COMP[1298] MSc Final Year Project
Interim Report

Security Implication in Java RMI

Mahabubur Rashid

MSc Computer Forensics and System Security

000582762

Contents

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.
(Baclawski, 1998)

2.2 How does RMI compare to RPC?

As Waldo (1998) 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 (1998) 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 Li, Mitchell and Tong (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.

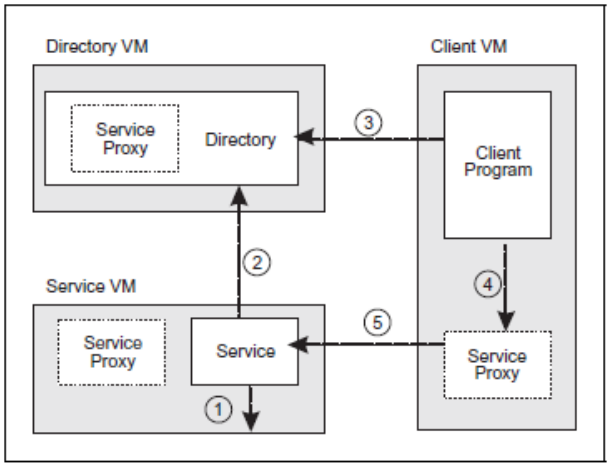


Figure 1. A normal client/service interaction

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:

Java is generally considered to be a safe language with good security features. However, some characteristic and features of Java may pose security threat and compromise safety if they are misused or incorrectly implemented. One such Java feature is *Serialization*.

W. Long (2005) of Carnegie Mellon University documented the issues with serialization in context of software security in his research showcase titled "Software Vulnerabilities in Java". This has been discussed in the following paragraphs.

3.2.1 How does Serialization work?

Serialization in a Java program enables the state of the program to be captured and preserved by writing it out to a byte stream. The preserved state can be later reinstated by another program as needed which is known as *deserialization*.

In order for a class to facilitate *Serialization*, the class needs to implement `Serializable` interface. Once the interface is implemented, all the fields of the class are captured. This includes non-public fields that would have been normally inaccessible, unless the fields are declared `transient` in which case they are not serialized.

For example, if we have an object called `someObject`, it can be serialized as follows:

```
ObjectOutputStream out = new ObjectOutputStream (newFileOutputStream
(“SerialOutput”));
out.writeObject(someObject);
out.flush();
```

The serialized object can then be deserialized as follows:

```
ObjectInputStreamo in = new ObjectInputStream (newFileInputStream
(“SerialOutput”));
someObject = (SomeClass) in.readObject();
```

3.2.2 How Serialization may affect in RMI applications:

In RMI application, a remote object (server-side) is exported to the Java RMI runtime so that it may receive Remote Method Invocation from the client-side. This is done by calling a *static* `exportObject` method on `UnicastRemoteObject` class. The process generates a `Stub` which is then bound to the RMI registry with a name in `String` form and obtained by the client via a lookup. Here’s an example:

```
RemoteInterface stub = (RemoteInterface) UnicastRemoteObject
                              .exportObject(remoteObject);
rmiRegistry.rebind("name", stub);
```

A generated Stub created using above method is a serialized object; this facilitates a Java method call to be transmitted over a network. However, this serialized object is written into byte stream and if the byte stream is readable then the values of the normally inaccessible fields can be read. Furthermore, it may also be possible to modify or forge the preserved values with the intention that when the object is deserialized, the values are corrupted.

Implementing a security manager in RMI application does not prevent the non-public fields from being serialized and deserialized (although permission must be granted to write to and read from a file or network if the byte stream is being stored or transmitted).

An example program demonstrating the issue of serializing non-public fields is given in section 5.

3.3 Case study -3: security concerns posed by lack of verification of type compatibility in RMI protocol implementation:

Vulnerabilities in Java are usually associated with the risk they pose to users of various web browsers. That’s completely natural taking into account the widespread use of Java Plugin software. There are however some other exploitation scenarios that are worth mentioning. This in particular concerns the possibility to exploit Java security issues on servers. Below, we present the idea behind two such scenarios that could facilitate the attack against server side Java software.

3.4.1 RMI protocol attack

RMI protocol[12] is the base protocol used for communication between clients and servers during Java Remote Method Invocation[13]. RMI protocol implementation supports the concept

of user provided codebases. A codebase is the URL value pointing to the remote resource where remote RMI server should look for unknown (non-system) classes. What's interesting is that Codebase URL can be provided by the RMI client as part of the RMI call. It will be taken into account by the RMI server if `java.rmi.server.useCodebaseOnly` *property is set to true. If true, RMI server will create* `RMIClassLoader` *instance with user provided Codebase URL. It will be further used as a base class loader during object deserialization by a* `MarshalInputStream`*.*

RMI implementation does not verify whether a deserialized object is type compatible with the input argument of a target method call. RMI server reads and instantiates object provided as an argument to the call with the use of `RMIClassLoader`*. If the object to read is of an unknown class, an attempt will be made to fetch class data from the Codebase URL provided by the user. That alone creates a possibility for remote loading and execution of user provided Java code.*

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`.

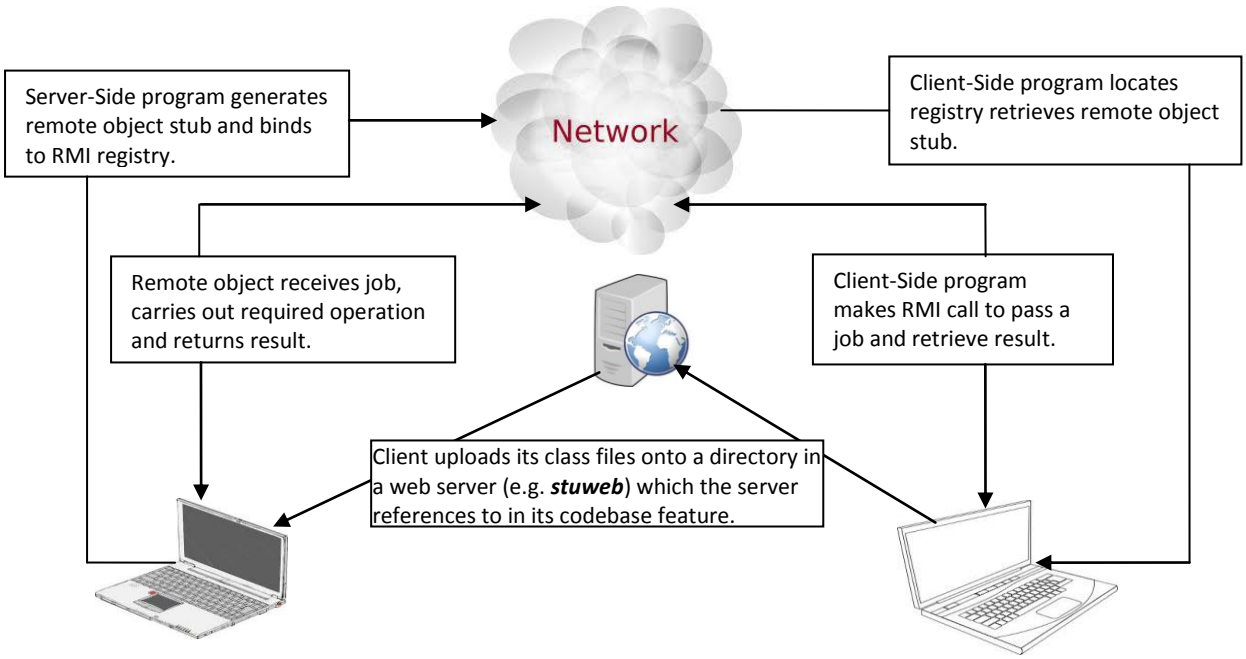
Without the step outlined above, the service will start up without any error; however, running the client application will throw an exception which will look like:

```
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:

```
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
```

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"));
   }
}
```

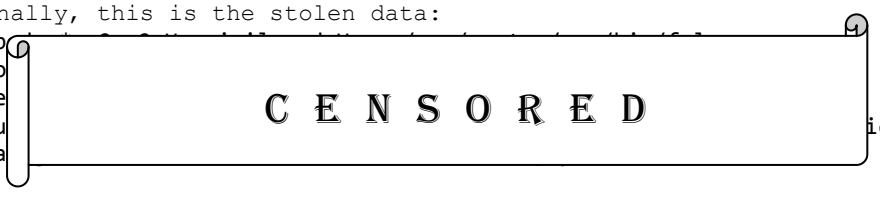
As the client program is run the server-side console as usually outputs:

```
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                                                              ico
_ta
```



5 Test application – non-public data serialization:

5.1 Overview:

This RMI application has a server application, a client program and a remote interface. The server application packages the following:

- a class, named `Employee`, which has some attributes of different public and non-public access levels and getter methods to retrieve the values of these attributes.
- a `Manager` class, which is a subclass of `Employee`, with a private access levelled field.
- a server-side program, referred to as `ServerSideProgram`, that creates remote objects of *Type* `Employee` and `Manager` initializing various fields and values.

The server-side program generates stubs with locally instantiated remote objects and binds them to the RMI registry. The client program will eventually look up in the RMI registry to obtain the stubs. It will then read the remote objects to find out whether the non-public field values of the remote objects are exposed.

5.2 Application components:

The application has the following components:

5.2.1 Remote interface:

This is the remote-interface for this RMI application. The interface is called and has the following definition:

```
public interface RemoteEmployeeInterface extends java.rmi.Remote {

  public String retrieveEmployeeName() throws RemoteException;
  public String retrieveEmployeeRole() throws RemoteException;
  public String retrievePersonalIdentificationNumber() throws RemoteException;
  public String retrieveEmployee() 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 only way a client may call a method on a remote object (which resides on the server-side) is by calling the method on the remote-interface object as the client doesn't have any knowledge of the server-side remote classes.

5.2.2 Server-side application:

Consisting of the following classes:

5.2.2.1 Employee:

This is an ordinary class that implements `RemoteEmployeeInterface` in order to be able to make remote object instances. The class has some usual fields that an ordinary Employee object would be expect of, such as `name`, `role` and `personalIdentificationNumber`. The access levels of these fields, however, are varied to cater for the purpose of this test. The class looks like this:

```
public class Employee implements RemoteEmployeeInterface {
  public String name;
  public String role = "Not assigned";
  protected long personalIdentificationNumber = 0L;
```

Finally, the class implements the methods defined in the `RemoteEmployeeInterface` interface which will eventually be called by the client program.

```
@Override
public String retrieveEmployeeName() throws RemoteException {…}

@Override
public String retrieveEmployeeRole() throws RemoteException {…}

@Override
public String retrievePersonalIdentificationNumber() throws RemoteException
{…}

@Override
public String retrieveEmployee() throws RemoteException {…}
```

A call to the *implemented* `retrieveEmployee` method returns the attributes belonging to an `Employee` object in *String* form.

5.2.2.2 Manager:

This class inherits the `Employee` class above and has a private access levelled field called `secretInfo`. To serve the purpose of a private field, the value of this field will only be exposed to other appropriate classes through its getter methods. The class looks like this:

```
public class Manager extends Employee {
  private String secretInfo = null;

  void setSecretInfo(String secretInfo) {…}
  public String getSecretInfo() {…}
```

The class generates its own `personalIdentificationNumber` via method `generatePersonalIdentificationNumber`, which accepts a `dob` parameter in *String* form and generates a `personalIdentificationNumber` based on some calculation.

```
public void generatePersonalIdentificationNumber(String dob) {…}
```

This class also provides its own implementation of the `retrieveEmployeeRole` method, which returns values of `role` and `secretInfo` specific for a `Manager` object back to the caller.

```
@Override
public String retrieveEmployeeRole() throws RemoteException {…}
```

5.2.2.3 ServerSideProgram:

This class initiates the process of the server application. The class first implements a Java security manager and starts the RMI registry, called `registry`.

```
System.setProperty("java.security.policy", "security/client.policy");
        if (System.getSecurityManager() == null) {
              System.setSecurityManager(new SecurityManager());
        }
        try {
              Registry registry = LocateRegistry.getRegistry();
```

It then creates instances of `Employee` and `Manager`, exports them to the java RMI runtime to create `stubs` and finally binds those `stubs` to the `registry`. The core code looks like this:

```
Employee transportingEmployee = new Employee("Michael");

RemoteEmployeeInterface employeeStub =
(RemoteEmployeeInterface)UnicastRemoteObject.exportObject(transporting
Employee, 0);
registry.rebind("employee1", employeeStub);

Manager aManager = new Manager("George Loukas");
aManager.setSecretInfo("George's secret info.");
Employee employeeWrapper = aManager; // wrapped Manager object
                                     //into a Employee wrapper

RemoteEmployeeInterface managerStub = (RemoteEmployeeInterface)
UnicastRemoteObject.exportObject(employeeWrapper, 0);
registry.rebind("employee2", managerStub);
```

5.2.3 Client-side program:

The client-side program makes use of a class, called `ClientSideProgram`, to carry out remote method invocation on `stubs`, obtained from the RMI registry, produced by the server-side program. The class starts with implementing Java security manager and locating the RMI registry on the network.

```
System.setProperty("java.security.policy", "security/client.policy");
  if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
  }

 try {
   Registry registry = LocateRegistry.getRegistry();
```

It then looks up the `stubs` (remote object reference) by the names they were bound by the server program and stores the `stubs` into remote-interface variables.

```
RemoteEmployeeInterface transportedEmployee1 = (RemoteEmployeeInterface)
                                               registry.lookup("employee1");

RemoteEmployeeInterface transportedEmployee2 = (RemoteEmployeeInterface)
                                                registry.lookup("employee2");
```

Finally, the program calls the remote methods on the obtained `stubs` to get the results.

```
System.out.println(transportedEmployee1.retrieveEmployee());
System.out.println(transportedEmployee2.retrieveEmployee());
System.out.println(transportedEmployee2.retrieveEmployeeRole());
```

5.3 Application in action:

The ServerSideProgram starts the service on machine A by creating instances of the remote objects Employee and Manager, generating stubs and binding the stubs with RMI registry. Once the service is up and running, a ClientSideProgram from machine B locates the registry and looks up the remote-object-reference in the registry. The remote methods are then invoked on the remote-object-references by the client program to fetch the details of the remote objects.

In this example, the ServerSideProgram creates an Employee object as follows:

```
Employee transportingEmployee = new Employee("Michael");
```

This creates an Employee object with fields {name="Michael"; role="Not assigned", personalIdentificationNumber=0L}. The field personalIdentificationNumber is a protected attribute; the value of this attribute can only be directly read or set (i.e. without any accessing method or property) by a child class of Employee or other classes in the same package. In other words, personalIdentificationNumber field is a restricted and non-public attribute and its value can only be disclosed under certain criteria.

The ServerSideProgram class also creates a Manager object as:

```
Manager aManager = new Manager("George Loukas");
```

Being a child object of Employee (i.e. Employee is a super class of Manager), the aManager object has direct access to attribute personalIdentificationNumber of Employee class (*protected* field) and sets the value of this attribute in a designated method named generatePersonalIdentificationNumber based on some calculations. Furthermore, the aManager object also sets a private attribute named secretInfo which is strictly visible to this object alone. In other words, these fields are non-public and designed not to be disclosed to the outside world.

```
aManager.generatePersonalIdentificationNumber("02/02/1975");
aManager.setSecretInfo("George's secret info.");
```

The object aManager is then stored in an Employee wrapper to facilitate stub generation to be transferred over the network.

```
Employee employeeWrapper = aManager;
```

Once created, these objects, namely transportingEmployee and employeeWrapper are exported to the RMI runtime. The resulting stubs, referred to as employeeStub and managerStub, are then bound to the RMI registry.

```
RemoteEmployeeInterface employeeStub = (RemoteEmployeeInterface)
                    UnicastRemoteObject.exportObject(transportingEmployee, 0);
RemoteEmployeeInterface managerStub = (RemoteEmployeeInterface)
```

```
                    UnicastRemoteObject.exportObject(employeeWrapper, 0);
registry.rebind("employee1", employeeStub);
registry.rebind("employee2", managerStub);
```

When the ServerSideProgram is run on machine A, the following output is printed on the console:

```
Sever side (Mahabuburs-MacBook-Pro.local/192.168.0.5):
=======================================
I am a server application serving some locally instantiated objects with noble
intention to make necessary information available to enquiring clients to satisfy the
objective of AVAILABILITY in CIA principle.
Hopefully I'm not breaching CONFIDENTIALITY by giving away non-public information.

Exported an employee object in RemoteEmployee wrapper...
Exported a manager object in RemoteEmployee wrapper...
```

The client program, named ClientSideProgram, on machine B eventually consumes the stubs. The details of the client program's operation have already been described in section 5.2.3.

When the ClientSideProgram is run, following information is printed on the client's console:

```
Client side (Toshiba/192.168.56.1):
=======================================
I am a client application enquiring some publicly disclosable data about employee.
```

The client program stores the first stub, named as "employee1" on the server-side (i.e. remote reference to the Employee object), into a RemoteEmployeeInterface variable transportedEmployee1:

```
RemoteEmployeeInterface transportedEmployee1 = (RemoteEmployeeInterface)
                                                registry.lookup("employee1");
```

, and when the remote method retrieveEmployee is called on transportedEmployee1 object

```
System.out.println(transportedEmployee1.retrieveEmployee());
```

, it prints the following:

```
Received first employee information pack, unpacking wrapper...
{
Name: Michael;
Role: Not assigned;
PID: 0
}
I got more that the information I need! At least the personalIdentificationNumber
isn't anything meaningful.
```

As the output shows, although a protected attribute, the personalIdentificationNumber of the Employee object is serialized on the server-side and transported over the network to the client side and deserialized. Thus a protected data of a class can be easily exposed inadvertently to unintended users if it is used improperly or implemented incorrectly.

The next stub, named as "employee2" on the server-side (i.e. remote reference to the `Manager` object) is stored in variable `transportedEmployee2`:

```
RemoteEmployeeInterface transportedEmployee2 = (RemoteEmployeeInterface)
                                               registry.lookup("employee2");
```

, and when the remote method `retrieveEmploye` is called on `transportedEmployee2` object

```
System.out.println(transportedEmployee2.retrieveEmployee());
```

, it prints the following:

```
Received second employee information pack, upacking wrapper...
{
Name: George Loukas;
Role: Management;
PID: 57912020
}
Why do I see personalIdentificationNumber? Isn't this supposed to be a protected
info?
```

The output shows that the value of the `personalIdentificationNumber` attribute is also serialized for a `Manager` object the same way as the parent `Employee` object. This demonstrates that although the `Manager` class itself doesn't implement the Java RMI remote interface, it also equally exhibits RMI data serialization characteristics (along with the consequence of any incorrect implementation) through inheritance from its parent class.

Finally, when the remote method `retrieveEmployeeRole` is called on `transportedEmployee2` object

```
System.out.println(transportedEmployee2.retrieveEmployeeRole());
```

, it prints the following:

```
{
Role: Management
Secret info: George's secret info.
}
Whoa!!! that's a lot of private data you've disclosed there!
```

This output demonstrates that because of improper use or incorrect implementation, not only the *protected* but also the *private* data can be equally vulnerable in an RMI application due to the nature of serialization technique. Even if the class itself doesn't implement the Java RMI remote interface, the behaviour is still inherited from its parent class.

6 Reflection:

Still to add.

7 Conclusion:

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

8 References and Keywords:

8.1 References:

a) Waldo, J. (1998), "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) Baclawski, Kenneth (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) Li, Ninghui; Mitchell, J.C.; Tong, D. (2004). "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) W. Long, Frederick (2005). Software Vulnerabilities in Java [Research Showcase]. [online]. Carnegie Mellon University, October 2005. Available from: http://repository.cmu.edu/sei/422/.

i) Security Explorations, “Security Vulnerabilities in Java SE”, Technical Report, Ver. 1.0.2, SE-2012-01 Project. URL: http://www.security-explorations.com/materials/se-2012-01-report.pdf

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;