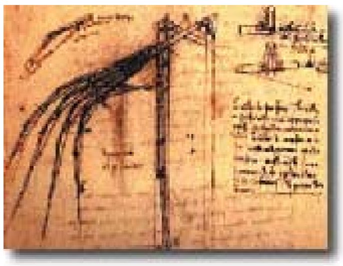
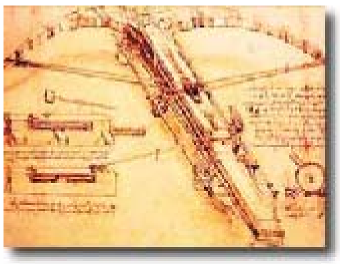
The natural choice for information security solutions





A Corsaire Guide:

Assessing Java Clients with the BeanShell

Author	Stephen de Vries
Document Reference	Assessing Java Clients with the BeanShell v1.0.doc
Document Revision	1.0
Date	15 June 2006

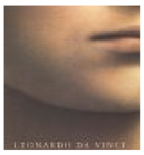


Table of Contents

1. Introduction

Assessing the security of Java applications, and particularly client-server applications, can be a tedious process of modifying the code, compiling, deploying, testing and repeat. This becomes even more difficult when the source code to the application is not available. What we require is an easy means of interacting with the internals of an application during execution without recompiling the code.

Enter the BeanShell (http://www.beanshell.org), which provides an interpreted, scripting environment that can plug in to any Java application or applet and allows users to inspect and manipulate objects dynamically. This paper demonstrates a technique for using the BeanShell to assess the security of a typical Java client-server application.

As an example we will use the Jeti Jabber client from http://jeti.sourceforge.net. This client differs from those that you are likely to encounter on an assessment in two important ways:

- it is open source; and
- the class files are not obfuscated.

Where source code is readily available, performing a source code analysis would typically be the recommended approach. However, inserting a hook to the BeanShell could still save a lot of time in the assessment, since it is a lot easier to work in an interactive shell environment than to get stuck in the- modify source, debug, compile, deploy- cycle.

Commercial applications written in Java are usually obfuscated to prevent attackers or copyright infringers from re-engineering the source. Obfuscators make it more difficult to decompile the byte code into source code – but they do not make it impossible. For the purposes of this demonstration, we will assume that the client is both closed source and has been obfuscated.

2. Obtain the Java Byte Code

The Jeti Jabber applet is loaded from the URL: http://jeti.jabberstudio.org/applet/jeti.html which contains the APPLET tag:

```
<APPLET
archive="applet.jar,plugins/alertwindow.jar,plugins/filetransfer.jar,plugins/vcard.jar,pl
ugins/titlescroller.jar
,plugins/emoticons.jar,plugins/groupchat.jar,plugins/sound.jar,plugins/xhtml.jar,plugins/
keyboardflash.jar,
,plugins/links.jar,plugins/metaltheme.jar,plugins/search.jar,plugins/xmppuri.jar"
CODE="nu.fw.jeti.applet.Jeti.class" width="200" height="400">
<PARAM NAME=EXITPAGE VALUE="javascript:window.close();">
```

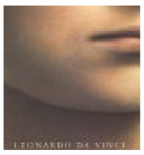
Most of the listed jar files are plug-ins, which do not provide core functionality. The jar file that contains the interesting bits is probably applet.jar.

Download this file and unpack it with:

```
jar –xvf applet.jar
```

3. Decompile the Classes

In order to understand how the application works and where the likely threats are, we should decompile the classes and inspect the generated source. Decompiling does not always work as expected and it will more than likely require some analysis to understand certain parts of the source.



Nevertheless, important information, like the fields, methods and classes, used by an application should be readily apparent. If an obfuscation tool has been used, this interpretation will be more difficult and time consuming – but far from impossible.

A number of free Java decompilers are available, including:

- *JAD* – One of the more popular decompilers but a little dated. There are many GUI decompilers that use JAD as the engine. The homepage for JAD is rather nomadic, so best to find it through a search engine.
- *Jode* (http://jode.sourceforge.net) – Decompiler written in Java with an easy to use GUI and command line interface.
- *JReverse Pro* (http://jrevpro.sourceforge.net) – Not well known, but a very capable decompiler written in Java.

There is not a lot of active development of free decompilers and you may have to try multiple decompilers on problematic files. Since the decompilers act on the Java byte code, they are usually specific to byte code versions.

A cursory analysis of the decompiled source code at this stage should improve our understanding of the application and guide us towards potential weaknesses. During this analysis we should be on the lookout for:

- The communications layer – Find out which classes perform the communications so that we can use these objects to perform low level operations on the server side.
- Client side security controls – Does the client enforce security controls such as authorisation and data validation? (What should really be tested is whether bypassing these controls on the client will allow us to perform unauthorised operations on the server side).
- Cryptography – Where are the cryptographic functions performed and how? Are there shared keys hard coded in the source?
- Authentication – How is authentication performed? How are credentials passed to the server?
- Session Management – How is session management performed, and is it feasible to hijack another user's session?

4. Insert the BeanShell

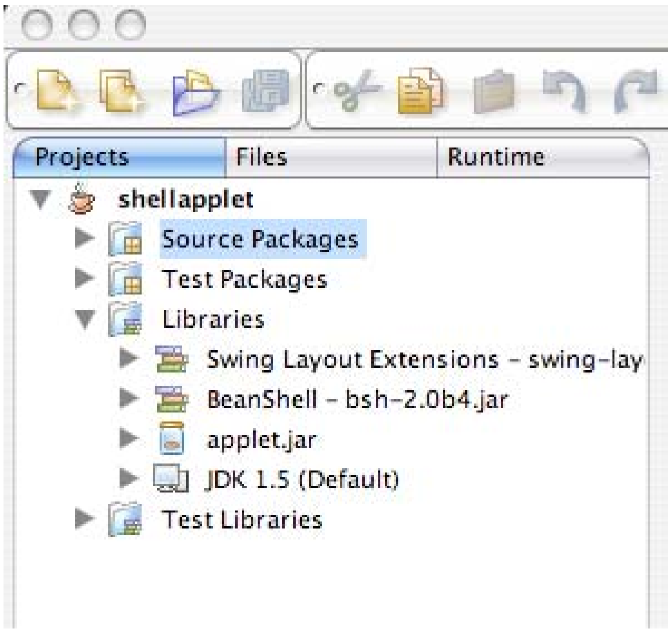
The Java BeanShell (www.beanshell.org) provides a convenient means for interacting with Java applications dynamically. It is similar to the interactive command line of Python or Ruby's IRB and allows us to view or modify objects in the application at runtime.

It is simple to insert the BeanShell into a new application where the source is available; all that is required is an import statement and the initialisation of an object. When the source is not readily available (or not in a compilable state, as is usually the case when it was obtained from a decompiler), then an additional step is needed.

One of the easiest ways of inserting new code into an existing application is to subclass one of the existing classes that provides the entry point into the application. For a Java application, this will be the class that contains the static void main method. For a Java applet, the main class is usually specified in the "CODE" value of the "APPLET" tag. In the case of Jeti, it is: nu.fw.jeti.applet.Jeti.class

To subclass this class:

1. Create a new Java Applet project in your favourite IDE.
2. Add the target jar file (in this case applet.jar from Jeti) and the BeanShell jar to the classpath of the project.



```
shellapplet
    Source Packages
    Test Packages
    Libraries
        Swing Layout Extensions - swing-lay
        BeanShell - bsh-2.0b4.jar
        applet.jar
        JDK 1.5 (Default)
    Test Libraries
```

3. Create a new class which extends the main Jeti class and insert the calls to the BeanShell:

```
package shellapplet;

import bsh.Interpreter;
import nu.fw.jeti.applet.Jeti;

public class MyApplet extends Jeti {
    Interpreter i;

    /** Initializes the applet MyApplet */
    public void init() {
        super.init();
        i = new Interpreter();
        try {
            i.set("app", this );
            i.eval("setAccessibility(true)");
            i.eval("server(7777)");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

4. Compile the project and create the new applet JAR file.

The lines highlighted in yellow in step 3 above are discussed in more detail here:

```
i.set("app", this);
```

This provides a reference to the current object (the MyApplet class), to the BeanShell interpreter and calls it "app". We will use this reference to access the applet object from within the interpreter.

```
i.eval("setAccessibility(true)");
```

Turns on unrestricted access to private and protected values. This is an invaluable feature when manipulating the internals of the code since with it enabled, we are not bound by the Java access restrictions.

```
i.eval("server(7777)");
```

Start the server on port 7777. Two servers are actually started, an HTTP server on port 7777 which serves an applet that acts as an interface to the interpreter (An applet that spawns a webserver that serves an applet!). A second shell into the interpreter is started on port 7778 that can be accessed with telnet.

5. Edit the Local Security Policy

Through the Applet, the BeanShell is going to start a local servers on ports 7777 and 7778 and make connections out to arbitrary hosts. However, such actions are prevented by the default local security policy for Java applets. The default Java security policy typically prevents Applets from performing dangerous operations including reading/writing from/to the local file system, executing system commands and controlling socket options. But since we need to perform some of these operations, the policy will have to be relaxed. The Java security policy is stored in the file .java.policy, usually in the user's home directory. Make a backup of this file; then add the following segment:

```
grant codeBase "http://localhost/applet/-" {
	permission java.security.SocketPermission "localhost:7777", "accept, connect, 
listen";
	permission java.security.SocketPermission "localhost:7778", "accept, connect, 
listen";
	permission java.lang.RuntimePermission "accessDeclaredMembers";
	permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
	permission java.util.PropertyPermission "debug", "read";
	permission java.util.PropertyPermission "trace", "read";
	permission java.util.PropertyPermission "localscoping", "read";
	permission java.util.PropertyPermission "outfile", "read";
	permission java.net.SocketPermission "*", "resolve, connect";
	permission java.io.FilePermission "/localhost/applet/*", "read, write";
	permission java.io.FilePermission "/localhost/applet/plugins/*", "read";
	permission java.io.FilePermission "file:/tmp/notes", "read,write";
};
```

This policy is specifically tailored for the Jeti applet and the BeanShell, you will have to derive a suitable policy for other applets.

Another method of relaxing the policy is to grant the applet all permissions. This is not advised when the code being tested is not completely trusted, since it would allow the applet to behave as a fully trusted application without any restrictions on the operations it can perform. All permissions can be granted with the following policy:

```
grant codeBase "http://localhost/applet/*" {
      permission java.security.AllPermission;
};
```

6. Deploy and Run

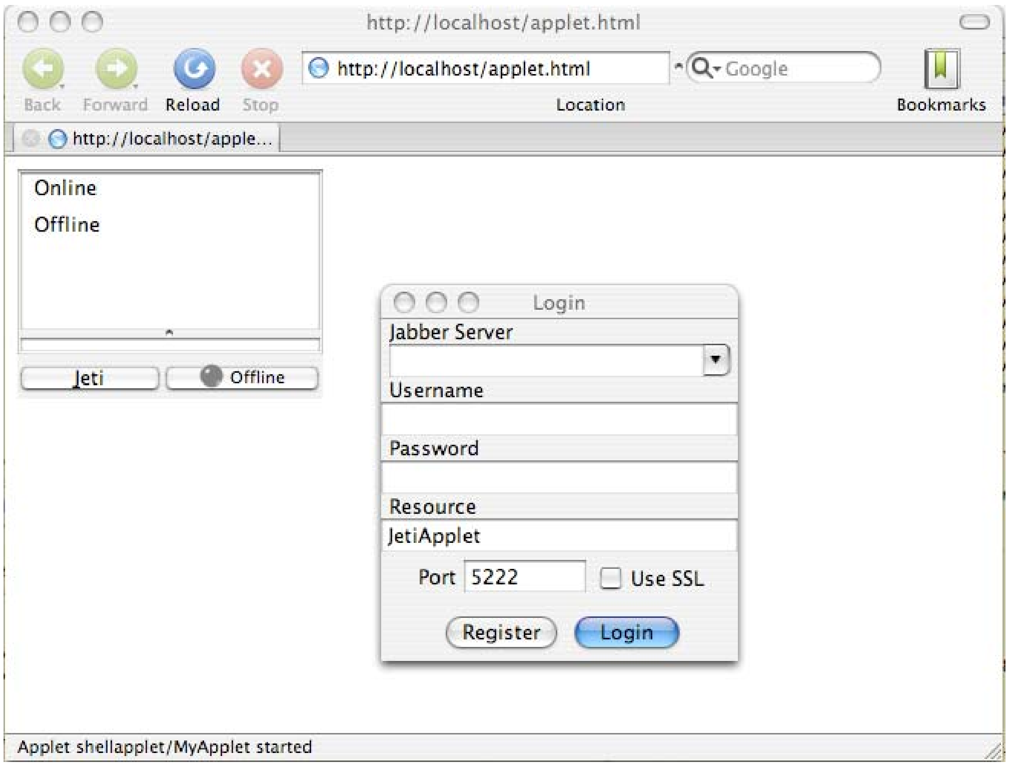
Next, start a local web server and copy the new applet jar file and the needed library jar files into a web accessible directory. In Jeti's case the necessary files are:

- applet.jar – the original jar from Jeti;
- shellapplet.jar – the newly created applet that extends Jeti and inserts the BeanShell; and
- bsh-2.0b4.jar – the BeanShell libraries.

To launch the applet, create an HTML file such as:

```
<html><body>
<applet code="shellapplet/MyApplet.class" archive="/applet/shellapplet.jar, /applet/bsh-
2.0b4.jar, /applet/applet.jar></applet>
</body>
</html>
```

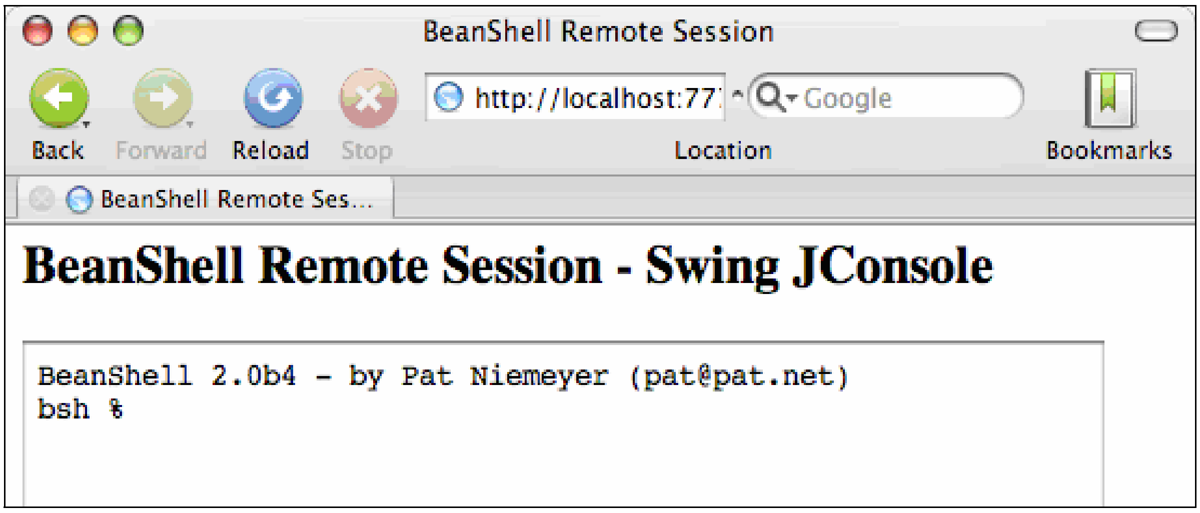
Viewing the page should start the applet:



The BeanShell should now also have been started on ports 7777 and 7778:

```
 [~]telnet localhost 7778
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh %
```

The BeanShell applet can be accessed from: http://localhost:7777/remote/jconsole.html through a browser:



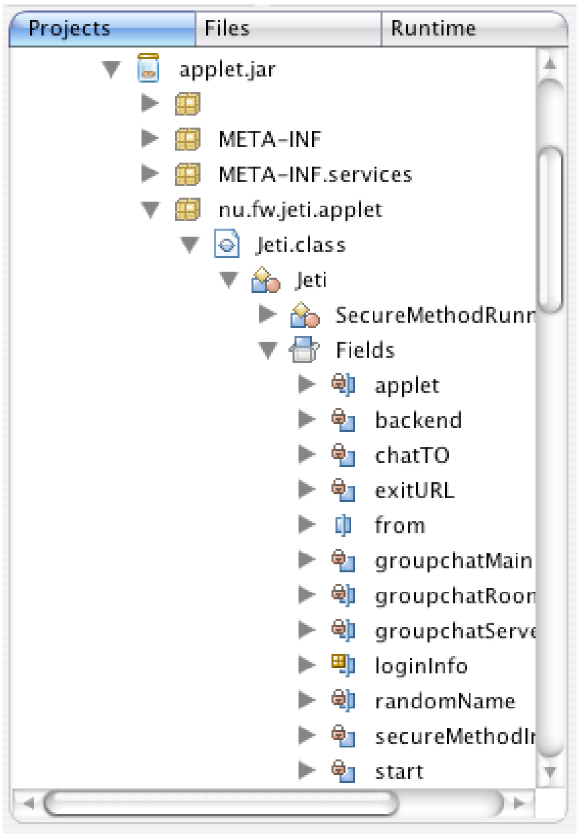
```
BeanShell 2.0b4 - by Pat Niemeyer (pat@pat.net)
bsh %
```

7. Inspect and Manipulate Objects

Now that the BeanShell has been started we can view and manipulate objects dynamically. When starting the BeanShell, recall that the applet itself was passed in as a reference called “app”. Using BeanShell syntax, we can access all methods and fields on this object. For example:

```
bsh % print(app.isLoggedIn());
false
bsh % app.login("corsairetest","amber.org.uk","test");
bsh % print(app.isLoggedIn());
true
```

The easiest way to discover the fields and methods of objects is to view the decompiled source, or to browse the applet.jar file in an IDE:


Projects
Files
Runtime

```
applet.jar
  
  META-INF
  META-INF.services
  nu.fw.jeti.applet
    Jeti.class
      Jeti
        SecureMethodRunr
        Fields
          applet
          backend
          chatTO
          exitURL
          from
          groupchatMain
          groupchatRoon
          groupchatServe
          loginInfo
          randomName
          secureMethodIr
          start
```

For a more hands-on approach, you could also use the Java reflection API in the BeanShell:

```
bsh % import java.lang.reflect.*;
bsh % Field[] fields = app.getClass().getSuperclass().getDeclaredFields();
bsh % for (int i=0;i<fields.length;i++) {print(fields[i].toGenericString());}
private nu.fw.jeti.backend.Start nu.fw.jeti.applet.Jeti.start
public static javax.swing.JLabel nu.fw.jeti.applet.Jeti.from
private static nu.fw.jeti.applet.Jeti nu.fw.jeti.applet.Jeti.applet
private java.net.URL nu.fw.jeti.applet.Jeti.exitURL
private static java.lang.String nu.fw.jeti.applet.Jeti.groupchatRoom
private java.lang.String nu.fw.jeti.applet.Jeti.chatTO
private static java.lang.String nu.fw.jeti.applet.Jeti.groupchatServer
static nu.fw.jeti.backend.LoginInfo nu.fw.jeti.applet.Jeti.loginInfo
private static boolean nu.fw.jeti.applet.Jeti.randomName
private nu.fw.jeti.jabber.Backend nu.fw.jeti.applet.Jeti.backend
private boolean nu.fw.jeti.applet.Jeti.started
private nu.fw.jeti.applet.Jeti$SecureMethodRunner
nu.fw.jeti.applet.Jeti.secureMethodInvoker
private boolean nu.fw.jeti.applet.Jeti.groupchatMain
static java.lang.Class
nu.fw.jeti.applet.Jeti.class$nu$fw$jeti$events$StatusChangeListener
```

Notice that because of the call to set Accessibility(true) when the shell was initialised, it's possible to access private fields and methods.

```
bsh % print(app.backend.connect.getMyJID());
corsairetest@amber.org.uk/JetiApplet
```

At this stage, we have a working platform for manipulating the internals of the application during execution via an interactive shell and can now test the security issues mentioned in section 0 above. In addition to providing low level access to the program API, the BeanShell can also be used to perform tedious repetitive tasks. For example, it would be trivial to perform a dictionary or brute force attack on the authentication mechanism by writing a simple BeanShell script.

For a complete guide to the BeanShell scripting environment see the documentation at http://www.beanshell.org.

8. Conclusion

The BeanShell provides a convenient means of inspecting and manipulating a Java application during execution. This allows the security tester to bypass security controls on the client and verify the security controls on the server. It also allows for the automation of tedious tests such as brute force testing.

The BeanShell can be inserted into a new application in a few simple steps. If the source code is not available, the BeanShell can be inserted by extending the class that acts as the entry point into the application. These techniques apply equally to full blown Java applications as well as applets.

9. References

- Java 2 Platform Standard Edition 5.0 API – http://java.sun.com
- BeanShell documentation – http://www.beanshell.com
- Covert Java - http://www.samspublishing.com/articles/article.asp?p=174217

10. Acknowledgements

This Guide was written by Stephen de Vries, Principal Consultant at Corsaire.

10.1 About The Author

Stephen de Vries is a Principal Consultant in Corsaire's Security Assessment team. He is currently co-leading the OWASP Java Project, helping Java and J2EE developers to build secure applications efficiently. He has worked in IT Security since 1998, and has been programming in a commercial environment since 1997. The last five years he's been focused on Ethical Hacking, Security Assessment and Audit at Corsaire, KPMG and Internet Security Systems. He has also been a contributing author and trainer on the ISS Ethical Assessing course.

10.2 About Corsaire

Corsaire is a market leader in information security consultancy and vulnerability research. Privately founded nine years ago, we provide a range of consulting, assessment and research services to help organisations measure their security posture and build a thorough compliant security program to support their business strategy.

We operate on an international basis with presences across Europe and the Asia-Pacific rim. Our clients include some of the world's best known blue-chip multinationals, many of whom are listed on the FTSE, DAX and Fortune 500 stock indices although we also have a selection of UK government authorities (including National Security Agencies – CESG and NISCC) and mid-range organisations.

Most of our clients have been drawn the e-banking, finance, telecommunications, insurance, legal, IT and retail sectors. They are all mature buyers, operate at the highest end of security and understand the difference 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