Procedure-level Authorization for Java Remote Method Invocation Using SSL Credentials

Alexander Shoulson
Center for Human Modeling and Simulation
Computer and Information Science
University of Pennsylvania

Kevin Pearson
Information Directorate
Air Force Research Laboratory

ABSTRACT

Due to the encapsulation of the RMI transport layer, methods invoked over RMI lose access to the sockets through which their particular instances were called. This prevents subroutines from first verifying the credentials of their caller to authorize each particular invocation. We present a technique within Java RMI enabling socket information to be re-embedded into the RMI transport and made accessible to methods on their invocation. This implementation is also thread-independent, which allows for safer parallel processing without the limitations of any thread assumptions.

KEYWORDS

Java, RMI, RPC, mobile code, SSL, Authorization

TABLE OF CONTENTS

LIST OF FIGURES

1. EXECUTIVE SUMMARY

As currently implemented, Java Remote Method Invocation (RMI) only provides the ability to perform authorization of object calls based on their exposure and the ability to successfully authenticate to the RMI object repository, called the Registry. This does not allow implementers to perform fine-grained access control, down to the procedure/method/function level, nor does it allow procedures to tailor responses or result sets based on the invoker's principal. The current specifications for Java RMI only allow the server the client's hostname or address, which can be an unreliable means of uniquely identifying the client due to multi-user systems or Network Address Translation (NAT) which allows multiple systems to effectively have the same public-facing Internet Protocol (IP) address.

Through our research, we have created a technique of providing unique identifier information that can be used to determine the security principal of a client from within the context of an RMI call's concrete implementation. This allows for fine-grained access control via authorization at the procedure level, allowing information owners and information handlers to better protect and control information and information dissemination.

2. INTRODUCTION

Remote Procedure Call (RPC) is a paradigm designed to allow distributed systems to communicate with one another using a standardized protocol abstracted in the form of invoking subroutines. This alleviates the burden of packaging and dispatching individual messages on the socket-layer, which could require a specialized protocol for each distributed system. Developing such a protocol would be costly, and the encoding/decoding process could be prone to errors. As such, RPC standards such as the Common Object Request Broker Architecture (CORBA) and the Distributed Common Object Model (DCOM) arose to provide standards that could be applied to a wide range of applications. Java Remote Method Invocation (RMI) represents an RPC system tailored to the Java architecture, capable of leveraging tools such as Java objects and interfaces. Java RMI is particularly flexible in its polymorphism which, when coupled with the type-safety, object-orientation, and platform independence already present in Java itself make RMI a useful tool for distributed processes (Maasen, et al., 2001), (Waldo, 1998).

For sensitive systems, cryptographic protocols such as the Secure Sockets Layer (SSL) allow for confidential communication between applications. This protection is twofold: The SSL protocol first uses public-key cryptography to authenticate the identity of the recipient of the data exchange (and, optionally, the sender), and then encrypts the data itself so that it is difficult to decipher if intercepted. After authentication, each system can trust the identity of the other peer and the integrity of their data (Wagner & Schneier, 1996). For secure RMI, Sun Microsystems provides the support for establishing the necessary protocol communication channels over secure sockets using the Java Secure Socket Extension (JSSE). This allows both client-to-server and server-to-client authentication, and protects the data passed in method invocations. This is done with the javax.rmi.ssl.SslRMIClientSocketFactory and javax.rmi.ssl.SslRMIServerSocketFactory classes. These two classes produce sockets that insert an SSL between the TCP communication layer and the RMI transport layer, as seen in Figure 1 on Page 2 (Adapted from (Konstantinou, 2001)). In this figure, dotted lines indicate overall roles in communication, while solid lines trace a single remote method invocation. It is important to note that the server application has no native access to the client's certificate under standard RMI protocol. (Konstantinou, 2001).

While the standard Java RMI implementation can authenticate a user on connection using public-key credentials, information pertaining to that user's certificate and principal (the most important identifier) is lost to the methods invoked. The RMI transport architecture does not provide innate access to the invoking user's socket or credentials. As such, once a user has access to an RMI remote object, that user is able to call any of the methods contained in the object. This makes individually graduated permissions impossible on a single RMI service. This prevents scalable fine-grained access control for invoking RMI methods on large systems that handle sensitive data.

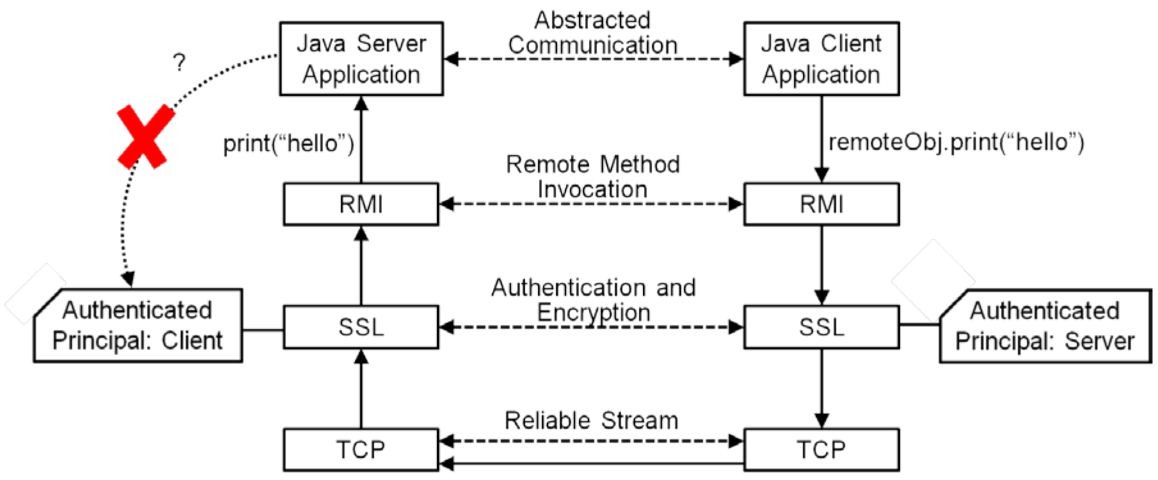

Figure 1: An overview of the layers involved in Java RMI.

At least one prior solution to this problem exists. The technique in question involves toring the socket information of the last connection read by the server to a thread-local variable. When any method is invoked, it polls the thread-local variable to retrieve the handle for the communicating socket, and then accesses the caller's principal. As is noted, this technique is "based on the assumption that the arguments are [decoded] in the same thread as the one used to invoke the method call" (Konstantinou, 2001). However, as is stated in the RMI specifications, "The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads." The specification goes on to explain that multiple invocations may execute in the same thread on the server (Sun Microsystems, 2006). If such an event occurs, this technique as presented could produce a possible race condition within the context of a thread, which could result in the wrong thread-local variable being accessed for a corresponding connection. While this solution could still be useful in some contexts, especially with extra book-keeping in terms of synchronization and mutual exclusion on these connection threads, our ultimate goal was to avoid these potential pitfalls entirely by working more strictly within the Java RMI specifications.

The remainder of this paper presents an alternate solution to thread-local variables. By exploiting a mechanic of Java RMI, we are capable of embedding socket information into the RMI transport itself. This achieves all of the desired functionality without relying on any assumptions about threading.

In practice, the Java RMI Application Programming Interface (API) provides a method for obtaining the IP address of a method's caller through the RMI transport layer. This method, called `getClientHost()` provides no information as to the port on which that particular client is connected, either remotely or locally. As such, given the prevalence of multi-user systems and network address translation, the method does not provide a unique identifier for a client. It is very possible that two systems may be connecting to the server from the same local area network, giving them the same host IP. We require a way to differentiate the two for secure individual subroutine permissions. Also, to perform authorization based on an individual, not a system, there must be means to acquire and query the security principal used for secure communication.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

For our attempts to develop an RMI authorization mechanism, we have certain constraints under which we operated. Primarily, the implementation shall require no extraneous code execution on the client side, all additional operations and processing will occur on the server. Secondly, authorization mechanisms shall conform to the Java RMI specifications. Implementation should be as lightweight as possible, not requiring significant changes to existing RMI services in order to provide authorization. Furthermore, the implementation of the authorization mechanism should be transparent and not interfere with RMI objects and services that do not wish to use such an authorization scheme.

We explored traceback of the standard Java RMI implementation to the point of transition where proprietary code is used in an attempt to discover a point in the execution stack where access is provided to both the client's security principal and the remote object's method. If such a junction point were to exist, injection of an authorization mechanism at that point could be investigated. For legal reasons and a lack of source to the propriety implementation of key interfaces, if no such junction point existed then the injection approach would be deemed infeasible.

We explored all aspects of the remote object access interfaces to determine any possible means to access the security principal of the client from within the execution of a remote object instance. We also attempted to gain access to the socket over which RMI calls were being transported, as a means of correlating the current execution context to a security principal that could be derived from the SSL socket context. If unable to discover means to access the socket directly from the execution context of a remote method call, then a method of correlating unique identifier that is accessible from both a socket factory implementation and the execution context of a remote method call would be explored.

4. RESULTS AND DISCUSSION

Our attempts to trace the Java RMI implementation to a junction point where access to the security principal and remote object's instance were unsuccessful. While there is likely some such junction point within the proprietary implementation where a fine-grained authorization mechanism could be injected, we were unable to pursue such an approach. Furthermore, with the junction point lying in proprietary implementation code, any mechanism injected there would likely create an unwanted dependency on that particular implementation and its version, which is out of the control of the server application developers. Thus, we put forth all effort in pursuit of the second approach, where we attempt to access the security principal from within the execution context of a remote method call in order to perform authorization, described below.

The solution to unique identification of connected clients is in exploiting the `getClientHost()` function within the RMI transport. Depending on implementation, the transport can behave in one of two ways. Either the transport fetches the host IP of each incoming socket once on creation and stores that information for when `getClientHost()` is later called, or a call to `getClientHost()` queries the socket directly for its host IP with no intermediate information storage. In either case, we overload our server's socket class to report metadata on that socket. On creation, each of our server sockets generates a 128-bit (to fit the IPv6 specification) securely random number that will serve as that socket's identifier. We then overload any function that requests the socket's IP address to perform a stack trace. If the RMI transport's `getClientHost()` is on the current call stack (or an equivalent function invoked by the RMI transport), our socket reports the overloaded IP address as its unique identifier. Otherwise, the socket will behave normally and report its actual host address.

In addition to overriding the canonical SSL sockets included in Java RMI, this solution utilizes what we call an AuthorizationManager to perform the necessary authorization for subroutine calls. Upon the creation of a socket, when that socket's unique 128-bit ID is generated, the AuthorizationManager stores a reference to that socket, keyed to the unique ID. That socket reference can be used for retrieving the user's principal for certificate authorization. All of these aspects combine so that when a method is invoked over RMI, the method retrieves the client socket's "surrogate IP" returned by `getClientHost()` (i.e. the 128-bit generated ID) . That ID is passed to the AuthorizationManager, which resolves the socket reference from its internal mapping, and compares that socket's principal to a list of accepted principles from a given authorization policy. The invoked method receives the authorization information and can either perform the prescribed action or throw a remote error regarding a lack of authority.

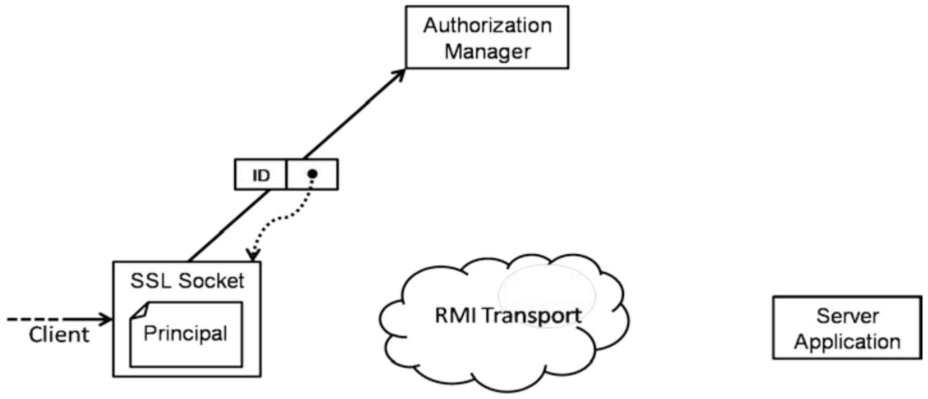

Figure 2: The creation and registration of an SSL socket.

On creation, in Figure 2 above, the SSL socket registers with the Authorization Manager, providing a reference to itself, keyed to its unique identifier. This reference and corresponding key will be used to access the client's socket throughout the invocation of any methods in question. Some maintenance can be done within the Authorization Manager to handle disconnecting clients and other related events dealing with the management of unique identifiers. This allows the server application to query the principal of the client connection in order to perform authorization, as shown in Figure 3.

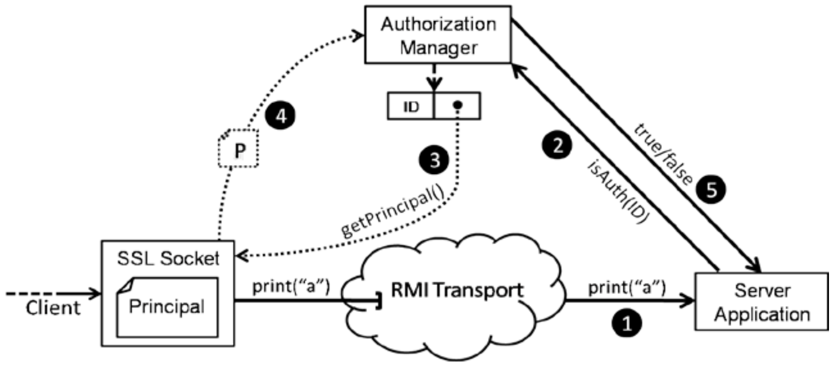

Figure 3: Steps involved in authorizing a method invocation

It is important to note that this implementation is entirely server-side, and requires no special code within the client application. Assuming client supports the Java RMI specifications and codebase, particularly the `javax.rmi.ssl.SslRMIClientSocketFactory` class and its related support classes, it can connect to an RMI server that uses this authorization scheme. In effect, this method of procedure-level authorization is entirely transparent to the client unless they attempt to invoke a method to which they do not have access.

5. CONCLUSIONS

In this paper we demonstrated a lightweight addition to RMI allowing more fine-grained authorization for user access. In particular, by leveraging RMI's ability to use custom-defined sockets and socket factories for its communication, we allow each process individually to be controlled according to a particular policy. Process-level control is important for larger systems because it removes the need to provide multiple servers offering different services at each level of access, along with allowing overlapping access levels. This scheme also has the added advantage of requiring no code on the client, working entirely server-side.

This authorization scheme works primarily by virtue of allowing a method to retrieve a unique identifier for the socket over which it was invoked, a capability that RMI does not provide. This method is safer because it works within the RMI API itself, and adheres to all of the assumptions and conditions of the official Java RMI specifications. While the exact implementation may need to be customized to some versions of RMI, the architecture of the scheme itself relies only on RMI being implemented according to its API and specifications. We consider this to be a reasonable assumption for any practical RMI application.

Naturally, if Sun Microsystems were to allow methods to uniquely identify their caller's socket and access the related principal, this workaround would no longer be needed. Still, since a thread-based RMI authorization scheme (another workaround) was published in 2001, it may be the case that no update to RMI will allow for unique client identification in the foreseeable future. In conclusion, we have shown that RMI can allow procedure-level access without the need to establish multiple registries or servers overall.

6. RECOMMENDATIONS AND FUTURE WORK

The technique described in this paper to obtain access to the security principal provides the ability to perform finer-grained access control to remote objects and their subcomponents. We understand that performing such authorization in the code of the remote object is cumbersome to developers, nor does it support adaptability as access rights change within an organization. To solve this problem, we intend to develop an authorization policy syntax and reference monitor implementation for controlling access to remote objects. We intend to support simple flat-file policy structures as well as integration with enterprise capabilities such as the Lightweight Directory Access Protocol (LDAP) and Microsoft Active Directory for providing group or role-based access control to remote objects.

7. REFERENCES

Konstantinou, A. V. (2001, Dec). *Using RMI Over SSL Authentication for Application-Level Access Control.* Retrieved June 21, 2010, from http://cs.columbia.edu/~akonstan/rmi-ssl

Maasen, J., Van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., et al. (2001). Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS) , 23* (6), 747-775.

Sun Microsystems. (2006). *Java Remote Method Invocation.* Retrieved Jun 21, 2010, from 3 - RMI System Overview - Thread Usage in Remote Method Invocations: http://java.sun.com/javase/6/docs/platform/rmi/spec/rmi-arch3.html

Wagner, D., & Schneier, B. (1996). Analysis of the SSL 3.0 Protocol. *WOEC '96: Proceedings of the 2nd Conference on Proceedings of the Second USENIC Workshop on Electronic Commerce* (pp. 4-4). Oakland, California: USENIX Association.

Waldo, J. (1998). Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency , 6* (3), 5-7.

8. LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API	Application Programming Interface
COBRA	Common Object Request Broker Architecture
DCOM	Distributed Common Object Model
IPv6	Internet Protocol Version 6
JDK	Java Development Kit
JSSE	Java Secure Socket Extension
LDAP	Lightweight Directory Access Protocol
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SSL	Secure Socket Layer
TCP	Transmission Control Protocol