','iJ823hdV');
INSERT INTO principal VALUES (3,'linust','js89PndVs');
INSERT INTO principal VALUES (4,'larryw','58Jnfjh4g');

CREATE TABLE principal_webservice_map (
  p_id int(11) NOT NULL default '0',
  ws_id int(11) NOT NULL default '0',
  PRIMARY KEY  (p_id,ws_id)
) TYPE=MyISAM;

INSERT INTO principal_webservice_map VALUES (1,1);
INSERT INTO principal_webservice_map VALUES (1,2);
INSERT INTO principal_webservice_map VALUES (1,3);
INSERT INTO principal_webservice_map VALUES (1,4);
INSERT INTO principal_webservice_map VALUES (1,5);
```

```
INSERT INTO principal_webservice_map VALUES (1,6);
INSERT INTO principal_webservice_map VALUES (1,7);
INSERT INTO principal_webservice_map VALUES (1,8);
INSERT INTO principal_webservice_map VALUES (1,9);
INSERT INTO principal_webservice_map VALUES (1,10);
INSERT INTO principal_webservice_map VALUES (1,11);
INSERT INTO principal_webservice_map VALUES (2,2);
INSERT INTO principal_webservice_map VALUES (2,4);
INSERT INTO principal_webservice_map VALUES (2,7);
INSERT INTO principal_webservice_map VALUES (2,11);
INSERT INTO principal_webservice_map VALUES (3,4);
INSERT INTO principal_webservice_map VALUES (3,5);
INSERT INTO principal_webservice_map VALUES (3,9);
INSERT INTO principal_webservice_map VALUES (4,1);
INSERT INTO principal_webservice_map VALUES (4,2);
INSERT INTO principal_webservice_map VALUES (4,4);
INSERT INTO principal_webservice_map VALUES (4,7);
INSERT INTO principal_webservice_map VALUES (4,11);

CREATE TABLE webservice (
  id int(11) NOT NULL auto_increment,
  uri varchar(64) NOT NULL default '',
  PRIMARY KEY  (id,uri)
) TYPE=MyISAM;

INSERT INTO webservice VALUES (1,'urn:test-authprovider');
INSERT INTO webservice VALUES (2,'urn:AddressFetcher');
INSERT INTO webservice VALUES (3,'urn:AddressFetcher2');
INSERT INTO webservice VALUES (4,'urn:xml-soap-demo-calcu-
lator');
INSERT INTO webservice VALUES (5,'urn:sum-COM');
INSERT INTO webservice VALUES (6,'urn:adder-COM');
INSERT INTO webservice VALUES (7,'urn:po-processor');
INSERT INTO webservice VALUES (8,'urn:mimetestprocessor');
INSERT INTO webservice VALUES (9,'urn:mimetest');
INSERT INTO webservice VALUES (10,'urn:xmltoday-delayed-
quotes');
INSERT INTO webservice VALUES (11,'urn:ejbhello');
INSERT INTO webservice VALUES (12,'urn:soap-unauthorized');
```

LISTING 2 Provider.java

```
package org.apache.soap.util;

import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.server.*;

public interface Provider {
    public void locate(DeploymentDescriptor dd,
                    Envelope              env,
                    Call                  call,
                    String                methodName,
                    String                targetObjectURI,
                    SOAPContext           reqContext)
                throws SOAPException;
    public void invoke(SOAPContext req, SOAPContext res)
      throws SOAPException;
}
```

LISTING 3 AuthProviderDB.java

```
public boolean isAuthenticated(String username, String
  password) {
 try  {
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT password FROM " +
   "principal WHERE username = '" + username + "'");
  if (rs.next()) {
   if(rs.getString("password").equals(password)) {
    return true;
   }
  }
  return false;
 }
 catch (SQLException e)  {
  return false;
 }
  }
```

```
public boolean isAuthorized(String username, String uri) {
    try  {
  PreparedStatement pstmt =
   con.prepareStatement("SELECT p.username FROM " +
   "principal_webservice_map pwm, " +
   "principal p, webservice ws " +
   "WHERE pwm.p_id = p.id " +
   "AND pwm.ws_id = ws.id " +
   "AND p.username = ? AND ws.uri = ?");

  pstmt.setString(1,username);
  pstmt.setString(2,uri);

  ResultSet rs = pstmt.executeQuery();
  if (rs.next()) {
   if(rs.getString("username").equals(username)) {
    return true;
   }
  }
  return false;
 }
 catch (SQLException e)  {
  System.err.println("SQLException: " + e.getMessage());
  return false;
 }
  }
```

LISTING 4 HelloWorld.java

```
package com.empsoft.soap.services;

public class HelloWorld {
    public String sayHello() {
        return("Hello, world!");
    }
}
```

LISTING 5 DeploymentDescriptor.xml

```
<isd:service xmlns:isd="http://xml.apache.org/xml-
  soap/deployment"
               id="urn:com-empsoft-soap-services-
                 HelloWorld">
  <isd:provider type="com.empsoft.soap.providers.AuthProvider"
                  scope="Application"
                  methods="sayHello">
    <isd:java class="com.empsoft.soap.services.
      HelloWorld"/>
  </isd:provider>
</isd:service>
```

LISTING 6 AuthProviderClient.java

```
String username = "markr";
String password = "Jsdh8qSD";

URL url = new URL(args[0]); // SOAP listener's URL from the
  command line
Call call = new Call();
call.setTargetObjectURI("urn:test-authprovider");
call.setMethodName("sayHello");

SOAPHTTPConnection hc = new SOAPHTTPConnection();
hc.setUserName(username);
hc.setPassword(password);
call.setSOAPTransport(hc);

Vector params = new Vector();
call.setParams(params);
Response resp = call.invoke(url, "");

if (resp.generatedFault()) {
    Fault fault = resp.getFault();
    System.out.println ("Ouch, the call failed: ");
    System.out.println ("Fault Code   = " +
      fault.getFaultCode());
    System.out.println ("Fault String = " +
   fault.getFaultString());
} else {
    Parameter result = resp.getReturnValue();
    System.out.println(result.getValue());
}
```

DOWNLOAD THE CODE @ www.xml-journal.com