Extending Jini with Decentralized Trust Management

Pasi Eronen, Johannes Lehtinen, Jukka Zitting, and Pekka Nikander
Helsinki University of Technology
Konemiehentie 2
FIN-02015 HUT, Finland
{pasi.eronen, johannes.lehtinen, jukka.zitting, pekka.nikander}@hut.fi

***Abstract*-Decentralized Trust Management, originally introduced by the PolicyMaker and SDSI prototypes, and currently promoted at least by the KeyNote2, SPKI, and TeSSA development efforts, provides a means of distributed authorization that seems to be especially suitable for distributed object systems and agent based systems. In this paper we introduce the SIESTA project, which studies how to integrate the ideas of decentralized trust management to the Jini environment. The focus of the functionality is on the use of SPKI certificates to secure Jini services. Controlling untrusted code is also an important issue because to use a Jini service one has to rely on proxy code loaded from the network. The resulting system allows decentralized authorization and trust management of Jini-based services and applications.**

I. Introduction

Traditionally, security has been based on identity authentication and locally stored access control lists even in distributed systems. However, this approach has a number of drawbacks, including, for example, the problem of protecting remote access control list management operations. An alternative to the traditional approach is the use of authorization certificates, as suggested, for example, in [1], [2], [3]. The HUT TeSSA project [4] has further refined these ideas in the context of object systems, and shown how these techniques can be applied to Java security in a distributed setting [5], [6], [7].

The basic Java facilities for secure, downloadable code are central in Sun's Jini technology [8]. Jini allows devices in a network to find each other and form communities (that is, groups of devices sharing some servsices). Jini services are accessed via proxies, which are downloaded from the network on demand.

In this paper, we introduce the SIESTA project, which is building a prototype of a secure, group aware distributed Personal Information Management (PIM) application on the top of Jini. The central component of this project, the Jini security library, is a generic authorization framework for Jini-based applications, as is explained in detail in this paper. The authorization framework provides facilities that let users to authorize pieces of code loaded from the network to perform certain operations, that also allow users themselves to be authorized to use services offered by other devices, and that make it possible for these authorizations to be dynamically delegated.

Specifically, the developed Jini authorization framework addresses the following issues.

- The framework allows both security-aware and security-unaware applications to be authorized to have access to Jini services. The access credentials are represented in the form of authorization certificates.
- It provides an interface for Jini service proxies to request delegated authorization from their hosting application. That is, a Jini service proxy downloaded by an application may ask for a certain permission that the application may have or have not, and if the application has the permission, the proxy may get it too.
- Security-aware applications may restrict when and which authorizations are delegated to which proxies.
- The framework takes care of obtaining all the necessary certificates when verifying access to a Jini service. The service need not implement this logic itself.

The rest of this paper is organized as follows. In Sect. II we describe the traditional ACL based access control, show the benefits of authorization certificate based access control in a distributed setting, and discuss authorization in the context of Jini. Sect. III describes our changes to the Jini operational model. Next, in Sect. IV, we describe the architecture of our solution. Finally, Sect. V contains our initial conclusions.

II. Access control

The traditional way to implement access control is to use access control lists (ACLs). A resource (for example, a confidential web page) has an ACL associated with it. The ACL contains the names of users allowed to access the resource. When the user tries to access the resource, user identification is performed, for example, with passwords or public-key cryptography. The service then checks the user name with the ACL, and either grants or denies access.

Authorization certificates provide a totally different approach to access control. Instead of storing a list of authorized users with each resource, we store the list of authorized resources with each user. The user has a number of "tickets" authorizing some action, and when she accesses the service, she hands over the relevant ticket. The service then checks if the ticket is valid.

The tickets used are authorization certificates; in our case, Simple Public Key Infrastructure (SPKI) certificates, as specified in RFC 2693 [2]. The main benefit of using authorization certificates instead of ACL is delegation. That is, an user who is authorized to use some resource can authorize somebody else to use the resource, if delegation is permitted by the service owner.

A. SPKI certificates

An SPKI certificate has five security related attributes: *issuer, subject, delegation, tag,* and *validity*, often represented as a 5-tuple (*I, S, D, T, V*). *Issuer* is the public key of the principal who issued the certificate, and the whole certificate is signed by the corresponding secret key to establish authenticity. *Subject* is the public key of the recipient of the permissions. *Delegation* is a boolean flag telling whether the subject may authorize other users or not. *Tag* is a service-specific field which describes the permissions included in the certificate, and *validity* describes the conditions under which the certificate is valid (for example, the time of expiration).

When using authorization certificates, the permissions are typically granted by issuing the administrator of a service a certificate which gives a permission to delegate any service-related permissions. The administrator may then delegate subsets of the permissions by issuing new authorization certificates. The new certificates may or may not include the delegation permission. Each certificate is signed by the issuer so that the the authenticity of the certificate can be confirmed. The user is authorized by a certificate chain beginning from the first issuer, i.e., the service, and ending to the last grantee or subject, the user. The service checks whether the chain is valid and whether the certificate chain as a whole implies the permission to perform the requested operation. Fig. 1 illustrates the concept of certificate chain.

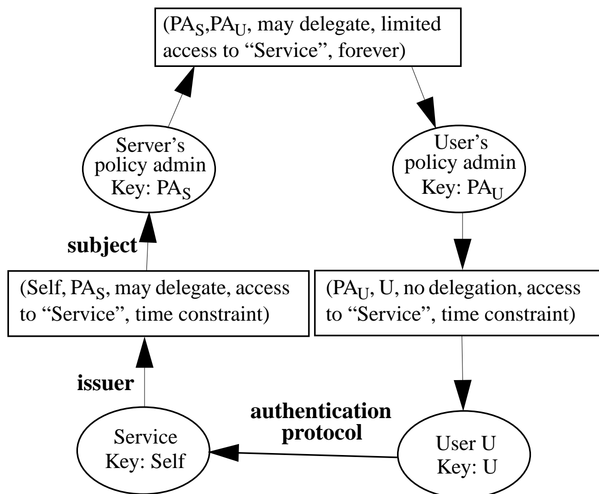

Fig. 1. Basic authorization certificate loop, represented as 5-tuples

B. Using authorization in Jini

One of our goals was to study how to add authorization and delegation to Jini. Therefore, we implemented a security framework providing authorization to Jini based services and applications. Fig. 2, below, illustrates the most important component related concepts and their associations.

- Applications and proxies are Java programs.
- Java programs and services are programs.
- Persons and programs are principals.
- A principal authorizes another principal by a certificate to perform an operation.
- A person runs a Java program using a JVM.
- An application uses a service through a proxy to perform an operation.

The users can receive authorization to use some Jini service (for example, a printer) from the administrator of the service. The authorization is issued as a SPKI certificate written to the user's key. The certificates and the user's private keys are stored in her computer.

The users can delegate a subset of their authorizations to trusted local applications. The authorizations delegated to the application depends on how the user trusts an application. For example, the user might trust a freeware drawing application to print correctly and she would delegate the corresponding permission to the application. However, the user does not give the drawing application the permission to access personal calendar files, because the application does not really need it, and it just might contain code that misbehaves.

The Jini security library provides a way for applications to use these authorizations with a service in Jini environment. One of the problems to be solved is how to prove these authorizations through the Jini proxy which is loaded from the network and can not be fully trusted by the user. The user's secret key is required to prove the user's authorizations but it must not be given to the proxy.

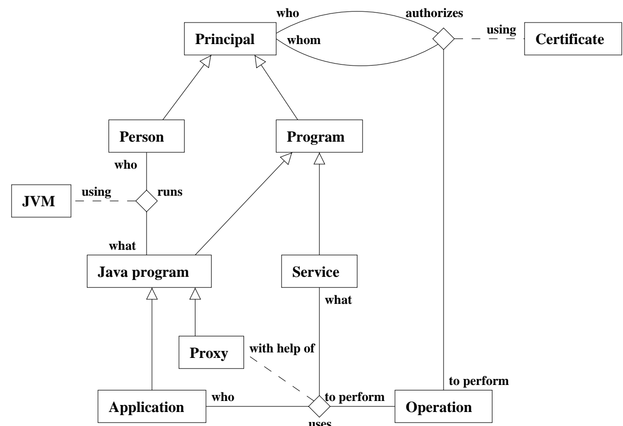

Fig. 2. The conceptual model of Jini authorization

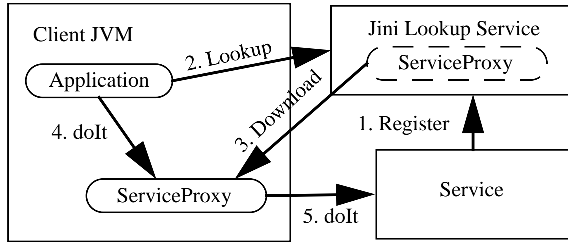

Fig. 3. Default behaviour in Jini Service lookup

III. Changes to the Jini operation model

The default behaviour of a Jini application and a Jini service is shown in Fig. 3, above, where a method `doIt` of the service is invoked by the application. Before the invocation is possible, a number of activities must take place.

1. The service registers its proxy with the lookup service.
2. The application, wishing to use the service, queries the lookup service for a service providing doIt.
3. The proxy is downloaded to the client.
4. The application calls some method on the proxy object, requesting it to do whatever the service does.
5. The proxy sends the request to the service.

With our authorization certificates, the picture is a little bit different (Fig. 4, below). In our example, the application is not security-aware.

As prerequisites, the service has authorized the administrator to use it, and the administrator has authorized the user. The user's key and the certificate proving the permission are stored in the JVM. When accessing the service, the four first steps are identical with the default Jini behaviour. Thereafter come the differences:

5. When the proxy receives a request, it knows that some authorization is needed. It asks the Jini Security library for this.
6. The Jini Security library first checks if the application is allowed to access this kind of service, and if it is, writes a short-lived certificate from the user's key to the proxy's temporary key.

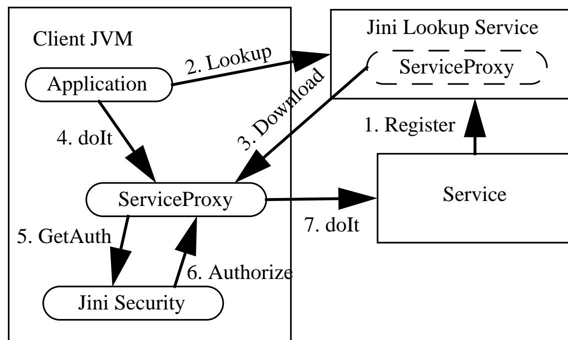

Fig. 4. Extending Jini service lookup with Authorization

7. The proxy contacts the service, sends all the necessary certificates and proves that it actually has the corresponding private key. The service checks this, and if the authorization is ok, allows access.

It is not necessary for the proxy to know its own private key. The framework gives the proxy a handle to the key, which allows signing data but does not reveal the actual private key. This way the key can be "revoked" immediately by simply setting a flag in the handle object.

Why does the proxy need a private key then? We could give it an object capable of proving the user's secret key and the necessary certificates. The problem is that as the authorization certificates are considered public, the proxy would then be able to prove any authorization delegated to the user if it just obtains the corresponding certificate. The proxy might also use the key to fake identities. In addition to preventing these kinds of threats, the use of a temporary key makes it also possible for the user to delegate a more restricted set of permissions to an application and further to a proxy.

IV. Implementation architecture

Our implementation is responsible for proving user authorizations to service, authenticating proxies and verifying authorizations. It consists of the following packages.

- `siesta.security.core` package contains the most important classes, such as `ProxySecurityManager`, which is responsible for proxy security.
- `siesta.security.spki` contains functions for accessing, encoding and decoding SPKI certificates, as well as verifying certificate signatures. The implementation is quite generic.
- `siesta.security.authorization` defines the SIESTA-specific semantics for SPKI certificates. That is, it defines *a meaning* for authorization tags and *an implementation* for processing them.
- `siesta.security.repository` contains a simple local certificate repository for storing authorization certificates, and `CertificateGatherer` which tries to find a complete certificate chain when proving authorization.

A. Functionality

The most important changes to the basic Jini functionality are included in the service registration, proxy download, and service access functions. The security-related functional modifications are explained next.

1) Registering a service: In Jini, each server must register the services it provides to a lookup service. This makes the services available for clients. To use the security framework, the service must know its own key pair, its service certificates (explained below), and the key used for signing the the proxy code. Service registration is performed as follows.

1. The service initializes a proxy as normally in Jini.
2. The service signs the proxy using `ProxySigner`. The `ProxySigner` stores to the proxy a number of items, including the service public key, the service certificates, a computed code certificate, and a computed data certificate. The code and data certificates are explained in detail in Appendix A.
3. The service sends the proxy to the lookup service as normally in Jini, but now the proxy contains a bunch of security related information.

2) Retrieving a service proxy: In order to access a service, an application must retrieve a proxy from a lookup service. This happens as usually in Jini. However, before the proxy can access any security functions, it has to register itself with `ProxySecurityManager`.

1. The `ProxySecurityManager` checks that is was really invoked by the proxy.
2. If the check is passed, the `ProxySecurityManager` creates a new `ProxySecurityAssociation` object and a temporary proxy key, and calls `ProxyVerifier.verify`.
3. The `ProxyVerifier` asks the proxy to calculate a message digest of its data, using `SignedProxy` interface method `updateDataDigest`, and verifies the data signature, the code signature and the service certificates.
4. If the verification is passed, the `ProxyVerifier` stores service public key and service information to a `ProxySecurityAssociation`. These allow the proxy to create credentials for accessing the service.

Since our initial application is a personal calendar, the concept of the service "owner", which has a public key distinct from the service key, is important. The service certificates are written by the owner to the service key, and contain the name of the Java interface implemented by the service (for example, `siesta.pim.calendar.CalendarService`). These certificates, together with conventional identity certificates (binding a human-readable name to a public key), can be used by an application to securely bind the owner's name and a specific proxy object together (for example, when the name is shown to the user).

3) Accessing the service: When the application wants to use the service, it invokes a service method on the proxy.

1. The proxy calls `ProxySecurityManager.getAuthorization`. The `ProxySecurityManager` finds the corresponding proxy registration, checks that the tag has the correct service key and calls `ApplicationAccessController` to check local application authorization.
2. The `ApplicationAccessController` inspects the Java call stack, and checks that the calling application is authorized to access this kind of service.
3. The `ProxySecurityManager` creates a short-lived certificate (from the user key to the proxy key). This certificate, together with other relevant certificates returned by `CertificateGatherer`, is then given to the proxy.
4. The proxy opens secure communication channel to service, authenticates the service key, and authenticates itself using the proxy key.
5. The proxy sends the service request and certificates. The service creates a new `ChainVerifier` object, creates a tag corresponding to the service request, and calls `ChainVerifier` to check it.
6. The `ChainVerifier` checks the individual certificates, and for each certificate chain, checks if it implies the requested tag.
7. Finally, if the authorization is OK, the service performs the requested action.

V. Conclusions

In this paper, we have briefly outlined how to add decentralized authorization and trust management to Jini by using strong cryptography and authorization certificates. In particular, we have developed a Jini security library that allows service proxies to dynamically request credentials that allow them to send requests to the corresponding services. In order to use such credentials, the application hosting the proxy must have appropriate administratively assigned permissions in the form of SPKI certificate chains. The resulting system allows fine-grained access control in distributed Jini environments.

We have implemented the first prototype version of the library, and our next goal is to finish the implementation and make it publicly available.

Acknowledgments

Special thanks go to the rest of the SIESTA team: Antti Mannisto, Petra Pietiläinen, and Satu Virtanen.

References

[1] Matt Blaze, Joan Feigenbaum, and Jack Lacy, "Decentralized Trust Management", In *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1996.

[2] Carl Ellison et al., *SPKI Certificate Theory,* RFC 2693, September 1999.

[3] Ronald Rivest and Butler Lampson, "SDSI - A Simple Distributed Security Infrastructure", *Proceedings of the 1996 Usenix Security Symposium*, 1996.

[4] Sanna Liimatainen et al., *Telecommunications Software Security Architecture*, Helsinki University of Technology, `http://www.tcm.hut.fi/Research/TeSSA`

[5] Pekka Nikander and Jonna Partanen, Distributed Policy Management for Java 1.2, in *Proceedings of Network and Distributed System Security Symposium,* 4-5 February 1999, San Diego, CA.

[6] Pekka Nikander, *An Architecture for Authorization and Delegation in Distributed Object-Oriented Agent Systems,* PhD Thesis, Helsinki University of Technology, March 1999.

[7] Jonna Partanen, *Using SPKI certificates for Access Control in Java 1.2,* Master's Thesis, Helsinki University of Technology, August 1998.
[8] Jim Waldo, The Jini Architecture for Network-Centric Computing, *Communications of the ACM,* Vol. 42, No. 7, July 1999.

APPENDIX I DESIGN NOTES

1) How the proxy code signing works: Since the introduction of JDK 1.1, Java has had the facilities for signing JAR files. The signature is stored as a triple (public key, message digest, signature), together with some (signer name, public key) certificates. However, this signing method has one serious shortcoming: it's not possible to set "expiry dates" for the signatures. In this project, we wanted that possibility, so we had to make some modifications.

There are basically two ways of achieving the expiration. The straight-forward way is to change the JAR file signature to contain the validity information. Using SPKI certificates this could look like this:

```
(issuer (public-key serviceKey)
 subject (object-hash jarMsgDigest)
 tag "siesta.system proxyCode"
 validity)
```

This would, however, require modification to the JAR file loading code. This is by no means impossible; it has been done in the TeSSA project. In this project, however, we decided to use another approach.

Our approach splits the signature to two parts. Instead of storing (public key, message digest, validity, signature) we store (public key 1, message digest, signature) using standard JAR signing code, and (public key 1, public key 2, validity, signature) using our own code. The combination of these is the desired result. Furthermore, our approach has a number of additional benefits:

- We don't have to re-sign the JAR file if it hasn't changed. Since the JAR files are stored on a web server, the application might not be able to easily access them.
- We don't have to modify the JAR file loading code.
- We can use existing JDK tools for creating a part of the signature.

The main drawback is that the signature expiry date isn't visible to standard Java components, only to our own special code (which knows how to combine the two certificates).

2) How the proxy data signing works: In addition to verifying the authenticity of the proxy bytecode, we would like to be able to verify the proxy data as well. The straight-forward way would be to calculate the message digest of the serialized proxy object, create a certificate, and store the certificate in the Jini lookup service. However, there are a number of problems with this:

- The application communicating with the lookup service would have to know how to get the certificate and pass it on to ProxyVerifier.
- Java deserialization code would have to be modified to calculate the digest before deserialization.

We solved this by asking the proxy object to calculate its own message digest. The proxy bytecode has been verified in this point, so the proxy isn't completely untrusted. The implications of doing this aren't yet quite clear. It seems that the service can't gain any advantage by returning the wrong message digest, since it could sign it anyway. On the other hand, a lazy service writer could defeat this check by always returning the same message digest (for example, zero).