

Introduction

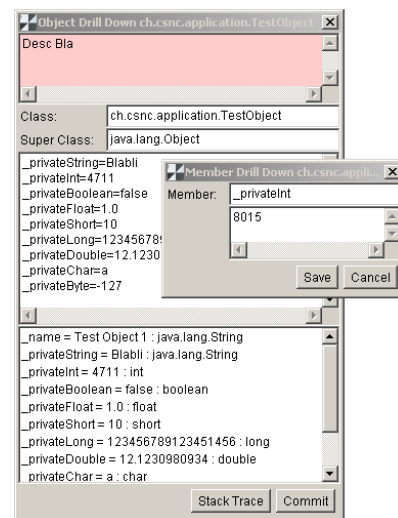
Penetration testers are often faced with the situation in which they have to test authentication, authorization and failure behavior. One question could be: Is it possible for a customer to access or modify the data of another customer? For browser applications to test this, they modify the requests sent to the server using some kind of inspection proxy, like @tstake WebProxy, Achilles or SSL-Proxy.

However, there are also non-browser client applications written in high-level languages like Java. Often these applications do not communicate in plaintext HTTP requests with the server but instead utilize some sort of binary communication. Such traffic cannot be decoded and modified easily due to their proprietary data format, which makes testing with proxy tools like the ones mentioned above almost impossible.

The Solution

To facilitate the penetration testing of client applications written in Java 1.2 and above, Compass Security has developed a tool called the *Java Object Inspector*. This tool allows inspection and modification of data records (i.e. member variables of Java objects) in running Java applications and applets.

To achieve this, a call to the *Java Object Inspector* is placed to the reverse engineered Java application and the desired Java object is passed as a parameter. During application execution, the member variables of the object are displayed in a pop-up dialog and the application execution is halted. The user is then able to modify the values to his needs. After completing the changes, application processing continues as usual.

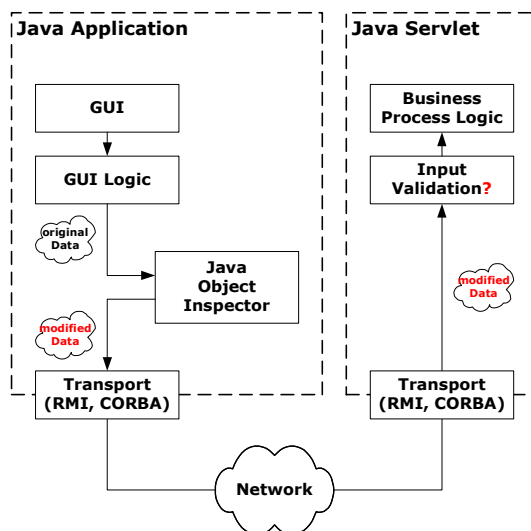


Modifying members with Java Object Inspector

The risks of breaking the tested application are minimal since only a very small part of the Java code is modified. Furthermore, analyzing the behavior of an application during execution is much easier than just reviewing the application's source code and hard-coding each and every test case manually into the application. Conducting penetration testing with the *Java Object Inspector* is done with ease!

Never Trust a Client

In applications, the data entered by the user is often sanitized on the client-side. In browser applications, this is done with JavaScript. However, an attacker can bypass such checks easily by intercepting a sane request with a proxy and changing it into a malicious request. This way, arbitrary data can be passed to the server.



Data flow with Java Object Inspector



Java Object Inspector 1.0

by Jan P. Monsch
jan.monsch@csnc.ch

If an application solely depends on client-side validation checks, the server infrastructure and the application users are at great risk. The following examples show the different threats and possible consequences:

- Reading or changing data of other users. Imagine a banking application in which a malicious user reads another's account information and pays his bills with the money of this other user.
- Syntactically and semantically uncorrected data can be passed into the business application. Imagine a malicious Internet banking user paying a bill worth CHF -300. Instead of being debited, his account gets a credit of CHF 300.
- Another variation of the previous situation could be that an attacker uses Cross-Site-Scripting to steal session credentials, like session cookies, to hijack the sessions of other users.
- Or another scenario could be that an attacker uses SQL injection to exploit the server-side infrastructure itself in order to gain access to the DMZ or, even worse, the Intranet.

Client-side software must be considered as being the most insecure component in a client-server application. Reverse engineering and modification of client software is always possible.

Therefore, data coming from clients must always be validated on the server. It must always validate the given input data according to the following criteria:

- Syntax correct - Are all characters in the phone number numeric?
- Semantically correct - Is the value of the payment a positive number?
- Does it contain characters which are dangerous? - `` ` ; : _ % & < > / \ * + # [] {}

- User authentication - Is the user logged on?
- Is the user authorized to access the requested data or function? - Is the user allowed to view the account balance for a given account number?

If validation fails, the request must immediately be aborted and a generic error message must be returned to the client.

Need for the Object Inspector

Testing validation and error behavior of web-browser based applications is rather easy since they communicate with the server in plaintext. But, if the application is a Java application or applet talking to the server using a proprietary binary protocol standard proxy based inspection software fails.

Now with the *Java Object Inspector* it is possible to change requests:

- within the application when the data is still available as an object
- before the objects are actually streamed into the transport layer
- with minimal code changes in the modified client application.

Executing a test case is now a matter of changing values using a dialog box.

The Glory Details

The reader now probably asks himself: "How does *Java Object Inspector* achieve this?" The little bit of magic is called Java Reflection API. It allows application developers in a generic and fully dynamic way to:

- create new instances of objects
- call methods of existing objects
- inspect and modify member variables of objects (public as well as private ones).

```
//Loads Class with a String
Class fileClass =
    Class.forName("java.lang.File");

//Gets the constructor for the File class:
Class[] constParamTypes =
    { String.class };
Constructor constructorFile =
    fileClass.getConstructor(constParamTypes);

//Calls the constructor of the File class:
Object[] constParams = {"C:\\\\"};
Object fileObj =
    constructorFile.newInstance(constParams);

//Calls the list method of the file object:
Method listMethod =
    fileClass.getMethod("list", new Class[0]);
String[] files = (String[])
    listMethod.invoke(fileObj, new Object[0]);
```

Directory listing with Java Reflection

```
//Gets the class of an object
Class class = object.getClass();

//Gets all the declared fields of the class
Field[] fields = class.getDeclaredFields();
AccessibleObject.setAccessible(
    fields, true);

//Gets the first member of the object
Object memberValue = field[0].get(object);
```

Accessing an object member variable

This functionality is also used by the Java serialization mechanism for streaming live Java objects into files or over a network (e.g. RMI). The serialization of objects is convenient because the developer does not have to define his own protocols. Another common application of Reflection is to create instances of classes from a string stored in a configuration file.

The *Java Object Inspector* uses this interface to inspect and modify the content of member variables. For penetration testers, all this magic is hidden away. The only thing they have to do is to place the following hook for inspecting the desired object into the code:

```
ObjectInspector.inspect(myObjectToInspect,
    "Title", "Description", InspectorColor.RED);
```

Java Object Inspector Hook

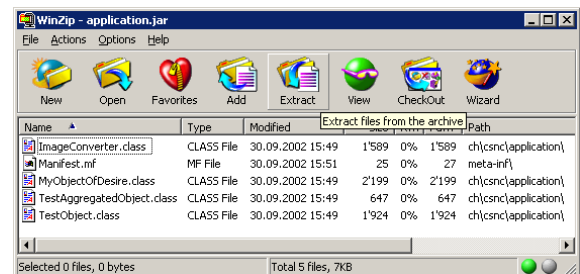
How to Prepare the Rod

Now the question is raised of how to place the hook into the Java application. The following paragraphs deal with the process of modifying the application to use the *Java Object Inspector*.

Get the Worms...

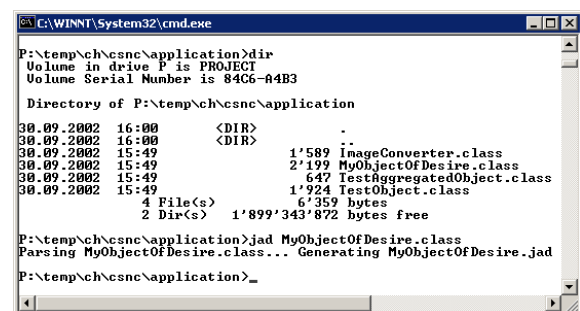
In order to utilize the *Java Object Inspector*, the penetration tester needs some source code to place the hook. But often, only the final binary build of a product, and no source code, is available. Therefore, the application must first be reverse engineered.

Java classes are often delivered as a group of JAR files (Java Archive), which are actually simple ZIP files. The Java classes can then be extracted from the archive using a tool like WinZip.



Content of JAR archive viewed in WinZip

These files are the actual Java byte code that can then be passed to tools like JAD or DeCafe to generate source code. Later this source can be compiled again into running byte code.



Decompiling a Java class with JAD

Prepare the Hook...

Since the class and method names originally chosen by the developer are often disclosed in the Java byte code, identifying the purpose of a class is quite simple.

Sometimes the developers use tools to obfuscate these names in the release build of their product. Although this makes class identification more difficult, it is not much of an obstacle since calls to the Java API are still visible.

Then the penetration tester has to find the section in the code in which he is most interested. When inspecting communication with the server, he or she has to find the code section where the data records, e.g. Java objects are put onto the network. There he or she places the call to the *Java Object Inspector* just before the object is streamed to the network connection.

Cast the Rod... Enjoy

In Java, the smallest dynamically loadable program unit is a single Java class. Therefore, compiling just the modified class is sufficient. The class file can then be placed together with the *Java Object Inspector* classes into the original JAR file. WinZip can be used for this task.

The only thing left to do is to start up the application, click through the GUI, and wait for the *Java Object Inspector* to pop-up.

Testing with Browsers

Since Java support in Internet Explorer, Netscape Communicator 4.79, and earlier versions is quite out-dated, most business applications will probably rely on Sun's Java Plug-in. The following paragraphs will, therefore, only deal with Sun's Java Plug-in.

Java Sandbox

A Java applet downloaded from the Internet is normally considered to be untrusted and is executed in a sandbox. The sandbox prevents access to system functionality, like file access and Reflection. To get *Java Object Inspector* running, the sandbox policy needs to be relaxed by adding Reflection permissions to the *java.policy* file.

```
// default permissions granted to
// all domains
grant {
    ...
    permission
    java.lang.reflect.ReflectPermission
    "suppressAccessChecks";
};
```

Added Reflection permission in java.policy

CLASSPATH Code Injection

When Java applet code is downloaded, it is normally stored in the browser cache and executed from there. Changing anything in the cache is not very feasible.

Here the Java CLASSPATH gives us a helping hand. It defines where the Java VM has to look for classes on the file system. Since classes on the local machine have priority over the ones loaded from network, we can put the modified JAR files and all needed libraries into the local CLASSPATH.



Java Plug-in Control Panel

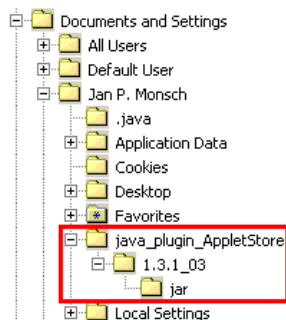
Persistent Applet Storage

Beginning with Java Plug-in 1.3, applets can be made persistent on the client machine. This has the advantage in that the applet must not be downloaded every time it is activated. Applet installation is activated with applet parameters in the HTML page in which the applet is embedded.

```
<OBJECT ....>
<PARAM NAME="archive"
      VALUE="application.jar">
....
<PARAM NAME="cache_option"
      VALUE="Plugin">
<PARAM NAME="cache_archive"
      VALUE="application.jar,library1.jar">
<PARAM NAME="cache_version"
      VALUE="1.2.3.4,1.2.3.4">
</OBJECT>
```

Enabling persistent applet caching

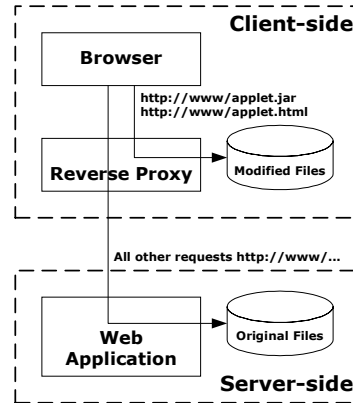
On Windows platforms, the persistent JAR files are stored in a subdirectory of "Document and Settings" of the currently logged-in user. The archive can then be directly modified there.



Location of java_plugin_AppletStore

Reverse Proxy Injection

Another way is to place a reverse proxy between the penetration tester's browser and the web server. The reverse proxy is configured so that the applet and the HTML page with the parameters are not loaded remotely. Instead, the reverse proxy provides them.



Injection with reverse proxy

Such a reverse proxy can be easily built with an Apache compiled with mod_proxy and mod_rewrite.

```
RewriteEngine on

RewriteRule ^/applet.jar /tap/applet.jar [P]
RewriteRule ^/(.*) /direct/$1 [P]

ProxyPass /tap/ http://127.0.0.1/
ProxyPassReverse / http://127.0.0.1/

ProxyPass /direct http://www/
ProxyPassReverse / http://www/
```

Sample httpd.conf for Apache

Signed Applets

The idea behind signing code is to allow the client machine to verify that the:

- downloaded application is from a trusted source
- application has not been tampered since release building.

Since such signatures are verified client-side, they can be removed easily. It is just a matter of repackaging the JAR file.



Java Object Inspector 1.0

by Jan P. Monsch
jan.monsch@csnc.ch

References

Tools

- ❑ Java Object Inspector Download:
<http://www.csnc.ch/uk/knowhow/download.shtml>
- ❑ JAD 1.5.8e – The fast Java Decompiler:
<http://kpdus.tripod.com/jad.html>
- ❑ Decafe 3.9:
<http://decafe.hypermart.net/>
- ❑ SSL-Proxy 3.13:
<http://www.csnc.ch/uk/knowhow/download.shtml>
- ❑ @tstake WebProxy 1.0:
http://www.atstake.com/research/tools/#vulnerability_scanning
- ❑ Achilles:
<http://www.digizen-security.com/projects.html>

Documentation

- ❑ Open Web Application Security Project:
<http://www.owasp.org/>
- ❑ Java Reflection API:
<http://java.sun.com/j2se/1.4/docs/api/java/lang/reflect/package-summary.html>
- ❑ Suppressing Reflective Access Control:
<http://java.sun.com/j2se/1.4/docs/guide/reflection/reflection.html>
- ❑ Appendix A: Security and Permissions:
<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/appA.html>
- ❑ Applet Caching and Installation in Java Plug-in
<http://java.sun.com/products/plugin/1.3/docs/appletcaching.html>
- ❑ Apache module mod_proxy
http://httpd.apache.org/docs/mod/mod_proxy.html

- ❑ Apache module mod_rewrite
http://httpd.apache.org/docs/mod/mod_rewrite.html

About the Author

Jan P. Monsch has a bachelor's degree in Electrical Engineering. After completing his studies in 1998, he worked as a developer for r3 security engineering ag / Entrust Technologies. One year later, he changed to SYSTOR AG and participated in implementing an Internet banking application for a large Swiss retail bank group. Since July 2002, he has been employed by Compass Security and supplements the team with his knowledge of application security.

About Compass Security AG

Compass Security Network Computing AG is an enterprise specialized in security assessment. The company was founded by Walter Sprenger and Ivan Buetler in February 1999 and has, since its epiphany, completed many security assessments both in Switzerland and abroad.

The company, at first, dealt solely with Standard Penetration Tests performed from external sources. At that time, many more Vulnerability Assessment Tools (ISS, CyberCop, Satan, etc) were also used. Unfortunately, these methods are limited, especially when using more complex applications in which the most basic tools often fail to function.

Compass later began working in the field of Application Security Reviews. In this scenario, E-Business applications are tested from external sources and the majority of functions are performed "by hand". Compass tests to see if it is possible for one user to log into another user's data. Questions of data security are essential.

For further details see: <http://www.csnc.ch>