COMP[1298] MSc Final Year Project
Initial Report

Security Implication in Java RMI

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

2 Technical background:

2.1 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)

2.2 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.

2.3 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.

2.4 Implementing security manager:

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.

3 Security Analysis of RMI technology

3.1 Case study -1: security concerns in use of proxy:

As described by Ninghui, Mitchell and Tong (Dec. 2004) in their paper titled 'Securing Java RMI-based Distributed Applications' on Computer Security Application Conference, in proxy-based architecture of Java RMI, a client interacts with a service through a proxy. This proxy is essentially some java code downloaded from a directory and installed on the client's machine. In case of critical systems built using the proxy-based architecture, the security of these systems is a natural concern. The use of proxies introduces security vulnerabilities that need to be dealt with before any system using technologies such as Java RMI can be relied upon.

An attacker may control the communication channels and may compromise the confidentiality and integrity of the client and of the service.

The use of proxies poses several challenges in the provision of security mechanisms for proxy-based systems. The job of providing authentication between client and service applications is much complicated by the use of a proxy. A proxy is neither fully controlled nor trusted by the either of the client or service and yet the communication is made through this proxy. A proxy is generated by the service and thus the client cannot trust the proxy more than it trusts the service. On the other hand, this proxy is executed by the client under its control and thus the service cannot fully trust the proxy either. One approach to resolving this lack of trusts may be to treat the proxy as part of an insecure communication channel that connects the client and the server, and to implement standard authentication techniques to provide mutual authentication between the client and the server.

A graphical representation of the interactions among the service, client, and proxy is depicted in Figure 1. Three virtual machines (VMs) are involved in these process: the client VM, the service VM, and the directory VM.

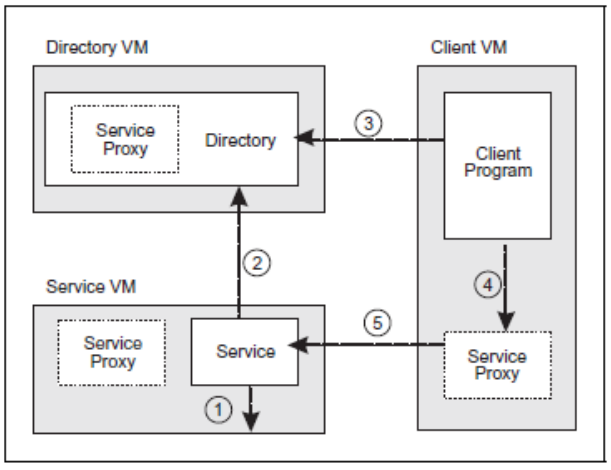


Figure 1. A normal client/service interaction

The whole process is done through the following steps:

In step 1 the service makes a call to the RMI runtime to export the service. This process creates a proxy for the service in the service VM.

In step 2 the service makes a remote call to a directory (e.g. rmiregistry) to register its proxy. With this process the directory VM gets the proxy object.

In step 3 the client looks up the service and downloads the proxy from the directory. This process enables the client VM to have the proxy.

Following the above steps, the client calls the proxy locally each time it needs to use the service as if it is the remote service. The proxy then communicates the client's request to the service to serve the desired outcome. Once the client has located and installed a proxy, it can reuse the proxy to make further requests as many times as needed as long as the proxy is considered valid by the service.

3.1.1 Vulnerability analysis in light of CIA:

In this vulnerability analysis for proxy-based systems, three parties are considered in the process which are namely the user, the service provider, and the attacker. It is assumed that the user has full control over the client VM and the service provider has full control over the service VM. An attacker, however, may control the network and the directory VM, in other words, the directory is considered as untrusted.

The risks it poses to the client are:

3.1.1.1 Client confidentiality:

The client may transfer sensitive information to the proxy when using the service. One example of this is that if the service provides access to the user's investment accounts, the user would want to make sure that his information and instructions are only sent to the service provider, not to anyone else.

Another way that the attacker may breach client's confidentiality is by eavesdropping on the communications channel between the service and the proxy or between the client and the proxy.

3.1.1.2 Client integrity:

The user's VM and local environment may be inflicted by the proxy. Thus the client's integrity may be compromised by the attacker by getting the client to install and run a bogus proxy either by compromising the directory or by actively attacking the communication channel between the directory VM and the client VM or the channel between the directory VM and the service VM.

This type of attack can also pose risks to the service and like those on the client the service's confidentiality and integrity as well as the service's availability is affected by the attack.

3.1.1.3 Service confidentiality:

The service may send sensitive information to the client through the proxy. An attacker who gained control over the directory VM or the communication channel and thus have access to this proxy can extract this information and compromise the service's confidentiality.

3.1.1.4 Service integrity:

Sometimes a service may need to perform certain tasks on behalf of its client, and an attacker who successfully impersonates a client may result in damage or malicious use. The attacker can also compromise the service confidentiality and integrity by controlling the communication between the proxy and the service or by convincing the client to accept a bogus proxy.

An example of such attack where an attacker impersonates a client is demonstrated in section 4.

3.1.1.5 Service availability:

When an attacker gains control over the directory VM, they can manipulate the service proxy such that the service is interrupted or denied. The client, being unaware of the attack, installs the proxy on his machine and calls the proxy with expectation of receiving the service when the service is denied. Thus the server suffers a denial-of-service attack by the attacker who controls the communication channels and/or the directory.

3.2 Case study -2: security concerns in serializing data in RMI application:

4 Test application - Breach of confidentiality:

4.1 Overview:

This RMI application consists of a server application, a client application and a remote interface. The server provides a service that receives various "jobs" from the client in order to run them on the server machine; this facilitates the client with the ability to perform a job that requires heavy resources that the client cannot provide. Once the job is performed on the server side the client can then invoke a remote method, through a proxy generated by the server, to retrieve the result of the job.

4.2 Application components:

The application has the following components:

4.2.1 Job Interface:

This is an interface that defines a job. A client-side class `implements` this interface to mark the class to be recognized as a job on the server-side. The result is returned to the client in the form of a `generic type` which is determined by the client-side class that defines the job. The interface looks like this:

```
public interface JobInterface<T> {
  T commenceJob();
}
```

Once a job is successfully instantiated on the client-side, it is then passed to the server to be performed through a proxy generated by the server application.

4.2.2 Remote Interface:

This interface is a subclass of `java.rmi.Remote` which makes it 'the Remote interface' for this RMI application. A server-side class `implements` this interface to provide the intended service. The interface is defined as below:

```
public interface RemoteJobInterface extends java.rmi.Remote {
  <T> T retrieveJobReport(JobInterface<T> job) throws RemoteException;
}
```

The interface is used for storing a `remote object stub` on the server-side application which is eventually bound to the `rmiregistry`. The client application, on the other hand, retrieves the `remote object stub` from the `rmiregistry` and uses the interface to `cast` the `stub` to a remote-interface object understandable by the client. The client then passes its job (i.e. a `JobInterface` object) to the server as parameter and retrieves the result by invoking the remote method (i.e. `retrieveJobReport` method) on this remote-interface object.

4.2.3 Server-side implementation:

The service provided by the server-side class, namely `RemoteJobExecutor`, receives a job from the client through remote method invocation, carries out the job on the server machine and then returns the result back to the client. The class provides its own implementation of the method signature defined in the remote-interface, `RemoteJobInterface`, to perform the task.

```
public class RemoteJobExecutor implements RemoteJobInterface {

@Override
 public <T> T retrieveJobReport(JobInterface<T> job) {
        …
            …
        return job.commenceJob();
 }
}
```

The method `retrieveJobReport` receives a `JobInterface` object by the name `job` and calls a `commenceJob` method on this object, which is implemented on the client side.

A program called `ServerSideProgram` triggers the operation by first creating an object of the service instance (named `jobExecutor`) and exporting it to the Java RMI runtime so that it may receive incoming remote calls.

```
RemoteJobInterface stub = (RemoteJobInterface) UnicastRemoteObject
                          .exportObject(jobExecutor, 0);
```

The process of exporting the object returns a remote object stub which is stored in a `RemoteJobInterface` variable called `stub`. Following this, an instance of Java RMI registry, called `registry`, is created and exported on the local host, port 1099:

```
Registry registry = LocateRegistry.createRegistry(1099);
```

Finally, `stub` is bound to `registry` to bind the remote object's stub to the name "RemoteJobExecutor" in the registry:

```
registry.rebind("RemoteJobExecutor", stub);
```

4.2.4 Client-side implementation:

The client-side defines a class called that turns a plain textual message into an encrypted message using a *symmetric encryption* method and *encryption key*. The class implements the interface `JobInterface` which makes it a “job” recognized by the service on the server-side. The encryption method is an enumeration type which is one of SUBSTITUTION, TRANSPOSITION or BLOCKING. The encryption method is selected by the user at the time of creating an instance of the job before passing it onto the server-side. In order to be transported over a network, the class also implements `Serializable`.

```
public class SymmetricEncryption implements JobInterface<String>, Serializable
{
```

```
public SymmetricEncryption(String message,
                SymEncryptionMethod encryptionMethod, int key) {…}

@Override
  public String commenceJob() {…}
}
```

A program on the client-side, named `ClientSideProgram`, first gets hold of the registry and looks up for the remote object reference by its name, i.e. "RemoteJobExecutor", which is stored in `RemoteJobInterface` variable called `jobExecutor`.

```
Registry registry = LocateRegistry.getRegistry();
RemoteJobInterface jobExecutor = (RemoteJobInterface) registry
                            .lookup("RemoteJobExecutor");
```

It then creates an object of `SymmetricEncryption` "job" and finally passes the "job" to the service by invoking a remote method on the remote object `jobExecutor`.

```
SymmetricEncryption symetricEncryp = new SymmetricEncryption("Hello RMI
application.", SymEncryptionMethod.SUBSTITUTION, 5);

String jobReport = jobExecutor.retrieveJobReport(symetricEncryp);
```

4.3 Things to take care of before running the service:

Before the service is started it needs to have prior knowledge of the classes it may receive from the client or clients. This could be achieved in the following ways:

- The service, when starting, has to have the clients' class files in its `classpath` when both the service and client are on the same machine, e.g. local host.
- The relevant class files can be placed in a designated directory and `Java RMI Codebase feature` may be used to point the service to this directory at service start time. Again, this works when the service and client are run on the same machine.
- In case of the service and client being on different remote machines, the clients' class files may be placed in a web server and the service would locate these files using http in the `Java RMI Codebase feature`.

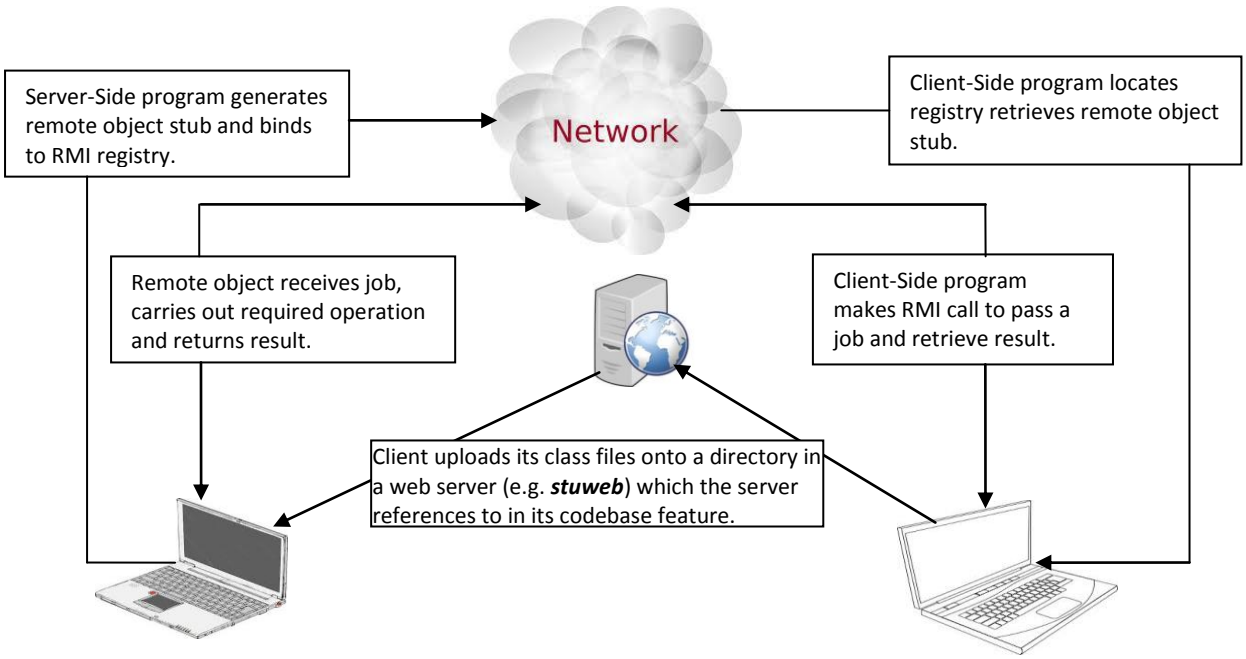
Without the step outlined above, the service will start up without any error; however, running the client application will throw an exception which will look like:

```
java.rmi.ServerException: RemoteException occurred in server thread; nested
exception is:
  java.rmi.UnmarshalException: error unmarshalling arguments; nested exception
is:
  java.lang.ClassNotFoundException:
uk.ac.rm950.breachOfConfidentialityClient.SymmetricEncryption
  at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:353)
…
…
java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(RemoteObjectI
nvocationHandler.java:194)
Remote job client exception.
```

4.4 Application in action:

The `ServerSideProgram` initiates the service on machine A by creating an instance of the remote object, generating a stub and finally starting the RMI registry and binding the stub with this registry. Once the service is up and running, a `ClientSideProgram` on machine B then locates the registry and looks up the remote-object-reference in the registry. The remote method is then invoked on the remote-object-reference by the client program to pass the job to the service and to retrieve the result.

In this example, the `ClientSideProgram` creates a `SymmetricEncryption` job with message *"Hello RMI application"*, symmetric method SUBSTITUTION, and encryption key 5. Upon invoking the remote method `retrieveJobReport`, the client passes the job to the server-side and receives the result in the form of `String`.



As the client application is run, the server-side console outputs:

```
Server side received a remote job from client machine: 192.168.0.7
The job is being carried out on server machine: mahabuburs-mbp/192.168.0.5
```

On the client-side, the console displays the result and outputs:

```
Client side:
================
Original message: Hello RMI application.
Encrypted message: Mjqqt%WRN%fuuqnhfynts3

The job was executed in machine: mahabuburs-mbp/192.168.0.5
```

4.5 Attack through impersonating a client:

We presume that an attacker gains access to the directory in the web server (***stuweb*** in this example) where the RMI service looks into for reference to client classes. The attacker then creates an evil version of the `SymmetricEncryption` job; let us call this class `SymmetricEncryptionEvil`. The class contains malicious code which, outside of doing the normal symmetric encryption operation, will try to steal password information from the server machine by reading the *passwd* file.

Once the *evil* class is updated on the ***stuweb*** web server, the attacker impersonates a legitimate client to create a job with the class and makes an RMI call. This RMI call passes the job to the server-side, executes the operation with the malicious code and the stolen data is then returned to the client (i.e. the attacker in this case) with the job report.

The `SymmetricEncryptionEvil` class may look like this:

```
public class SymmetricEncryptionEvil implements JobInterface<String>,
Serializable {

   public SymmetricEncryptionEvil(String message,
              SymEncryptionMethod encryptionMethod, int key) {…}

   @Override
   public String commenceJob() {
   …
   …
   return String.format("Original message: %s\nEncrypted message: %s"
                           + "\n\nThe job was executed in machine: %s."
                           + "\n\nFinally, this is the stolen data: %s",
                        originalMessage,encryptedMessage, getIPAddress(),
                           stealFileContent("less -N /etc/passwd"));
   }
}
```

As the client program is run the server-side console as usually outputs:

```
Sever side:
==============
RemoteJobExecutor bound
Server side received a remote job from client machine: 192.168.0.7
The job is being carried out on server machine: mahabuburs-mbp/192.168.0.5
```

And to the attacker's satisfaction, the client-side (i.e. attacker's) console displays:

```
Client side:
================
Original message: Hello RMI application.
Encrypted message: Mjqqt%WRN%fuuqnhfynts3

The job was executed in machine: mahabuburs-mbp/192.168.0.5.
Finally, this is the stolen data:
nob
roo
dae
_uu                                                          bin/uucico
_ta
```

5 Test application -:

6 Reflection:

Still to write

7 Conclusion:

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

8 References and Keywords:

8.1 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]. Available from: 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.

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.

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

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.

g) Ninghui Li; Mitchell, J.C.; Tong, D., "Securing Java RMI-based distributed applications," *Computer Security Applications Conference, 2004. 20th Annual* , vol., no., pp.262,271, 6-10 Dec. 2004, doi: 10.1109/CSAC.2004.34. [online]. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1377233&isnumber=30059.

h)

8.2 Keywords:

a) RMI – Remote Method Invocation; RPC – Remote Procedure Call; Remote class; Serializable class; object-oriented languages; object-oriented programming; programming environments; remote procedure calls; Distributed Common Object Model; Java Remote Method Invocation; Java virtual machine; RMI system; RPC mechanism; RPC systems; distributed applications; distributed systems method call; remote procedure call systems; Java; Operating systems; Skeleton; client-server system; message authentication;