**COMP[1298] MSc Final Year Project Proposal**

**Security Implication in Java RMI**

**Mahabubur Rashid**

**MSc Computer Forensics and System Security**

**000582762**

**Contents**

[1 Overview: 3](#_Toc375059748)

[2 Aims and Objectives: 4](#_Toc375059749)

[2.1 Aim: 4](#_Toc375059750)

[2.2 Objectives: 4](#_Toc375059751)

[2.2.1 Proposal, Literature Review and Initial Report 4](#_Toc375059752)

[2.2.2 Application Development 4](#_Toc375059753)

[2.2.3 Analysis of security aspects 4](#_Toc375059754)

[2.2.4 Report Writing 4](#_Toc375059755)

[3 Gantt Chart: 6](#_Toc375059756)

[4 Review of documentation: 7](#_Toc375059757)

[5 Conclusion: 11](#_Toc375059758)

[6 References and Keywords: 12](#_Toc375059759)

[6.1 References: 12](#_Toc375059760)

[6.2 Keywords: 12](#_Toc375059761)

# 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 how these security issues pose any threat to the host of a remote object when unaddressed.

# Aims and Objectives:

## Aim:

Investigating the security implication in java RMI and how these can be exploited to gain access to a system.

## Objectives:

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

### Proposal, Literature Review and Initial Report

Proposal and planning

Initial literature review and documentation

### Application Development

Coding and Testing of a simple application

Meeting supervisor for feedback on application

Enhancement and extension coupled with testing

Documentation

### Analysis of security aspects

Literature review and documentation

Exploitation of security vulnerability

Documentation

### 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 | Researching ideas and project selection | 23/09/13 | 30/11/13 |  | 20 | 100% |  |
| 1.1 | Exploring security issues in Mobile platform | 23/09/13 | 03/11/13 |  | 13 | 100% |  |
| 1.2 | Investigating project ideas in other areas | 09/11/13 | 24/11/13 |  | 6 | 100% |  |
| 1.3 | Project selection | 30/11/13 | 30/11/13 | 1.1, 1.2 | 1 | 100% |  |
| 2 | Proposal, Literature Review and Initial Report | 25/11/13 | 22/12/13 |  | 9 | 29% | Project Title: Investigation of security implications of Java RMI. |
| 2.1 | Proposal and planning | 15/12/13 | 21/12/13 | 1.3FS +4 days | 2 | 40% |  |
| 2.2 | Meeting supervisor for feedback | 18/12/13 | 18/12/13 |  | 1 |  |  |
| 2.3 | Initial literature review and documentation | 25/11/13 | 22/12/13 |  | 9 | 30% |  |
| 3 | Application Development | 23/12/13 | 02/03/14 |  | 21 |  |  |
| 3.1 | Coding and Testing of a simple application | 23/12/13 | 18/01/14 |  | 8 |  |  |
| 3.2 | Meeting supervisor for feedback on application | 13/01/14 | 13/01/14 |  | 1 |  |  |
| 3.3 | Enhancement and extension coupled with testing | 17/01/14 | 09/02/14 |  | 9 |  |  |
| 3.4 | Documentation | 15/02/14 | 02/03/14 |  | 6 |  |  |
| 4 | Analysis of security aspects | 08/03/14 | 13/04/14 |  | 12 |  |  |
| 4.1 | Literature review and documentation | 08/03/14 | 16/03/14 |  | 4 |  |  |
| 4.2 | Exploitation of security vulnerability | 22/03/14 | 30/03/14 |  | 4 |  |  |
| 4.3 | Documentation | 05/04/14 | 13/04/14 |  | 4 |  |  |
| 5 | Report Writing | 15/04/14 | 18/05/14 |  | 11 |  |  |
| 5.1 | Consolidation of documentations | 15/04/14 | 20/04/14 |  | 3 |  |  |
| 5.2 | Reviewing and Critical evaluation: what worked well and what didn't | 26/04/14 | 27/04/14 |  | 2 |  |  |
| 5.3 | Evaluation of strengths and weakness | 03/05/14 | 03/05/14 |  | 1 |  |  |
| 5.4 | Future works | 04/05/14 | 04/05/14 |  | 1 |  |  |
| 5.5 | Tidying up, referencing and finalising report | 10/05/14 | 18/05/14 |  | 4 |  |  |

# Gantt Chart:

# Review of documentation:

**What is RMI?**

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

**How does RMI work?**

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:

1. Like any ordinary object within the address space where the object is constructed.
2. 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*? The answer to this is 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.

(Kenneth Baclawski, 1998; StackOverflow, June 2011)

**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.

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 class has a large number of methods whose name begins with check. For example, checkConnect (String host, int port). If a check method returns, then the permission was granted. For example, if a call to checkConnect returns normally, then the current security policy allows the program to establish a socket connection to the server socket at the specified host and port. If the current security policy does not allow one to connect to this host and port, then the call throws an exception.*

*Defining and installing a security manager was the original technique for specifying a security policy in Java. Unfortunately, it is very difficult to design such a class so that it does not leave any security holes. For this reason, a new technique was introduced in Java 1.2, which is backward compatible with the old technique. In the default security manager, all check methods (except checkPermission) are implemented by calling the checkPermission method. The type of permission being checked is specified by the parameter of type Permission passed to the checkPermission method. For example, the checkConnectmethod calls checkPermission with a SocketPermission object. The default implementation of checkPermission is to call the checkPermission method of the AccessController class. This method checks whether the specified permission is implied by a list of granted permissions. The Permissions class is used for maintaining lists of granted permissions and for checking whether a particular permission has been granted.*

*This is the mechanism whereby the security manager checks permissions, but it does not explain how one specifies or changes the security policy. For this purpose there is yet another class, named Policy. Like SecurityManager, each program has a current security policy that can be obtained by calling Policy.getPolicy(), and one can set the current security policy using Policy.setPolicy, if one has permission to do so. The security policy is typically specified by a policy configuration file (or "policy file" for short) which is read when the program starts and any time that a request is made to refresh the security policy. The policy file defines the permissions contained in a Policy object.*

***RMI's Multiplexing Protocol***

*The purpose of multiplexing is to provide a model where two endpoints can each open multiple full duplex connections to the other endpoint in an environment where only one of the endpoints is able to open such a bidirectional connection using some other facility (e.g., a TCP connection). RMI use this simple multiplexing protocol to allow a client to connect to an RMI server object in some situations where that is otherwise not possible. For example, some security managers for applet environments disallow the creation of server sockets to listen for incoming connections, thereby preventing such applets to export RMI objects and service remote calls from direct socket connections. If the applet can open a normal socket connection to its codebase host, however, then it can use the multiplexing protocol over that connection to allow the codebase host to invoke methods on RMI objects exported by the applet. This section describes how the format and rules of the multiplexing protocol.*

***Definitions***

*This sections defines some terms as they are used in the rest of the description of the protocol.*

*An endpoint is one of the two users of a connection using the multiplexing protocol.*

*The multiplexing protocol must layer on top of one existing bidirectional, reliable byte stream, presumably initiated by one of the endpoints to the other. In current RMI usage, this is always a TCP connection, made with a java.net.Socket object. This connection will be referred to as the concrete connection.*

*The multiplexing protocol facilitates the use of virtual connections, which are themselves bidirectional, reliable byte streams, representing a particular session between two endpoints. The set of virtual connections between two endpoints over a single concrete connection comprises a multiplexed connection. Using the multiplexing protocol, virtual connections can be opened and closed by either endpoint. The state of an virtual connection with respect to a given endpoint is defined by the elements of the multiplexing protocol that are sent and received over the concrete connection. Such state involves if the connection is open or closed, the actual data that has been transmitted across, and the related flow control mechanisms. If not otherwise qualified, the term connection used in the remainder of this section means virtual connection.*

*A virtual connections within a given multiplexed connection is identified by a 16 bit integer, known as the connection identifier. Thus, there exist 65,536 possible virtual connections in one multiplexed connection. The implementation may limit the number of these virtual connections that may be used simultaneously.*

*Connection State and Flow Control*

*Connections are manipulated using the various operations defined by the multiplexing protocol. The following are the names of the operations defined by the protocol: OPEN, CLOSE, CLOSEACK, REQUEST, and TRANSMIT. The exact format and rules for all the operations are detailed in*[*Section 10.6.3, "Protocol Format*](http://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-protocol7.html#3494)*."*

*The OPEN, CLOSE, and CLOSEACK operations control connections becoming opened and closed, while the REQUEST and TRANSMIT operations are used to transmit data across an open connection within the constraints of the flow control mechanism.*

***Connection States***

*A virtual connection is open with respect to a particular endpoint if the endpoint has sent an OPEN operation for that connection, or it has received an OPEN operation for that connection (and it had not been subsequently closed). The various protocol operations are described below.*

*A virtual connection is pending close with respect to a particular endpoint if the endpoint has sent a CLOSE operation for that connection, but it has not yet received a subsequent CLOSE or CLOSEACK operation for that connection.*

*A virtual connection is closed with respect to a particular endpoint if it has never been opened, or if it has received a CLOSE or a CLOSEACK operation for that connection (and it has not been subsequently opened).*

***Flow Control***

*The multiplexing protocol uses a simple packeting flow control mechanism to allow multiple virtual connections to exist in parallel over the same concrete connection. The high level requirement of the flow control mechanism is that the state of all virtual connections is independent; the state of one connection may not affect the behavior of others. For example, if the data buffers handling data coming in from one connection become full, this cannot prevent the transmission and processing of data for any other connection. This is necessary if the continuation of one connection is dependent on the completion of the use of another connection, such as would happen with recursive RMI calls. Therefore, the practical implication is that the implementation must always be able to consume and process all of the multiplexing protocol data ready for input on the concrete connection (assuming that it conforms to this specification).*

*Each endpoint has two state values associated with each connection: how many bytes of data the endpoint has requested but not received (input request count) and how many bytes the other endpoint has requested but have not been supplied by this endpoint (output request count).*

*An endpoint's output request count is increased when it receives a REQUEST operation from the other endpoint, and it is decreased when it sends a TRANSMIT operation. An endpoint's input request count is increased when it sends a REQUEST operation, and it is decreased when it receives a TRANSMIT operation. It is a protocol violation if either of these values becomes negative.*

*It is a protocol violation for an endpoint to send a REQUEST operation that would increase its input request count to more bytes that it can currently handle without blocking. It should, however, make sure that its input request count is greater than zero if the user of the connection is waiting to read data.*

*It is a protocol violation for an endpoint to send a TRANSMIT operation containing more bytes that its output request count. It may buffer outgoing data until the user of the connection requests that data written to the connection be explicitly flushed. If data must be sent over the connection, however, by either an explicit flush or because the implementation's output buffers are full, then the user of the connection may be blocked until sufficient TRANSMIT operations can proceed.*

*Beyond the rules outlined above, implementations are free to send REQUEST and TRANSMIT operations as deemed appropriate. For example, an endpoint may request more data for a connection even if its input buffer is not empty.*

# Conclusion:

Some conclusion here, don’t know yet what it should be.

# References and Keywords:

## References:

1. Kenneth Baclawski(1998). *Java RMI Tutorial*. [online]. Available from: <http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html>. [Accessed: 18/11/2013].
2. 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].
3. Wikipedia. *Java remote method invocation*. [online]. Available from: <http://en.wikipedia.org/wiki/Java_remote_method_invocation>. [Accessed: 12/12/2013].
4. 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].

## Keywords:

RMI – Remote Method Invocation

RPC – Remote Procedure Call

Remote class

Serializable class