ATTACK & DEFENSE
labs

Attacking JAVA Serialized Communication

By: Manish S. Saindane

Black Hat Europe 2010

Introduction

Many applications written in JAVA make use of Object Serialization to transfer full blown objects across the network via byte streams or to store them on the file system. How Object Serialization works is beyond the scope of this whitepaper. To understand the Serialization of objects and details of the Serialization protocol check the Java Serialization Protocol Specification provided by Sun (Sun Microsystems). This whitepaper introduces a new technique to intercept such Serialized communication and modify it to perform penetration testing with almost the same ease as testing regular web applications.

This technique is more efficient than the currently used methods. It will give the penetration tester the same control and power that an application developer has without most of the drawbacks that are present in the current methods used for testing applications communicating via Serialized Objects.

Usually this kind of communication is used by Thick/Smart Clients (JAVA Applets, Swing/SWT clients, etc.) to transfer data to the Application server. The communication could be over HTTP or any other protocol. Let's assume HTTP communication for simplicity. Unlike typical web applications, these clients do not pass data in plain text as part of POST requests. The POST data usually consists of Serialized Objects, either passed as an octet-stream or in a g-zipped format.

The Current Scenario and Challenges Faced

While testing applications communicating via Serialized Objects, current tools/application interception proxies allow very limited functionality to intercept and modify the requests and responses like in typical web applications. There has been some good work done so far to try and tackle such applications. Some of the methods available till date include:

- Modifying raw HEX data in the POST request using a HEX editor and pass it to the server.
- Decompiling client-side code and possibly modifying and rebuilding it.
- Injecting a BeanShell within the client to inspect and modify objects dynamically, introduced by Stephen de Vries from Corsaire Ltd. (Vries, 2006).
- And finally a very interesting technique called Runtime Protocol Analysis, introduced by Shay Chen from Hacktics Ltd. at an OWASP meet (Chen, 2008).

Let us first quickly go through each of the above mentioned techniques. This is very important in order to highlight the work done by some absolutely fine researchers and it will also give us a fair idea of what each technique allows us to do and some possible obstacles that we could face while using them in the real world.

While modifying the HEX data is the most simplest of them all, it is very limited. We cannot inspect or modify the current objects or its variables; especially if the variables are not of the simplest form. For example when the variables themselves are some custom objects. It is only possible to tamper with simple variables like integers and strings. But there is a catch. By directly modifying the HEX

data, we might corrupt the Serialized Byte Stream which may result in the application server throwing an error.

NOTE: *Most of the strings within the serialized data are encoded in modified UTF-8 format in which every string is preceded by 2-byte length information (for strings having a length less than 65536 bytes) of the string. Hence if you modify any string in HEX, you also need to modify the 2-byte length that precedes the string to reflect the new length.*

Also existing interception proxies usually incorporate very basic HEX editors and working with such data becomes difficult in these cases.

Decompiling the classes within the client-side jars allows us to carefully study the application logic and identify threats that lie within the client-side code. Things like hardcoded values, sensitive functions, cryptographic algorithms and other client-side validation controls can be identified and used to plan out attacks. Though, decompiling code may not always be that easy for applications making use of obfuscation techniques. There are some really neat decompilers available which might allow us to decompile code which uses simpler obfuscation techniques. Some of the popular decompilers available include JAD, Jode, JD and DJ Java Decompiler.

It might be easy to decompile and possibly modify this client side code, rebuild and package it for smaller applications. But the same might not be the case while working with huge enterprise applications. Also if the signature for each client side jars is validated on the server side, then this technique may be futile. This check could possibly be bypassed by intercepting the traffic and modifying the signature value being passed with that of the original jar.

The technique described by Stephen is quite interesting in which he has explained the use of BeanShell scripting language which he uses to plug in to a client side JAVA application. Though the technique is interesting, it is much broader than the current topic we are discussing i.e. to analyze and tamper Serialized Communication. This technique could be handy in identifying client-side security controls. I recommend the readers to go through his white paper for more details. The only pre-requisites I see for this technique, is that the penetration tester must be comfortable handling the JAVA language and basic compiling techniques. Though the demonstration in the paper looks simple enough, I have yet to try this out myself on an enterprise level application and check its feasibility and simplicity.

The final technique out of the above mentioned techniques that I'm going to discuss here is somewhere closer to our objective. It is called RPA (Runtime Protocol Analysis) which was introduced by Shay Chen at an OWASP Israel meet. In this technique he discussed the idea of creating a custom runtime protocol analyzer to read data from the serialized objects, analyze them, modify and then send the request to the destination server.

For this technique to work, the client communication is first sniffed over the network using sniffers like Wireshark, etc. and then broken down into individual request packets. These individual requests are then modified and sent to the custom protocol analyzer. To understand the complete process in detail, please refer to his presentation.

The only drawback (as I see) with this method is that the whole process is not seamless. There are too many steps involved viz. sniffing communication, spitting into individual packets, casting

exceptions to analyze the protocol structure, etc. This becomes very tedious when conducting a penetration test on a huge enterprise application. It would be much easier for a pen-tester to have some kind of a tool which uses the above mentioned technique but with the same ease as that available while testing normal web applications. Also since the packets are split as individual requests, it might not work for scenarios where multiple requests/responses are to be analyzed.

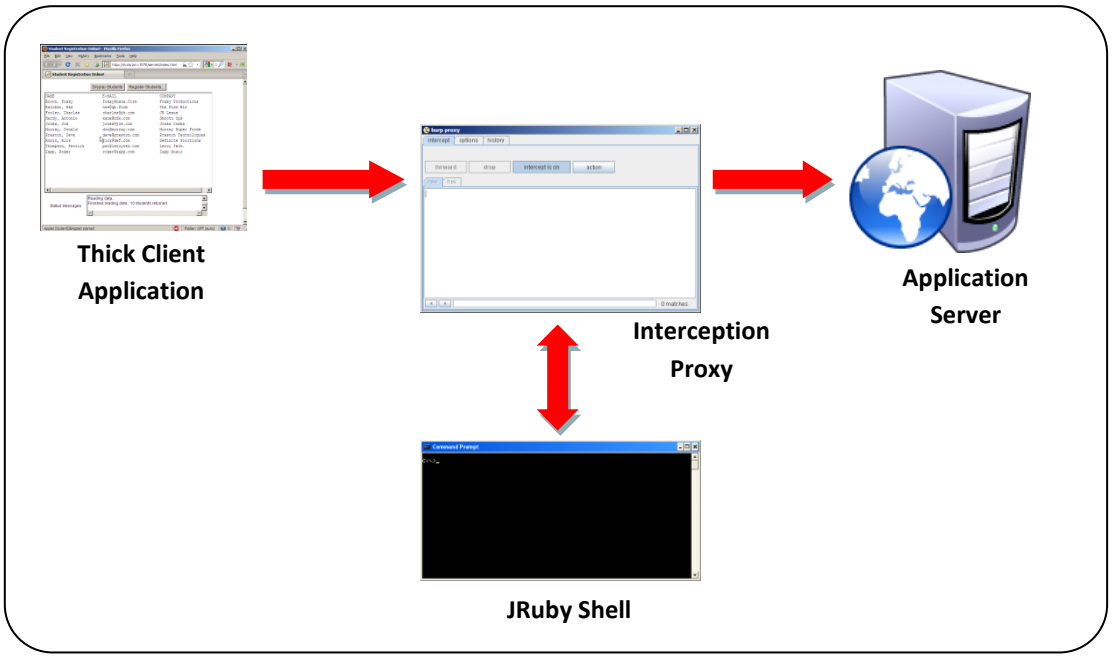
There is one more important point I would like to mention before we move ahead. Many a times while conducting a penetration test, I have found applications which do not contain any configuration settings to specify the upstream proxy for communication. In such a scenario, a simple sniffer could be written to capture the traffic being sent to the destination application server which can be used to forward a copy of the requests to our local interception proxy. For example a utility specifically developed for this purpose called Sniff-n-Spit can be downloaded from here.

For most purpose I prefer using Burp Proxy as my interception proxy as it provides some really good utilities to play around with. It also has an option to forward the intercepted request to any host. Thus the copy of the requests passed by our sniffer to the interception proxy may be successfully forwarded to the destination server.

Suggested Solution

In this article I would be building up on the already available techniques that we discussed above and try to optimize and simplify the whole process. The technique that I'm proposing involves utilizing the existing application interception proxies and adding functionalities (by means of writing custom plug-ins) specific to those that would be necessary for penetration testing of JAVA Serialized Communication.

The setup for this solution would look like this:



In this technique we will try to overcome the drawbacks of the currently available methods by utilizing alternative technologies to achieve similar results. Also the core objective of this technique is to be as seamless as possible and give enough power to the penetration tester to experiment and achieve the desired results.

In RPA, we had to explicitly cast the value of the object derived using the `readObject` method and induce an error to find the actual class of the object. This step was necessary because JAVA is a statically typed language and knowing the type of an object beforehand is necessary. To remove this hindrance we can make use of JRuby which is a dynamically typed language (i.e. the type of a variable depends upon what was last assigned to it). We will build the analyzer in JRuby instead.

Other advantages of using JRuby are:

- It has an easier syntax. Ruby programmers (as well as people with non-programming background) would be fairly at ease with this language.
- We can make use of almost all existing JAVA libraries and it runs on the JVM.
- It provides a nice Interactive shell which allows the execution of Ruby as well as JAVA code with immediate response and in real-time.
- We can use it as a plug-in system for existing JAVA applications.

For this demonstration purpose I will be making use of Burp Proxy as it exposes some APIs to interact with the suite and write your own plug-ins in JAVA using the `IBurpExtender` interface. This interface has been ported to JRuby by **Eric Monti** (Thanks for modifying the code on request for handling binary data) from Matasano Security and it is called **Buby** (Monti). For more information on this please check the references below. Free-edition of BurpSuite version 1.2.1 was used while writing this paper.

I have written a custom plug-in for Burp specifically for this purpose which induces an interactive JRuby shell on every request/response (containing serialized data) that is captured by the interception proxy (***NOTE:*** *The Interception can be stopped or started for individual requests using the custom button provided by the plug-in).* Also the object is read from the Serialized Byte Stream and made available as a variable in this shell for the penetration tester to work with. The plug-in also includes some helper methods that simplifies and reduces the penetration tester's work.

The methods made available by this plug-in allows the penetration tester to print the structure of the object read from the Serialized Byte Stream, and gain important information like the constructors available, the variables defined and also the methods defined within the current object. The plug-in internally makes use of JAVA De-serialization and Reflection APIs for accessing the entire object and allows access to its private and protected variables. Using the methods provided by the plug-in along with the methods exposed by the current object, the penetration tester can easily view/modify the values of these variables and craft custom objects all together.

For a detailed demonstration of the whole process with example, please check this video. In this video we demonstrate a sample application which makes use of Serialized Objects for transferring data from the client to server and vice-versa. While the video is self explanatory, the interesting points to note are the way we are able to inject an interactive JRuby shell within the interception proxy to work on individual requests/responses. Also it may be the case that there are no setter

(`setXXX`) methods available in the client classes to change the values of the variables within the class. In that case, the helper methods from the plug-in can be used to access the private variables and modify their values. Once we are satisfied with the results, we can re-serialize this new object and write it to the `ObjectOutputStream` and send it to the destination server.

Thus using the currently available tools in our arsenal; customizing and utilizing alternative technologies, we are able to test JAVA Serialized Communication. This way we can get an almost seamless method to test JAVA Serialized Communication without having to worry about sniffing the traffic, splitting it into individual packets, modifying HEX bytes and setting up Tomcat servers, etc. All we have to know is some basics in JRuby and the rest (i.e. JAVA Object Serialization & Reflection) is taken care of by this technique and the plug-in.

The technique described here not only tries to simplify most of the penetration tester's work but also gives him/her complete freedom to write JRuby code at runtime using the injected shell to analyze and modify the JAVA Objects, thus making this a very powerful combination of simplicity and customization.

This is just a small example of what can be done. It can very well be extended for other non-HTTP protocols by implementing a custom proxy written in JRuby (or JAVA plus extended with JRuby). Check out the OWASP proxy project for an API to build your own proxy.

References

Chen, S. (2008). Achilles' heel – Hacking Through Java Protocols. *OWASP Israel.* Herzliya.

Monti, E. (n.d.). *Buby*. Retrieved from http://emonti.github.com/buby/

Sun Microsystems. (n.d.). *Java Object Serialization Specification*. Retrieved from sun.com: http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html

Vries, S. d. (2006, June 15). *Assessing JAVA Clients with the BeanShell.* Retrieved from Corsaire: http://research.corsaire.com/whitepapers/060816-assessing-java-clients-with-the-beanshell.pdf