SECURING RMI COMMUNICATION

Vincent Naessens
K.U.Leuven, Campus Kortrijk (KULAK)
vincent.naessens@kulak.ac.be

Bart Vanhaute
K.U.Leuven, Dept. of Computer Science, DistriNet
bart.vanhaute@cskuleuven.ac.be

Bart De Decker
K. U.Leuven, Dept. of Computer Science, DistriNet
bart.dedecker@cs.kuleuven.ac.be

Abstract: Application programmers often have to protect their applications themselves in order to achieve secure applications. Therefore, they have to possess a lot of knowledge about security related issues. The solution to this problem is to separate the security-related modules as much as possible from the real application and transparently invoke these security modules. By doing this, the application programmer can build his distributed application without considering the security requirements.

The case study presents how to achieve transparent security in the RMI (remote method invocation) system, an API provided by Java to implement applications in a distributed environment. The presented framework is also flexible enough to support different levels of security.

Keywords: open distributed system, security framework

1. INTRODUCTION

Enterprises are increasingly dependent on their information systems to support their business activities. Compromise of these systems either — in terms of loss or inaccuracy of information or competitors gaining access to it — can be extremely costly to the enterprise. Security breaches, are becoming more frequent and varied. These may often be due to accidental misuse of the system, such as users accidentally gaining unauthorized access to information. Commercial as well as government systems may also be subject to malicious attacks (for example, to gain access to sensitive information). Distributed systems are more vulnerable to security breaches than the more traditional systems, as there are more places where the system can be attacked. Therefore, security is needed in distributed systems. This case study presents how to achieve transparent security in the RMI system.

Security protects an information system from unauthorized attempts to access information or interfere with its operation. The key security features we are concerned with are:

- **identification** and **authentication** to verify parties who they claim to be.
- **authorization** and **access control** to decide whether some party can execute some action.
- **protection of communication** between parties. This requires trust to be established between the client and the server, which involves authentication of clients to servers and authentication of servers to clients. It also requires integrity and confidentiality protection of messages in transit,
- **audit trail** of actions.

Apart from these security requirements, **administration** of security information is also needed.

In client/server applications, objects located at one host are communicating with objects running on other hosts. The key security features can be provided at two levels: at the location[1] level and at the object level. Security features provided at the location level secure communication between two hosts. This kind of security is independent of the objects communicating between these hosts. Each object can also be individually protected if security is provided at the object level. It is clear that security provided at the object level is more fine-grained than security provided at the

[1]Locations will mostly correspond with hosts; more precisely, they correspond to Java Virtual Machine instantiations.

location level. We will discuss how each of these security features can be built into the system.

The main goal of this case study is to provide a flexible and transparent security framework for the RMI system. *Flexibility* means that it must be possible to incorporate different mechanisms and services, according to the degree of security that is required. *Transparency* mains that applications are not aware of the security aspects built into the system. Hence, each of the security features should be implemented into the RMI system itself. That way, application programmers do not have to recompile their applications to work with the secured framework.

A first section briefly describes the architecture of the RMI system. The second section introduces the security components and discusses where these services should be added in the RMI system. By including these components in the RMI system itself, they are transparent with respect to the application. The third section presents a security framework for RMI that is flexible enough to support different levels of security. The next two sections discuss the transparency and the flexibility of the framework. Next, we refer to some related work in this area. The paper ends with a general conclusion.

2. THE RMI SYSTEM

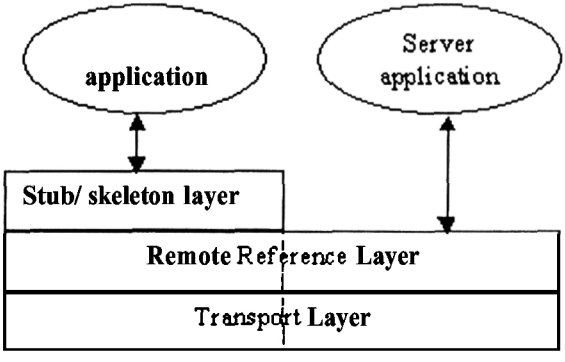

Figure 1. the RMI system

The RMI system [1] consists of three layers: the *stub/skeleton layer,* the *remote reference layer* and the *transport layer.* The application itself runs on top of this RMI system. When a client invokes an operation on a server object, a stub object passes the method to the reference layer that initiates the

call. The remote references are mapped to locations. A specific reference semantics is executed at that moment depending on the implementation of the reference layer. For instance, this layer can support point-to-point calls, calls to replicated objects, etc. The remote reference layer also sets up a connection to the server side by creating a new connection or reusing an existing connection. Depending on the implementation of the transport layer, TCP [2], UDP [3] or other types of connections are supported. When a server receives information on an incoming connection, the information will be forwarded to the reference layer that executes code according to a specific semantics. Finally, the remote object executes the method and sends the result back to the client side in the same way.

3. SECURITY COMPONENTS

To achieve a secure execution environment, some security components must be added into the distributed system. The security components discussed in this paper are the association component, the authentication component, the access control component, and the audit trail component. This section shows where these four security services are added into the RMI system. By including these services in the RMI system, they are transparent to the application.

Services can be added at two levels: the location level and the object level. Services provided at the location level are executed between hosts. Information provided at that level are the IP addresses of the communicating hosts, the principals executing at each of the two hosts, etc. Services provided at the object level are executed between objects. More information is available at that level. The method name and parameters of the remote invocation are known. Moreover, an object can be running on behalf of a certain principal. An access controller at the object level can make use of this information.

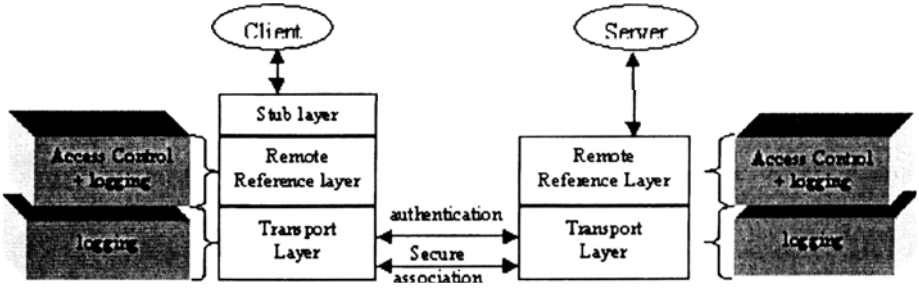

Figure 2. Security services in the RMI system.

3.1 Secure association service

Before messages are sent over the wire, a **secure association** must be established between two hosts: the client and the server. This service is provided at the location level. As a result of this phase, both parties possess a key that will be used to exchange further messages. Thus, setting up a secure association guarantees the confidentiality of the data that is exchanged between both parties. The secure RMI system performs this task after the connection is established and before the actual method invocation from the stub object to the server object takes place. This task can be fully executed at the transport layer, making use of the connection. The resulting key is also kept at the transport layer. As this service does not require any information about the objects, the same secure association can and will be reused over multiple calls between the two hosts.

3.2 Authentication service

Once a secure association is set up, an **authentication service** can be executed. Often, both parties will want to know the correct identity of the party they are dealing with, for instance as basis for authorization decisions. Alternatively, they may want to act anonymously. Authentication can be performed in a kind of handshake phase where trust is gained in the other party's identity and where security attributes are exchanged. This service can be fully performed at the transport layer, immediately after a secure association is set up. The resulting security attributes are also stored at the transport layer. Depending on the implementation, authentication is executed at the location level and/or at the object level. The presented framework only presents authentication at the location level.. This corresponds to the idea that ***users*** are typically controlling locations, and they are the principals we want to authenticate.

3.3 Access control service

The **access control service** (or authorisation service) gives a party the possibility to allow/disallow an action of the other party involved in the communication. In an object oriented environment, access decisions can be based on the method and the parameters that are sent to the server. This service is performed at the object level. Thus, access control must be performed at the reference layer, after the necessary information is unmarshalled and before the method will be invoked. This service can also make use of the security information that is stored at the transport layer.

3.4 Audit trail service

The **audit trail service** is responsible for logging information. Two types of logging are introduced. In the transport layer (i.e. at location level), information about the authentication procedure is logged. At the reference level (i.e. at object level), information about the authorisation and the method invocation is logged.

4. THE SECURITY FRAMEWORK

We developped a security framework for RMI that is flexible enough to support different security levels and mechanisms. By consulting a property file, the security components are loaded into the RMI system at runtime. By changing the values of this property file, other components are loaded into the system. On the one hand, objects are loaded that are responsible for holding security information. They are called **security context objects** (or security contexts). On the other hand, objects are loaded that are responsible for executing a specific security service. They are called **security service objects** (or security services). Security services can modify the information stored in the security contexts and query them to make decisions.

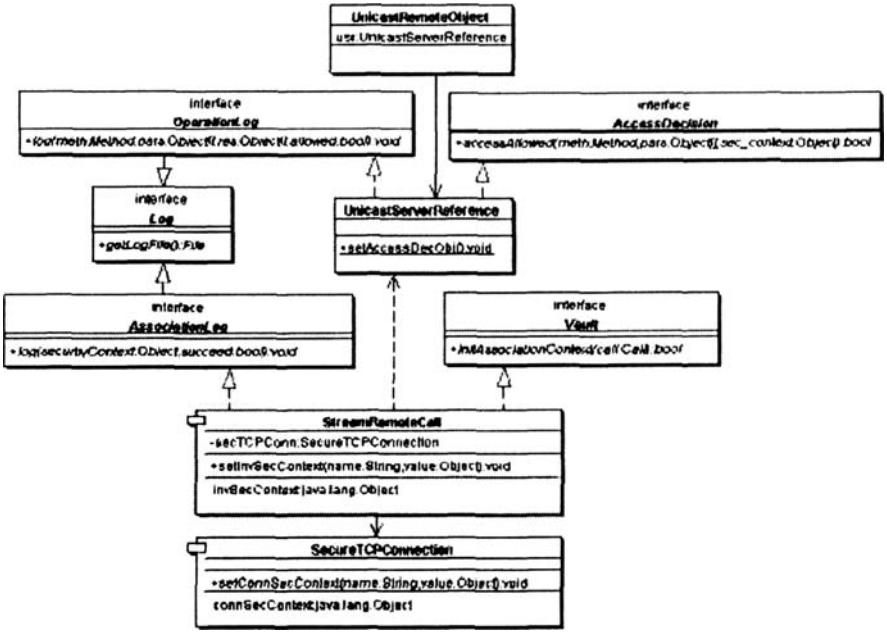

Figure 3. The security framework

4.1 Security context objects

To obtain a secure execution environment, two types of security contexts are introduced in the RMI system: a *connection security context* and *an invocation security context.* They are responsible for storing security-related data.

A **connection security context** contains security information specific for a particular connection. This context contains information exchanged during the secure association phase and the authentication phase at location level. More specifically, a connection security context can hold a session key, the time when the connection is created, the user or client that makes use of the connection, etc. Thus, every time a new connection is created, a corresponding new connection security context is initiated at the same level in the RMI system i.e. at the transport level. A connection security context disappears when the corresponding connection is closed.

An **invocation security context** holds information that is specific for a particular invocation such as the time the invocation is executed, the operation that must be executed and the parameters that belong to the operation. Thus, a new invocation security context is created each time a new call is initiated and is removed when the method call is finished. This is analogous to the first type of security context. When authentication is executed at object level, additional information is added into this context.

Remark that a connection security context can be considered a part of an invocation security context. Every invocation security context holds a pointer to a connection security context. However, the lifetime of a connection security context can be longer than the lifetime of an invocation security context. This is because the same connection can be reused during subsequent method calls.

4.2 Security service objects

Security service objects are responsible for executing some kind of security service. When a client invokes a method on a server object, a secure association is established and a particular authentication protocol is performed between the client and the server. To achieve these two tasks, a **vault object** [4] is introduced at the transport level. A vault object can perform these two tasks itself or delegate the work to an association object and an authentication object.

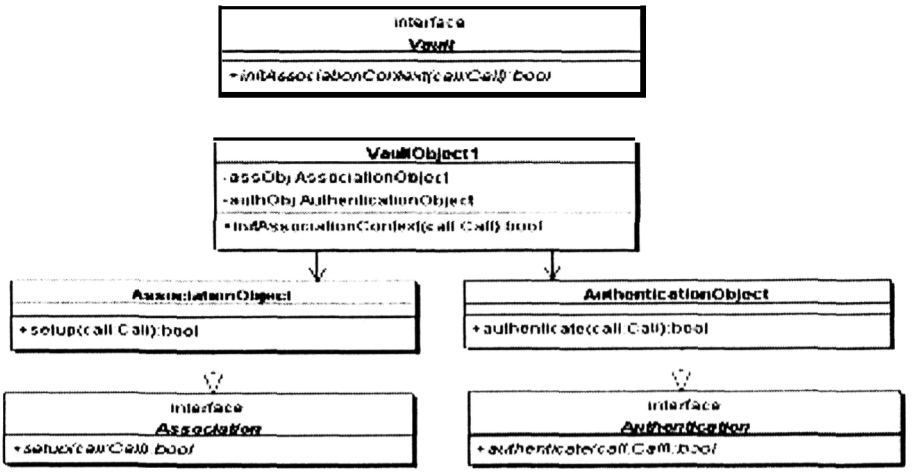

Figure 4. TheVault Object

When a call is initiated at the client side and a new unsafe connection is created, the **association object** can decide to exchange a session key with the association object on the server side. Several encryption libraries provide implementations of key agreement algorithms [5]. The resulting session key is stored at both sides in the connection security context. As a result of this step, further information can be sent in encrypted form to the connection object. In other words, encryption is done on top of a connection and therefore, it does not affect the implementation of a connection type. Moreover, if the association object sees that the connection itself is implemented to support secure communication (for instance by using SSL secure sockets), it can decide not to execute this first step. When a connection already exists, the association object can decide to update the connection security context if necessary. For instance when the time a particular key is valid, is exceeded, the vault object can ask for a new key agreement session to take place.

After this, the vault object calls an **authentication object** if authentication is not already done. Depending on its implementation, the authentication object explicitly asks the user for authentication information or makes use of credentials that are created when the user logs in on the system. These credentials are generated automatically when the user logs in on the system. It can happen that authentication is performed in several successive steps. For instance, the server side can ask for additional credentials or can conclude that the authentication data are not valid any more. In these two cases, the authentication continues. Authentication

information can be sent along a secure data stream making use of the session key obtained in the previous step. The authentication information is stored in the connection security context and can later be used to make access decisions.

Access control in an object-oriented environment mostly depends on the method that must be executed and the parameters of the method call. At that point in the execution, the information must be in an unmarshalled form. Marshalling and unmarshalling happens in the remote reference layer. This information is passed to the invocation security context object. After this information is set, an **access control object** can make a decision using the information kept by the security context. At the client side, access control can be checked just before marshalling information; at the server side access control executed after unmarshalling the operation and parameters and just before the information is dispatched to the application level.

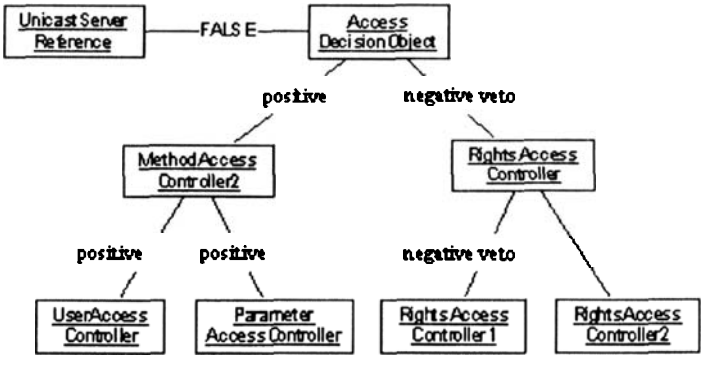

Figure 5 .Access controller

To provide a flexible access control mechanism, the access control object can be implemented using the composite design pattern [6]. A tree of access controllers makes an access decision. At the leaf level, the access controllers give a negative veto or advice, or a positive veto or advice to the intermediate nodes in the tree. This information is propagated to the top level of the tree that makes a final decision. Each access controller makes a particular decision. For instance, there can be user access controllers, rights access controllers, **...** These access controllers can be implemented totally independent of the actual application. To give the application the possibility to attach his own access controller to the tree, it can give a series of access controllers to the constructor of an application object. The constructor then appends the controllers to the tree in a predefined way.

Two types of log objects are introduced in the system: association log objects and operation log objects. **Association loggers** are introduced at the transport level and log information concerning the association. For instance, an association logger can save which client is trying to make a connection, if the authentication is successful, etc **... Operation loggers** are introduced at the reference layer and log information about the operations that have to be executed or that have already been executed. For instance, an operation logger can store the method a client tries to execute, the return value of the access control decision object, the result of the method call, etc. In contrast to vault objects and access control objects, we want to provide the possibility to pick up several log objects at each level.

5. TRANSPARENCY

Because the presented security features are all built into the RMI system, it can be reused for every **application.** Access control and operation logging happens at the reference layer; setting up a secure association and logging associations happens at the transport layer. This also implies that stub objects remain the same. Therefore, the rmic compiler that generates stubs, does not have to be changed. This implies full transparency from the point of view of the application programmer.

Providing full transparency to the **end user** of the application is difficult to achieve. A secure distributed system wants the user to be authenticated at some point in the execution. Depending on the implementation of the authentication object, the user has to do it explicitly during the application runtime or the authentication object can make use of the credentials created when the user logs on the system.

From the point of view of the **administrator** of the system, one can say that he has to make a decision about which security components have to be loaded into the RMI system. He has to make a property file. The RMI system consults this property file at runtime in order to know which instances of the security components to create.

The presented framework can also be considered to be relatively transparent to the **RMI implementation** because security components are added to the system by loading security related objects and not by adapting the implementation of existing objects in the system. For instance, a typical connection implementation (UDP or TCP) does not have to be adapted because encryption is provided on top of it.

6. FLEXIBILITY

Four types of objects are introduced in the framework: security context objects, access control objects, log objects and vault objects. In turn, a vault object can call an association object and an authentication object. An appropriate interface for each of these object types is available so that the RMI system can invoke a method of an object via this interface. The property file indicates which objects to load at runtime in the system. Separating the security components from the RMI system this way provides us a flexible way of working. Although a secure RMI package can provide us with implementations of each of these objects, new implementations can be introduced as long as they implement methods of the interface in an appropriate way.

Flexibility is also needed within the proposed security components. For instance, by implementing an access decision object as a tree of access controllers, new access controllers can be added dynamically. Vault objects present a similar degree of flexibility in that way they can decide to contact an association object and an authentication object, contact one of those two types of objects or contact no other object at all according to the level of security that is preferred in the system.

7. RELATED WORK

The **Java Secure Socket Extension** (JSSE) [7] is a Java optional package that provides Secure Socket Layer (SSL) and Transport Layer Security (TLS) support for the Java 2 Platform. Using JSSE, developers can provide for the secure passage of data between a client and a server. Secure sockets can be added into the RMI system at transport level to set up a secure association. This way, they are transparent in front of application programmers. In the presented framework, the Vault object is responsible for setting up a secure association between two hosts. An implementation of that Vault object can use JSSE.

The **Java Authentication and Authorization Service** (JAAS) [7] is a framework that supplements the Java 2 platform with principal-based authentication and access control capabilities. It includes a Java implementation of the standard Pluggable Authentication Module (PAM) architecture, and provides support for user-based, group-based, or role-based access controls. These modules can also be added transparently into the presented framework. The Java Authentication Service provides

authentication at object level. The framework we presented provides authentication at location level. However, we can extend the RMI security framework with authentication at object level as suggested in paragraph 3. The authorisation modules of JAAS can also be inserted into the framework in the Access Decision Object. But the security framework is flexible enough to support other types of access control. For instance, access control can also be based on the parameters and the operation that is invoked. Because Java has not specified standards for other types of authorisation, we have to make an own implementation of each of these services if that is required.

The **Common Object Services** specification (CORBASec) [4] describes security related tasks and requirements needed for CORBA. The specification is quite long and attempts to address an extremely wide range of security issues. The topic of distributed objects is complicated enough when considered on its own and it certainly does not get any simpler with the addition of security. Due to this, there are many issues that are underspecified and open to interpretation at this time, which gives scope for R&D in this area. To further extend the RMI security architecture with more advanced security services like delegation, a lot of inspiration can be found in this specification. Depending of the implementation of an ORB, different services are provided. This is similar with the flexibility of the presented RMI security framework.

The Java Community [8] is working on the definition of a high-level API for network security in JavaTM 2 Standard Edition RMI, covering basic security mechanisms: authentication (including delegation), confidentiality, and integrity. The main problem is that the proposals are not transparent enough towards applications. Our framework tries to achieve more transparency towards application programmers because all of the security features are built into the RMI system itself. However, the framework also enables application programmers to load their own security modules into the RMI system.

8. CONCLUSION

The presented framework gives the possibility to add different security services to the RMI system: setting up a secure association, authentication, authorisation and logging. These services are added to the RMI system in a transparent and flexible way. The implementation of the suggested objects in the framework depends on the level of security and the degree of complexity

that is needed. A simple implementation can already provide a good level of security. For a more advanced implementation of each of these objects, a lot of principles suggested by security specifications of other distributed systems such as CORBA [4], can be used.

REFERENCES

1. Java™ Remote Method Invocation Specification - JDK1.2 Beta 1, October 1997
2. RMI implementation provided by SUN using TCP connections. http://iava.sun.com/products/jdk/rmi/
3. RMI implementation provided by Ninja using UDP connections. http://ninja.cs.berkelev.edu/ninja/
4. Corba Security Service Specification - november 1996.
5. IAIK: http://www.iaik.tu-graz.ac.at/
   ABA: http://aba.net.au/
6. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 'Design Patterns, elements of reusable objectoriented software', Addison-Wesley 1995
7. http://www.java.sun.com/security/
8. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_076_rmisecuritv.html