JavaSnoop: How to hack anything in Java

Arshan Dabirsiaghi

Director of Research, Aspect Security

arshan.dabirsiaghi@aspectsecurity.com

Blackhat USA, Las Vegas, NV 2010

Abstract. Many applications in the enterprise world feature thick Java clients. Testing the security of such applications is considered practically more difficult than a similar browser-based client because inspecting, intercepting and altering application data is easy in the browser. With DOM inspection tools like Firebug and WebKit Web Inspector, and HTTP proxy tools such as WebScarab, Fiddler and Burp, assessing the trust boundary between the client and server has become mostly commoditized in web applications.

Security practitioners have been struggling to reach the same level of effectiveness when testing thick Java clients. Researchers have previously tried to statically alter the application code through decompilation and recompilation to add BeanShell script "hooks". Also, work has been done to create proxies that can parse simple serialized objects, a common way of sending data between a Java client and server.

The purpose of this paper is to describe an alternate approach to testing the security of a Java application. This approach utilizes instrumentation and Java agents to make altering traffic, inspecting data and otherwise attacking a Java application endpoint much easier than ever before. The implementation of this approach is a tool called JavaSnoop.

Keywords: application security, instrumentation, agent, Java, virtual machine.

1 Problem definition

On an application architecture diagram, the trust boundaries between clients and servers look generally the same, despite the technologies used on both ends of the communication. It's intuitively obvious that security decisions can't be safely made on the client since theoretically a user can ultimately make their client behave however they please. Therefore, one can always safely assume that a server can be susceptible to attacks executed by a malicious client.

If that client is a web browser, crafting these malicious attacks is very easy. Usually, all an attacker (or a security tester) has to do is configure their browser to use an

HTTP proxy that will allow them to intercept and alter outbound HTTP traffic. Using this proxy, the attacker can launch injection or privilege escalation attacks, or even perform scripted actions.

If that client is an RIA or desktop program, crafting those same attacks can become substantially more difficult. Consider a Microsoft Silverlight client that communicates with simple Web Services over a network. To test the security of the application, the attacker must alter the existing client to act malicious (hack the client), proxy the traffic as you would to a similar web application (hack the traffic), or interface with the server directly (hack the server).

1.1 Statically altering the client

Statically altering client programs (or, "hacking the client") in RIAs is not easy. Many RIAs live out their entire lifecycle in the memory of an already-existing process. Altering this memory to cause malicious application behaviour is not cost-effective. Reverse engineering memory layout and safely altering process memory directly are both extremely difficult.

For RIAs that live on disk, altering clients has been shown to be at least marginally more viable. The RIA discussed in the paper, Java, should in theory be accommodating to this approach. If an attacker has access to a binary (such as a JAR file), they should be able to perform the following steps:

1. Decompile the code from binary to pure source
2. Alter the code to either contain stageloading code or attack code directly
3. Recompile the code from the altered source and re-run

Although relatively time consuming given the constraints of a typical security assessment, this approach seems reasonable on paper. Unfortunately, in practice this process turns out to be very error-prone.

Decompiling binary Java often results in source code that has a number of compilation errors[1]. These errors are introduced by bugs in the decompilers themselves or the result of special build processes, which show that the compilation and decompilation processes are not, in practice, 100% deterministic. These compilation bugs are costly to chase down and fix, especially when combined with obfuscated code. Overcoming these hurdles make for a protracted process which often ends up not producing the "evil client" due to time restrictions. There is also the unfortunate fact that many security testers are not qualified technically to "recover" such decompiled source.

On top of this, test iterations with this method are slow compared to approaches where the application's generated traffic is altered in transit.

1.2 Intercepting application communication

The second option for sending test attacks to the server is to alter the application's traffic. If the application uses HTTP (as many do), efficient testing is easy as configuring the application to use an HTTP testing proxy as would be done for a normal browser client. The application may have an interface for setting up a proxy, or the default proxy for the Java process can be set with command-line switches to the Java executable. Altering the command line for a Java Web Start (JWS) program or an Applet is possible but not straightforward. An example of using the command line switches is shown in Figure 1.

```
$ java -Dhttp.proxyHost=localhost -Dhttp.proxyPort=8080
```

Fig. 1. A Java process being started whose HTTP libraries will by default use the HTTP proxy located at localhost:8080.

There are multiple problems to be encountered when using this method. The first common error occurs when the application communication is over SSL and the certificate or certificates involved are sanity checked by either party. When this is done, an unexpected certificate will be noticed and the communication is typically terminated.

Even if the application doesn't use SSL or doesn't check the certificates, it is possible the application uses HTTP as a transport protocol but doesn't use a plaintext data format. Applications sometimes use serialized objects or protocols on top of HTTP, which makes intercepting traffic and tampering with the data inefficient at best. There are generic man-in-the-middle tools that can theoretically proxy the network traffic, but they generally require far too much customization to be cost-effective during a small to medium length engagement. There has been recent work to create an HTTP proxy plugin that allows users to edit serialized Java objects in custom editors[2]. Although the interception is seamless in this work, easy editing of application data is still not possible even using today's advanced toolsets.

1.3 Attacking the server directly

It's sometimes possible to attack the server directly without need for the client. This is especially true for web services or plaintext HTTP endpoints. However, attackers still need to discover what "valid" transmissions look like in order to send effective attacks.

2 Solution: The JavaSnoop tool

JavaSnoop is a new breed of tool that was created to alleviate the problems discussed in the previous section. The goal of JavaSnoop was to make a security testing program for Java applications that had the following qualities:

1. Allow easy interception of any method in the JVM
2. Allow the editing of return values and parameters
3. Allow custom Java to be inserted into any method
4. Able to work on any type of Java application (J2SE, Applet, or Java Web Start)
5. Able to work on already-running Java processes
6. Not require any target source code (original or decompiled)

The only way to accomplish these goals is to add stageloading hooks into the bytecode of the classes targeted. To do that without recompilation was impossible before Java 5.0. However, Java 5.0's addition of non-native Java agents and the Instrumentation class opened the door for advanced modification of a JVM at runtime[3].

It wasn't until Java 6.0 came out with the Attach API that the seamless, inter-process modification of a running JVM became practical. The Attach API is a Sun extension that provides a way for a Java process to "attach" to another JVM at runtime. This bridge can be used to load Java agents onto the remote virtual machine. Those agents can then redefine classes or retrieve information about the JVM to which it's attached[4]. This mechanism allows JavaSnoop to satisfy the requirements listed above. JavaSnoop can use the Attach API and the Instrumentation class to jump into another JVM on the machine and install various "hooks" throughout class methods on that system. These hooks are then used by an agent to communicate with a GUI that allows the JavaSnoop user to "intercept" calls within the JVM.

To redefine a class, one must be provide a raw byte array containing the bytecode of the modified class. This means that to have meaningful testing, the class must do everything it's originally designed to do, but also perform these JavaSnoop-related tasks.

This required selective modification of existing class bytecode. Although Java bytecode is relatively easy to understand and implement, a bytecode generation library was used to speed up development time. This library, Javassist[5], is used for general-purpose class manipulation. It allows a user to abstractly insert additional Java code into Java class methods.

Because all types of Java processes still reside within a standard JVM, JavaSnoop is able to "attach" to any type of Java process on a machine. It should be noted that JavaSnoop can also kick off a process from scratch in case the user wants to intercept events that would be missed by attaching to the process after startup.

2.1 Hooks

The purpose of JavaSnoop is to install "hooks" into methods. Those hooks can perform one or many of the following actions:

- Edit method parameters
- Edit method return value
- Pause method
- Execute user-supplied script at the beginning of the method
- Execute user-supplied script at the end of the method
- Print the parameters to the console (or to a file)

There are lots of reasons to "hook" a method. Perhaps the application returns "false" from a method that performs a license check, and the attacker would like to change that value to "true". Maybe the application sends serialized objects over SSL, and you would like to tamper with the object just before it reaches the call that sends it over the network. It's possible that there is a time-of-check-time-of-use vulnerability that you would like to make easier to exploit by pausing the application at a certain call.

Complex situations could occur in which the JavaSnoop user required writing custom Java code in order to exploit some condition or initiate their own actions instead of intercepting existing ones. To accommodate this, JavaSnoop allows the user to execute arbitrary Java code at the beginning and end of any method.

Perhaps what you're looking for is less complex, and you just want to see what the parameters sent to certain methods are. Printing the parameters of `MessageDigest.getInstance(String algorithm)` calls would show which hashing algorithms are being used on the client. This common and simple function is also possible.

Since JavaSnoop makes application data and traffic easy to tamper with, figuring out the right method to hook has become the difficult part of the assessment. Although nothing can substitute code review for understanding an application's logic, a user without access to the source code has a few options for finding the right hook. The user can choose a Java API they suspect may play a role in a test, they can search for methods by name or class, and they can use a special investigative mode of JavaSnoop, called "Canary Mode".

2.1.1 Hooking a Java API

Users can hook public or private Java API as long as the method is not native. Hooking Java API in Applets with anything besides custom-written scripts will fail because of the nature of the Applet class loader and some of the security restrictions on Applets.

Most of the time, however, users will be more interested in hooking an application's custom code.

2.1.2 Searching by method name or class

Users can look up classes to hook methods within if they know or can guess the relevant class name. Many applets have a relatively small number of classes, which makes browsing the entire class tree relatively quick.

Alternatively, users can search for methods by name.

2.1.3 Canary Mode

Even after searching and guessing, it may be difficult to find what methods to intercept. It's likely that attackers are interested in methods where data they put into the UI ends up going. If the flow of their data through the class methods could somehow be seen, it may end helping the user find functions to hook.

Discovering this lifetime is the purpose of "Canary Mode", a unique and useful feature of JavaSnoop. In this mode, you define some "canary" value that you want to trace through the system. This should be some unique value that you're going to enter into the application somewhere, probably through a form field or a properties file. Once this value is chosen, Canary Mode can be started. JavaSnoop will then remove all other hooks currently in use, and then add canary "listeners" to every method in the JVM that has the data type of the canary as a parameter.

Each time the canary is found being sent to a method, a "chirp" is sent back to JavaSnoop, letting the user know what method operated on the canary value. In a way, this amounts to a very primitive, clumsy form of data flow analysis.

Instrumenting a sizeable percentage of methods in the JVM is a process costly in cycles. And once those methods are instrumented, the application will perform slowly since the canary "listeners" are being executed constantly. Therefore, Canary Mode is a standalone modal that cannot be executed concurrently with other hooks. It is also possible to limit the methods hooked to a certain package.

Canary Mode can be used with all primitive data types and the String object.

2.2 Conditions

Hooks can sometimes be needed on functions that are called many, many times. To make sure JavaSnoop only executes when needed, a hook can have zero or more "conditions".

A user can ask JavaSnoop to only hook when the specified conditions are met, or aren't met. Each condition is a logical test on a parameter passed to the function. For instance, a method that has boolean and String parameters can have two conditions that specify the hook's actions should only be performed when the boolean value is "true" and the String parameter contains the word "security".

2.3 Sessions

JavaSnoop supports the idea of persistent "sessions", not unlike other security tools. A session consists of a number of "hooks" and process information. Using this feature, users easily begin where they left off auditing an application, including console output.

3 Practical usage information

The JavaSnoop tool has been tested on Java applications of all types and on different operating systems. Some important details are worth covering here regarding its usage.

3.1 The tools.jar problem, solution

JavaSnoop makes use of the Attach API, a JRE-specific library. The library that contains the Attach API is tools.jar. This library is only installed in JDK distributions, and is OS-dependent. Therefore, for JavaSnoop to work on your machine, you must copy this jar file from your successfully installed JDK into JavaSnoop's lib directory.

Not having the right version of this file will probably cause an exception that mentions special operating system classes that are not for your machine.

3.2 The permission problem, solution

The JavaSnoop agent that gets installed into applications during operation requires permissions not usually granted to untrusted code. For instance, the agent opens up a background thread and server socket in order to communicate with the GUI. Because these operations are not usually granted to untrusted code, the JavaSnoop user must grant all privileges (java.security.AllPermission) to the application being assessed. J2SE applications executed normally on the desktop already have this permission, but the user must alter their java.policy files for the agent to work on applets and Java Web Start programs. Extreme care should be taken when granting this privilege, since this Java policy is the same one that will be used in the user's browser as they execute arbitrary applets on the web.

Permissions can be granted to specific sites and codebase URLs in order to restrict the likelihood of erroneously giving permission to arbitrary malicious applets. There is an open question of whether AllPermission can be granted to a particular site on Java 1.6+ on Windows Vista, but other operating systems have not exhibited this problem[6].

3.3 JavaSnoop doesn't make overcoming obfuscated code any easier

Obfuscated code isn't any easier to understand using JavaSnoop. However, Canary Mode can be used to find methods that may be directly pertinent to communication. Also, public or private Java API can still be searched and hooked without issue.

3.4 JavaSnoop only works on newer JREs

JavaSnoop's reliance on Attach API means the JRE of JavaSnoop and that of the target process must be 1.6+. Since the attacker controls their own environment, it should be relatively simple to enforce this.

Attackers may run into issues where a Java desktop application they are using relies on its own distributed version of the JRE, which is of a lower version. In most cases it's possible to find this JRE (usually located in the application's installation directory) and replace it with an updated version.

Acknowledgments. Thanks to Aspect Security's Nick Sanidas, David Anderson and Jeff Williams, who helped germinate the idea and wrote some useful code. Thanks are also due to Dave Wichers and David Lindner, also of Aspect Security and Marcin Wielgoszewski of Gotham Digital Security for feedback, criticism and support.

References

1. Dyer, Dave: Java decompilers compared. http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html (1997)
2. Saindane, Manish: Attacking JAVA Serialized Communication. https://media.blackhat.com/bh-eu-10/whitepapers/Saindane/BlackHat-EU-2010-Attacking-JAVA-Serialized-Communication-wp.pdf (2010)
3. Sun Microsystems: New Features and Enhancements: J2SE 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#instrument (2004)
4. Zukowski, John: CORE JAVA TECHNOLOGIES TECH TIPS: The Attach API. http://blogs.sun.com/CoreJavaTechTips/entry/the_attach_api (2007)
5. Chiba, Shigeru: Javassist. http://www.csg.is.titech.ac.jp/~chiba/javassist/ (2009)
6. User: nahsra. How do I grant a site's applet an AllPermission privilege? http://stackoverflow.com/questions/2828075/how-do-i-grant-a-sites-applet-an-allpermission-privilege (2010)