COMP[1298] Postgraduate Project Proposal

Investigation of Java RMI application for potential security vulnerability

Mahabubur Rashid
MSc Computer Forensics and System Security
000582762

1 Overview:

RMI or Remote Method Invocation is a java API that is used to perform the Object Oriented equivalent of Remote Procedure Calls with support for serialized objects transfer and distributed garbage collection. Since the transfer of objects takes place between machines in different address spaces, there are various security measures which are controlled by java security manager.

This project will investigate what these security measures are, how and to what extent they are controlled by the java security manager, whether there are any holes in the current scope of the java security manager, and consequently whether such security issues pose any threat to the host of a remote object when unaddressed.

2 Aims and Objectives:

2.1 Aim:

Evaluating the security strength of Java RMI.

2.2 Objectives:

The project is broken down in various objectives. These are:

2.2.1 Proposal, Literature Review and Initial Report

Proposal and planning

RMI Technology based literature review and documentation

Reflection & Evaluation of the feedback and implementing any suggestion

2.2.2 Analysis of RMI security aspects

RMI security focused literature review and documentation

Reflection & Evaluation of the feedback and implementing any suggestion

2.2.3 Test bed prototype development

Coding and Testing of a simple RMI based prototype application and GUI layer

An attempt to exploiting security vulnerability by adding malicious code on the Client Side

Documentation

2.2.4 Report Writing

Consolidation of documentations

Reviewing and Critical evaluation: what worked well and what didn't

Evaluation of strengths and weakness

Future works

Tidying up, referencing and finalising report

Task ID	Task Name	Start Date	End Date	Dependencies	Duration	% Complete	Comments
1	**Initial works**						
1.1	Researching ideas and Project selection	Not Applicable				100%	These tasks are now completed.
1.2	Getting supervisor's approval on project idea						
1.3	Hands on RMI technology, fixing & resolving technical issues, research, working on various tutorials, getting a basic working application						
2	**Proposal, Literature Review and Initial Report**	**01/07/14**	**08/07/14**		**9**	**46%**	**Project Title: Investigation of security implications of Java RMI.**

2.1	Proposal and planning	01/07/14	02/07/14		2	100%	
2.2	Submission of proposal	03/07/14	03/07/14	2.1	1	100%	
2.3	RMI Technology based literature review and documentation + upload on Project Portal	04/07/14	07/07/14	2.2	4	30%	

2.4	**Feedback from Supervisor (Afternoon session would be helpful)**	**07/07/14**	**07/07/14**		**1**		**A literature focused feedback at this stage will be regarded very beneficial and would help me:** **1. Identify and close any gaps in RMI technology based research and documentation** **2. Gauge the depth of research and documentation needed for security analysis** **3. Discuss the appropriateness of the planned testbed prototype application**

2.5	Reflection & Evaluation of the feedback and implementing any suggestion	08/07/14	09/07/14		1		
3	**Analysis of RMI security aspects**	**10/07/14**	**28/07/14**		**18**		
3.1	RMI security focused literature review and documentation	10/07/14	20/07/14		11		

3.2	**Feedback from Supervisor**	**21/07/14**	**21/07/14**		**1**		**Need to discuss what exploitation and experiments can be carried out:** **1. What to be achieved?** **2. How to demostrate this achivement?** **3. Revisit the planned testbed prototype application**

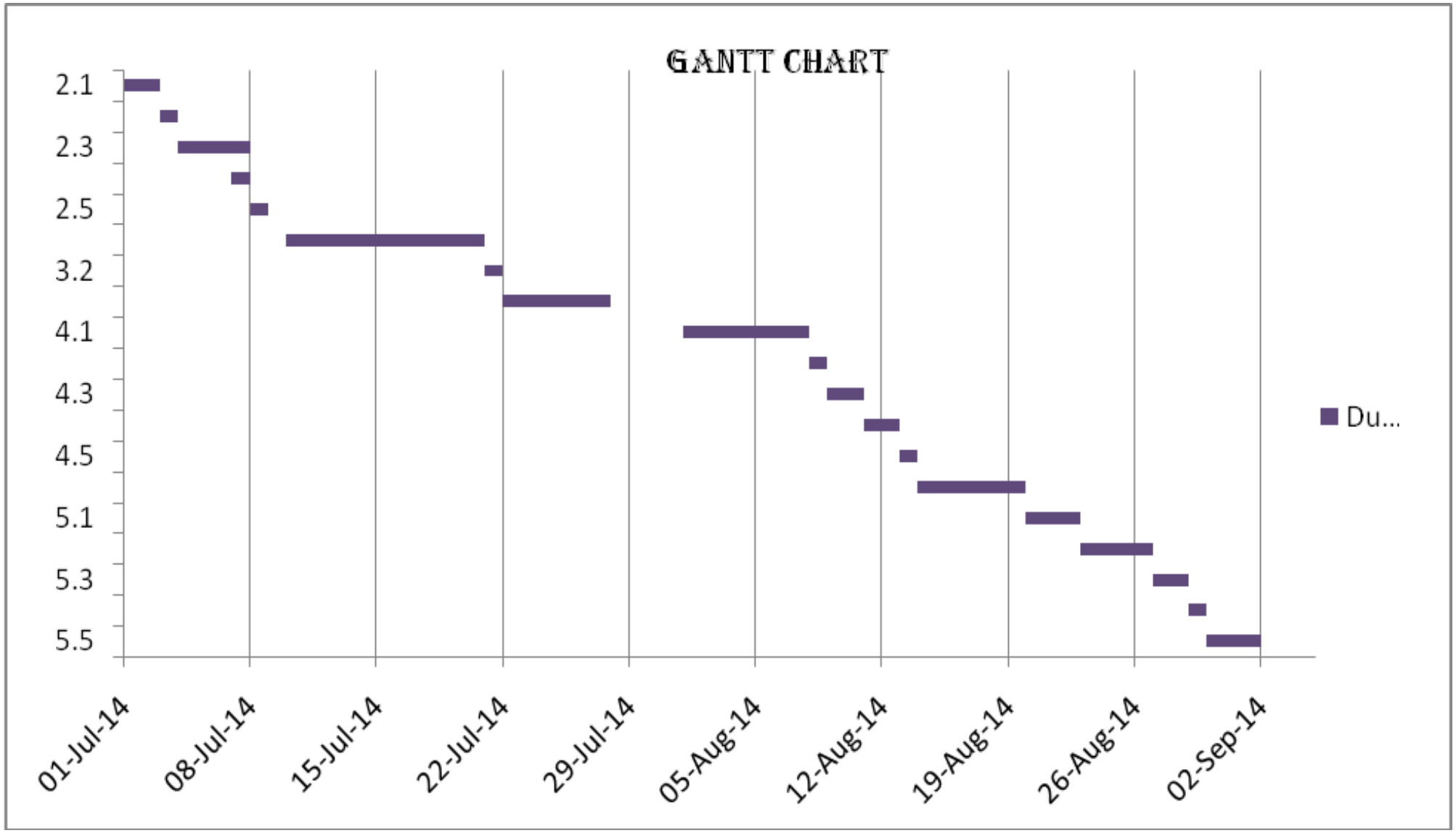
3.3	Reflection & Evaluation of the feedback and implementing any suggestion	22/07/14	28/07/14		6		
4	**Test bed prototype development**	**01/08/14**	**19/08/14**		**20**		
4.1	Prototype Application and GUI layer	01/08/14	07/08/14		7		Currently the plan is to write a simple program that carries out various exhuastive calculations with different number of threads and gauges the performance for lapsed time. This application will measure whether increasing number of threads will increase performance for a given task.

4.2	**Feedback from Supervisor (Afternoon session due to other appointments in the morning)**	**08/08/14**	**08/08/14**		**1**		**Feedback here may be focused on: 1. How the application serves the purpose 2. What can be done to cause a security vulnerability and exploit it**
4.3	Adding security vulnerability code to the Client Side	09/08/14	10/08/14		2		Client application will try to access user areas on the Server side
4.4	Exploitation of security vulnerability	11/08/14	12/08/14		2		

4.5	**Feedback from Supervisor**	**13/08/14**	**13/08/14**		**1**		**Discuss the findings, what and what not went well, how to present this effectively.**
4.6	Documentation	14/08/14	19/08/14		6		
5	**Report Writing**	**20/08/14**	**02/09/14**		**14**		
5.1	Consolidation of documentations	20/08/14	22/08/14		3		
5.2	Reviewing and Critical evaluation: what worked well and what didn't	23/08/14	26/08/14		4		

5.3	Evaluation of strengths and weakness	27/08/14	28/08/14		2		
5.4	Future works	29/08/14	29/08/14		1		
5.5	Tidying up, referencing and finalising report	30/08/14	02/09/14		3		

3 Gantt Chart:



4 Background research:

What is RMI?

Java RMI is a mechanism to invoke a method on a remote object that exists in a different address space. The address that owns the object on which a method is invoked may be on the same machine or different one altogether. RMI can be seen as an object oriented way of RPC.
(Kenneth Baclawski, 1998)

How does RMI compare to RPC?

As Waldo, J. explains in his comparative discussion, Java RMI or Remote Method Invocation is a java based Remote Procedure Call or RPC. This has many features which are similar to other RPC systems, such as allowing an object running in one Java virtual machine or JVM to make a method call on an object running in another JVM, which is often a different physical machine.

Although the RMI system seems like just another RPC mechanism on the surface, with a much closer look, RMI represents a very different evolutionary progression. This progression results in a system that not only differs in detail but also in the very set of assumptions made about the underlying distributed systems it operates in. This leads to differences in the programming model, capabilities, and the interaction mechanisms of the code with its implementation and the built distributed systems.

In RMI, like an RPC system, the proxy on the client (calling) side is known as a stub, while the proxy on the server (receiving) side is known as the skeleton. Stubs are compiled into calling code and render any call into some machine-neutral data representation, this process is called marshalling. The data then gets transmitted to the skeleton on the receiving machine which translates the transmitted information into appropriate data types for that machine; this process is known as unmarshalling. Skeleton identifies the code that needs to be called with the transmitted information, and will make the call. Skeletons also marshal any return values and transmit them back to the stub that made the call, which in turn unmarshals the return values before returning to the calling code.

According to the Oracle's functional definitions of stub and skeleton, when a stub's method is invoked, it performs the following:

- initiates a connection with the remote JVM which contains the remote object,
- marshals, i.e. writes and transmits, the parameters to the remote JVM,
- waits for the result of the method invocation,
- unmarshals, i.e. reads, the return value or any exceptions returned, and
- returns the value to the caller.

On the other side, a skeleton is responsible to dispatch a call to the actual remote object implementation. When a incoming invocation is received by a skeleton, it does the following:

- unmarshals, i.e. reads, the parameters for the remote method,
- invokes the method on the actual remote object implementation, and
- marshals, i.e. writes and transmits, the result, i.e. a return value or exception, to the caller.

Waldo, J. carries on in his paper, the generation of Stub and skeleton in an RPC system happens automatically, based on language and machine-neutral interface-definition language descriptions (IDL) of the calls that can be made. This describes the procedures or methods that may be called over a network and the data that is passed to and from those procedures or methods. With the help of this description, the programmer can invoke an IDL compiler that produces stub and skeleton source code for marshaling, transmitting, and unmarshaling the data. This code is then compiled for the target machine and linked to the relevant application code. In RMI stubs and skeletons are generated by the `rmic` compiler.

While RPC, having a machine-neutral IDL, deal with heterogeneous language, RMI assumes that both the client and server are java classes running in Java Virtual Machine which makes the network a homogeneous collection of machines. The RMI system assumes that all objects forming a distributed system are written in Java. The concept has been made to be single-language assumption by RMI's designers to simplify the overall system. Another benefit of single-language assumption is that the RMI system does not need a language-neutral IDL but simply uses the Java interface construction to declare accessible interfaces remotely.

The Java-centric design of RMI lets the system take advantage of the Java environment's dynamic nature, allowing code to load any time during execution. While in RPC systems all code needed for communication between processes have to be available at some time prior to that communication, RMI makes extensive use of dynamic code loading, from stubs representing remote objects to real objects that can be passed from one part of a distributed system to another. The ability of downloading code between machines plays the most prominent role to pass full objects into and out of an RMI call. RMI's single-language assumption allows almost any Java object to pass as a parameter or return value in a remote call. Remote objects are passed by reference by passing a copy of the object's stub code whereas non-remote objects are passed by value by creating a copy of the object in the destination.

The objects that are passed through the network in an RMI system are real objects, not just data that makes up the object's state. This distinction becomes obvious when a subtype of a declared type passes from distributed computation member to another. Passing a subtype object could result in an unknown object being returned. As a subtype can change the behaviour of known methods in an object, simply treating the object as an instance of a known type might change the results of making a method call on that object. To avoid such changes, RMI uses a Java Object Serialization package to marshal and reconstruct objects. This package annotates any object with sufficient information to identify the object's exact type and its implementation code. So when an object of a previously unknown type is received on an RMI call, the system fetches the code for that object, verifies it, and dynamically loads it into the receiving process. In RMI system, therefore, distributed computations can use all the standard object-oriented design patterns that rely on polymorphism. On the contrary, RPC system does not allow polymorphism; it means that the transmitted object's type or its reference type cannot be a subtype of the type expected by the skeleton.

In the RMI system, a remote object's stub is originated with the object and can be different for any two objects with the same apparent type. The system locates and loads these stubs at runtime as it determines what the exact stub type is. This association makes the stub an extension of the remote object in the client's address space, rather than some way of contacting the remote object built into the client. This approach allows programmers to build a variety of "smart" proxies.

Technical implementation of RMI:

According to Baclawski (1998), RMI employs three processes to support remote invocation. These are:

1. Client process: this is the process that invokes method on a remote object.
2. Server process: this process owns the remote object. The remote object is an ordinary object that resides in the address space of the server process.
3. Object Registry: this is a name server that relates objects to their names. Once a remote object is registered in the Object Registry, this can be accessed by a client process using the name of the object.

There are two types of classes used in java RMI, namely a Remote class and a Serializable class.

A Remote class is a class whose instances are used remotely. Such objects are sometime referred to as *remote object* and can be referenced in two ways:

i. Like any ordinary object within the address space where the object is constructed.

ii. Within a remote address spaces using *object handle*. There are some limitations on how an *object handle* can be used but mostly it can be used the same way as an ordinary object.

A Serializable class is a class whose instances can be copied from one address space to another. An instance of this class is often referred to as *serializable object*.

When a *serializable object* is passed as a parameter or return value of a remote method invocation, the value of the object is copied from one address space to the other. However, in such case for a *remote object*, the *object handle* is copied from one address space to the other. This begs the question, what happens when a class is both *Remote* and *Serializable*? A blog in StackOverflow (June 2011) addresses the answer to this as the object is serialized instead of being passed as a remote reference provided that the class of the object implements both the *Remote* and *Serializable*. Although this may be possible in theory, it is a poor design to mix the two notions and may make the design difficult to understand.

Implementing security:

Note that both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one

machine to the other. Such a download could violate system security. This section discusses java security model as it relates to RMI.

As Baclawski (1998) continues in his article, a program written in java can specify a security manager to determine its security policy. A program usually needs a security manager to be specified. The Security policy is set by constructing a SecurityManager object and calling the *setSecurityManager* method of the System class. Some operations require a security manager. For example, RMI will only download a Serializable class from another machine if there is a defined security manager which allows the downloading of the respecting class from the machine. The RMISecurityManager class defines an example of a security manager that normally allows such downloads.

The SecurityManager leverages a large number of methods with names beginning with check. If a check method returns, then the permission was granted. For example, a valid return from a call to *checkConnect* with a specified host and port means that the current security policy allows the program to establish a socket connection to the server socket at the specified host and port. If the security policy does not allow one to connect to this host and port, then the call throws an exception.

The original technique for specifying a security policy in Java was to define and install a security manager. Unfortunately, it is very difficult to design a class that does not leave any security holes and as a result of that a new technique was introduced in Java 1.2, which is said to be backward compatible with the old technique. In the default security manager, all check methods except checkPermission are implemented by calling the checkPermission method. A parameter is passed to the checkPermission method to specify the type of permission being checked. For example, the checkConnectmethod calls checkPermission with a SocketPermission object. The default implementation of checkPermission calls the checkPermission method of the AccessController class. This method verifies whether the specified permission is implied by a list of granted permissions. The Permissions class is used to maintain lists of granted permissions and for checking whether a particular permission has been granted.

While the security manager is a mechanism that checks permissions, it does not explain how one specifies or changes the security policy. This is done through another class named Policy. Like SecurityManager, each program has a current security policy that can be obtained by calling *Policy.getPolicy()*, and the current security policy can be set using *Policy.setPolicy*, providing one has permission to do so. The security policy is generally specified by a policy configuration file, otherwise known as "policy file", which is read by the program when it starts and any time a request is made to refresh the security policy.

5 References and Keywords:

5.1 References:

a) Waldo, J., "Remote procedure calls and Java Remote Method Invocation," Concurrency, IEEE , vol.6, no.3, pp.5,7, Jul-Sep 1998. [online].URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=708248&isnumber=15346.

b) Kenneth Baclawski(1998). Java RMI Tutorial. [online]. Available from: http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html. [Accessed: 18/11/2013].

c) Docs.oracle.com. Stubs and Skeletons. [online]. Available from: http://docs.oracle.com/javase/8/docs/platform/rmi/spec/rmi-arch2.html.

d) StackOverflow (June 2011). Java RMI, making an object serializeable AND remote. [online]. Available from: http://stackoverflow.com/questions/6268435/java-rmi-making-an-object-serializeable-and-remote. [Accessed: 12/12/2013].

e) Wikipedia. Java remote method invocation. [online]. Available from: http://en.wikipedia.org/wiki/Java_remote_method_invocation. [Accessed: 12/12/2013].

f) Docs.oracle.com. Java Remote Method Invocation: 10 - RMI Wire Protocol. [online]. Available from: http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-protocol7.html. [online].

5.2 Keywords:

RMI – Remote Method Invocation

RPC – Remote Procedure Call

Remote class

Serializable class

object-oriented languages

object-oriented programming

programming environments

remote procedure calls

Common Object Request Broker Architecture

Corba

DCOM

Distributed Common Object Model

Java Developer Kit

Java Remote Method Invocation

Java virtual machine

NCS

RMI system

RPC mechanism

RPC systems

distributed applications

distributed systems method call

remote procedure call systems

Instruction sets

Java

Operating systems

Production

Program processors

Programming profession

Skeleton

Sun

Virtual machining