A Security Infrastructure for Distributed Java Applications

Dirk Balfanz
Princeton University
balfanz@cs.princeton.edu

Drew Dean
Xerox PARC
ddean@parc.xerox.com

Mike Spreitzer
Xerox PARC
spreitze@parc.xerox.com

Abstract

We describe the design and implementation of a security infrastructure for a distributed Java application. This work is inspired by SDSI/SPKI, but has a few twists of its own. We define a logic for access control, such that access is granted iff a proof that it should be granted is derivable in the logic. Our logic supports linked local name spaces, privilege delegation across administrative domains, and attribute certificates. We use SSL to establish secure channels through which principals can "speak", and have implemented our access control system in Java. While we implemented our infrastructure for the Placeless Documents System, our design is applicable to other applications as well. We discuss general issues related to building secure, distributed Java applications that we discovered.

1. Introduction

While everyone knows that distributed systems need cryptography to be secure, amazingly little cryptography is actually deployed today. This is particularly true inside corporate networks, where a firewall is assumed to keep all of the "bad" people out. The failure to deploy cryptography more widely has many causes:

- Policy issues (*e.g.*, export controls) have splintered the market for software.
- The X.509 standard has a hierarchical model for certification authorities (CAs). Of course, everyone wants to be at the root of the tree. This has generated much discussion, but little action. Corporations generally want a hierarchy under their own control.
- Lacking a common root CA, it is difficult to authenticate across CA boundaries.

Even if these issues were solved, the question remains whether identity-based certificates (binding names to cryptographic keys) are actually useful. More recent work emphasizes attribute certificates as a more scalable answer (see, for example [6]). The computer does not care *who* the user is, only whether or not the user should be able to perform some action.

We have designed and implemented a cryptographic infrastructure for the Placeless Documents System, a large, distributed middleware package written in Java. We attempt to carefully navigate most of the perils described above, *e.g.*, by allowing, but not requiring, a centralized CA. Our work owes much to the SDSI/SPKI work [7], but when we began work, no satisfactory implementation of those systems was available in Java. We took a principled approach to the design of our access control mechanism, so that requests supply credentials that lead to a proof that a request is valid.

The rest of this paper is structured as follows: In Section 2, we give an overview of the Placeless Documents System and of our overall design. In Section 3 we present our design in more detail: We define *principals*, *permissions*, and introduce our access control logic. In Section 4 we highlight a few aspects of our implementation. In Section 5, we discuss our experience working with access control logics. In Section 6, we discuss future work, and Section 7 concludes.

2. Overview

2.1. The Placeless Documents System

The Placeless Documents System is a distributed document management system developed at Xerox PARC [4]. Its major features include

- the ability to manage seamlessly documents of different kinds (*e.g.* files, email messages, Web pages, etc.), and
- a query language over document *properties* (as opposed to a directory hierarchy in which documents are stored), which allows to retrieve specific documents from their storage location.

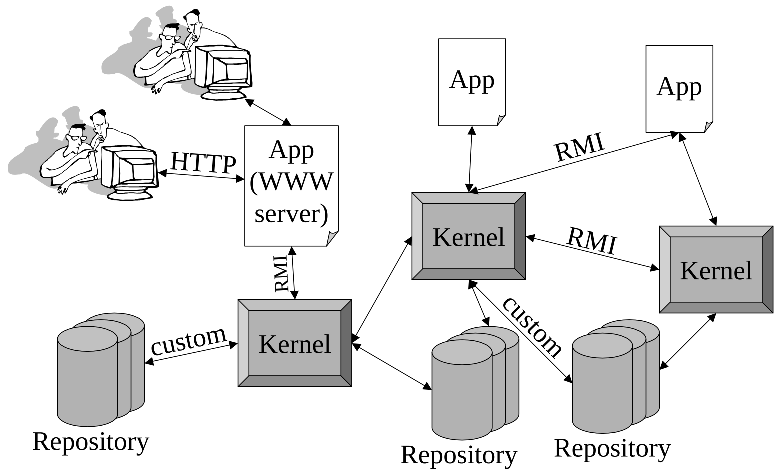

Figure 1. Placeless Documents. Kernels communicate with each other, and with Placeless applications, via RMI. Some applications may be used by multiple users at the same time. Kernels might need to manage documents owned by different users.

For the purposes of this paper, we focus on the following properties of the Placeless Documents System and its current implementation:

- It is a heavily distributed system. It is implemented in Java. The various components of the system communicate via Java RMI.
- Every user of the system runs one *kernel*, which manages the *document space* for that user. A space is the conceptual entity that holds all of a user's documents. A kernel is the actual program that implements the space abstraction.[1]
- Applications that would like to use the Placeless Documents System act as clients of kernels. All communication with kernels is via Java RMI. Kernels can also communicate with each other, if it turns out that a document is managed by another kernel than the one it was originally requested from.
- The document contents are not stored inside Placeless. Instead, Placeless stores document contents in their natural *repositories*, *e.g.* files on the file system, Web pages on a Web server, and so forth.
- Every document is managed by exactly one kernel. If a different kernel asks for access to that document, the other kernel must access the document through the document's managing kernel. The kernel is responsible for enforcing access control.

[1]Actually, a user can run more than one kernel. However, there is always a standard kernel, or *gatekeeper*, that lets the user access his or her space.

Typical Placeless applications include browser-like graphical interfaces that let users manage their (and others) documents. There also exist bridges between Placeless and, for example, the World Wide Web, enabling users to access Placeless documents with any Web browser. Note that in this last case, different users use one and the same Placeless application (which acts as a Web server at the front end and connects to Placeless kernels at the back end) to access their respective documents.

Figure 1 shows the various components in the Placeless Documents System and how they interact.

2.2. Authentication

To facilitate authentication of the various Placeless components to each other, we changed Java RMI to use the Secure Socket Layer (SSL) [8] as the underlying transport. This gives us two properties:

1. We assure the confidentiality and integrity of the communication between two components.
2. Components can authenticate each other, *i.e.* they learn at least the public key of the party they are communicating with.

2.3. Access Control

Every kernel has a policy that specifies what the various principals are allowed to do with documents managed by

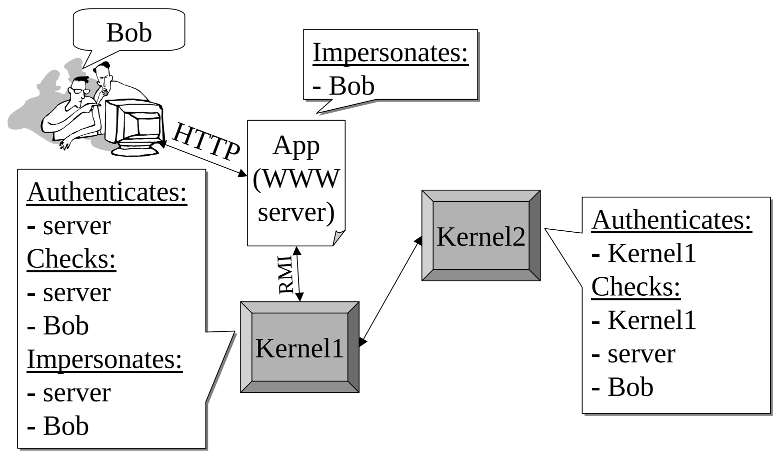

Figure 2. Impersonation in Placeless is similar to a quoting chain in ABLP logic.

that kernel. A policy is a set of *statements*. There are statements about which names are used for which principals, and about what kind of privileges are extended to those principals. The statements are actually expressions of an access control logic (see Section 3.3). When a client requests an operation to be performed in a kernel, that kernel knows what principal is requesting the operation, because the request came over an authenticated channel, and what operation is to be performed. That request then is also expressed as a statement of said access control logic. An inference engine can then verify whether the policy supports that request.

Clients of kernels can *impersonate* principals. That means that they announce on whose behalf they act. They are free to lie about on whose behalf they are acting. The kernel performing the operation will authenticate the client, and will then make sure that both the client and the principal it impersonates have enough privileges to perform the operation. In fact, if a kernel gets a request that is already identified as an impersonation and needs to turn around to another kernel to have that request answered, then the other kernel will have to check three principals: the first kernel, the first kernel's direct client, and the principal that client is impersonating. This is similar to a quoting chain "A says that B says that C says that X" in ABLP logic [2]. There, too, we know that A is making the statement, but we do not know whether A is lying, *e.g.* whether B really said what A is claiming. Figure 2 illustrates this approach. The fact that we do not authenticate principals other than the immediate peer of an SSL connection does not represent a security hole, since lying does not allow an attacker to gain any advantage. We provide the ability to impersonate other principals merely as a form of the principle of least privilege. Presumably, the Web server in Figure 2 has a lot of privileges. It impersonates other principals to make sure that it is only exercising privileges that the impersonated principal also has.

3. Design

3.1. Principals

There are five different kinds of principals in our system. They are all represented as subclasses of our Java class `Principal`, and explained in detail below:

3.1.1. LocalName. A LocalName is a string referring to a principal in someone's name space. For example, "Bob" is a LocalName. Which principal this LocalName refers to depends on the name space in which it is evaluated. Alice might call a different principal "Bob" than Charlie does. We call a LocalName *relative* because its interpretation is relative to some name space.

3.1.2. GlobalName. A GlobalName is a public key. The reason we call public keys "global names" is that it is like a string that everybody agrees on how to interpret. While we might both have different ideas about who "Bob" is, we do not have different ideas about who the principal with the public key "0x43453456 … " is. We call a GlobalName *absolute* since it does not depend on a name space.

3.1.3. ReferencePrincipal. A ReferencePrincipal is a series of LocalNames or GlobalNames. If there is a GlobalName in the series, we call the ReferencePrincipal *absolute*, otherwise we call it *relative*. Relative ReferencePrincipals are relative to a principal's name space. Absolute

ReferencePrincipals are global in the sense that every principal will agree on the identity of an absolute ReferencePrincipal (see Section 3.3 on why that is). The ReferencePrincipal

$$ReferencePrincipal(LocalName(Bob), LocalName(Alice))$$

is the principal called "Alice" in the name space of the principal we call "Bob". From now on, we will simply write $Bob's\ Alice$ instead.

3.1.4. SelfPrincipal. The SelfPrincipal is the principal representing the local system. The SelfPrincipal has a private key to sign certificates, authenticate itself over an SSL connection, and so forth. The identity of the SelfPrincipal is established when a Placeless client or kernel is started. The user has to supply a private key to the Placeless client or kernel to use for its SelfPrincipal. In this paper, we will call the SelfPrincipal simply $Self$.

3.1.5. AnyPrincipal. The AnyPrincipal represents the all-embracing group that everybody is a member of.

3.1.6. Subset Relationships of Principals. We define a subset relationship over principals. If X and Y are principals, then we say that $X \subseteq Y$ if either X and Y are identical or Y is the AnyPrincipal.

3.2. Permissions

In our system, we use Java Permission classes. While we do not use any of the pre-defined permission classes to describe possible actions in Placeless, we do use the notion that permissions can *imply* each other [9]. Permission A implies permission B if the set of possible actions that permission A describes is a superset of the set of possible actions that permission B describes. For example, the permission `ReadPermission(/usr/local/*)` implies the permission `ReadPremission(/usr/local/foo)`.

As we will see below, we will also be interested in the *intersection* of permissions. Formally, we define the intersection of permission A and B, $A \sqcap B$, as

$$\begin{aligned} &A \sqcap B = X \\ &\quad \text{iff } (A \rightarrow X) \wedge (B \rightarrow X) \\ &\quad \wedge (\forall Y\ s.t.\ A \rightarrow Y \wedge B \rightarrow Y . X \rightarrow Y) \end{aligned}$$

(Read $\rightarrow$ as "implies.") As it turns out for the kinds of permission classes we use, the intersection $A \sqcap B$ is always either A, B, or the *empty permission*, *i.e.* the permission describing no action at all, which is implied by all other permissions.

In Placeless, we have permission classes denoting the following actions:

- Reading from documents. This includes reading document contents as well as reading the values of document properties.
- Writing to documents and deletion of documents. This includes the modification, deletion, and creation of properties.
- Creation of documents.
- Notification about events happening in kernels.

These are called primitive permissions. Our logic treats them as uninterpreted symbols. We parameterize our logic over a finite set of primitive permissions. We also have special permissions denoting name bindings of principals and permission delegation, which are discussed below.

3.3. Access Control Logic

We have defined an access control logic that we use for Placeless[2]. The logic heavily borrows from SDSI and SPKI [7] and can be seen as (yet another) attempt to formalize systems of that kind. It is also influenced by ABLP-style [2] logics and tries to fit naturally with the Java `Permission` classes, which is a reason why we did not follow the SDSI/SPKI approach by the letter.

Expressions of the logic are *statements*. A statement is of the form

$$Principal : Permission,$$

where *Principal* is the principal making the statement. We can interpret a statement as meaning "the principal making the statement asserts that it is ok to perform the action associated with the permission mentioned in the statement". So, the statement

$$LocalName(Bob) : ReadPermission$$

says that Bob wants to read a document, while the statement

$$Self : ReadPermission$$

means that it is ok to read the document. The colon can roughly be pronounced as "says" with similar meaning as in ABLP logic.

[2]We should note that the access control logic can be used in more general settings as well, it is not particularly customized for Placeless.

Policies are expressed as a list of statements. As we will see later, the statement

$$\mathit{Self} : \mathit{DelegationPermission}(\mathit{LocalName}(\mathit{Bob}), \mathit{ReadPermission})$$

means that we give Bob read permission. The statement

$$\mathit{Self} : \mathit{BindingPermission}(\mathit{Mother's\,Brother}, \mathit{LocalName}(\mathit{Uncle}))$$

means that we locally bind the name "Uncle" to the principal that is bound to the name "Brother" in the name space of the principal we call "Mother".

Before we introduce the inference rules of our logic, we need to look at two special permission classes.

3.3.1. DelegationPermission. A DelegationPermission is a subclass of `java.security.Permission`. The constructor of a DelegationPermission takes two arguments: A principal, and a permission. As far as our logic is concerned, a DelegationPermission is a term of the logic that also takes a principal and a permission:

$$\mathit{Principal} : \mathit{DelegationPermission}(\mathit{Principal}, \mathit{Permission}).$$

As will become apparent from the inference rules of the logic, the statement

$$\mathit{Self} : \mathit{DelegationPermission}(X, P)$$

means that we have given permission P to principal X. As another example, the statement

$$\mathit{GlobalName}(\mathsf{SomeKey}) : \mathit{DelegationPermission}(X, P)$$

means that the principal with the key SomeKey has delegated permission P to principal X.[3]

For every new permission class we introduce, we need to define when an instance of this class implies another permission. A DelegationPermission can only imply other DelegationPermissions. $\mathit{DelegationPermission}(X, P)$ implies $\mathit{DelegationPermission}(Y, Q)$ if P implies Q and principal X is a superset of principal Y (*e.g.* if X is the AnyPrincipal).

3.3.2. BindingPermission. A BindingPermission is another subclass of `java.security.Permission`. Its constructor takes two principals, which are *bound* to each other. But note that this binding is not commutative — a $\mathit{BindingPermission}(X, Y)$ does not imply a $\mathit{BindingPermission}(Y, X)$. Rather, a BindingPermission establishes a kind of subset-superset relationship between the two principals. For example,

$$\mathit{Self} : \mathit{BindingPermission}(\mathit{Bob}, \mathit{Managers})$$

can be interpreted as saying that Bob is a member of the group Managers. Sometimes, it is easier to think of BindingPermissions as establishing a "speaks-for" relationship. For example,

$$\mathit{Self} : \mathit{BindingPermission}(\mathit{GlobalName}(\mathsf{BobsKey}), \mathit{LocalName}(\mathit{Bob}))$$

says that the key BobsKey speaks for the principal I call Bob.

A BindingPermission cannot imply permissions other than BindingPermissions. $\mathit{BindingPermission}(X, Y)$ implies $\mathit{BindingPermission}(X', Y')$ if $X' \subseteq X$ and $Y' \subseteq Y$ (for example, if X or Y are the AnyPrincipal).

3.3.3. Why Binding and Delegation Permissions Are Separate. Both BindingPermission(..., ...) and DelegationPermission(..., ...) are instances of a more general "speaks for" relation. Why do we separate them? We keep them separate to model the separation found in a corporate enterprise: a human resources or corporate security office is in charge of issuing a credential saying that John Doe is an employee of Company X, while Doe's management chain (or other colleagues) are responsible for setting access controls on their documents. Thus, while we support attribute certificates, we also find names to be useful for access control: people are used to thinking in terms of names, and names provide a useful level of indirection to keys. We write policies in terms of names, not keys, so that a user's key can change over time without requiring changes to the security policy.

3.3.4. Inference Rules. Our access control logic includes inference rules that allow us to infer new statements from a set of statements already held to be true. The inference rules are introduced below:

The first inference rule is called *Delegation*:

$$\frac{\begin{array}{l}\mathit{Self} : \mathit{DelegationPermission}(X, P)\\ X' : P'\\ X' \subseteq X\end{array}}{\mathit{Self} : P \sqcap P'|_{X'}} \qquad \text{(Del)}$$

[3]The name X should be evaluated in the name space of the principal making the statement, as should any principal that appears within P.

Here, the bar operator "|" localizes principals within permissions. It is defined as follows:

$$\begin{aligned}
\mathit{DelegationPermission}(Y,P)|_X &= \\
&\mathit{DelegationPermission}(Y@X,P|_X) \\
\mathit{BindingPermission}(Y,Z)|_X &= \\
&\mathit{BindingPermission}(Y@X,X's\,Z)
\end{aligned}$$

For all other permissions, it is defined as

$$P|_X = P.$$

The @-operator is similar to a ReferencePrincipal, but not the same:

$$\begin{aligned}
Y@X &= X's\,Y && \text{if } Y \text{ is relative} \\
&= Y && \text{if } Y \text{ is absolute}
\end{aligned}$$

For example, recall that this is how we would give Bob permission to read a document:

$$\begin{aligned}
&\mathit{Self} : \mathit{DelegationPermission}(\mathit{LocalName}(\mathit{Bob}), \\
&\qquad\qquad\qquad\qquad \mathit{ReadPermission})
\end{aligned}$$

Bob's request to read the document can be represented in the logic as

$$\mathit{LocalName}(\mathit{Bob}) : \mathit{ReadPermission}$$

Application of rule (Del) now yields

$$\mathit{Self} : \mathit{ReadPermission},$$

which essentially means that it is ok to go ahead and read the document.

Here is another example: Let us assume that we already believe the statements

$$\begin{aligned}
&\mathit{Self} : \mathit{DelegationPermission}(\\
&\qquad \mathit{AnyPrincipal}, \mathit{DelegationPermission}(\\
&\qquad\qquad \mathit{AnyPrincipal}, \mathit{ReadPermission}))
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{LocalName}(\mathit{Bob}) : \mathit{DelegationPermission}(\\
&\qquad \mathit{LocalName}(\mathit{Alice}), \mathit{ReadPermission})
\end{aligned}$$

We can apply rule (Del), since $\mathit{LocalName}(\mathit{Bob})$ is a subset of *AnyPrincipal.* The resulting statement is

$$\mathit{Self} : \mathit{DelegationPermission}(\mathit{Bob's\,Alice}, \mathit{ReadPermission})$$

The second inference rule is called *Transitivity*:

$$\frac{\begin{array}{l}\mathit{Self} : \mathit{BindingPermission}(X,Y) \\ \mathit{Self} : \mathit{BindingPermission}(Y,Z)\end{array}}{\mathit{Self} : \mathit{BindingPermission}(X,Z)} \quad \text{(Trans)}$$

The third inference rule is called *Name Space Linking*:

$$\frac{X : \mathit{BindingPermission}(Y,Z)}{\mathit{Self} : \mathit{BindingPermission}(Y@X, X's\,Z)} \quad \text{(Link)}$$

The fourth inference rule is called *Containment*:

$$\frac{\begin{array}{l}\mathit{Self} : \mathit{BindingPermission}(X,Y) \\ X : P\end{array}}{Y : P} \quad \text{(Cont)}$$

There are more inference rules that allow unrestricted delegation (*i.e.* delegation through an arbitrary number of levels of nesting), but they are not part of our current implementation, and are not relevant for the discussion in this paper.

3.3.5. Access Control Decisions. As we have seen in Figure 2, a request to perform a certain action is presented to a kernel together with a list of principals. The kernel has authenticated the first principal itself, but the other principals are not authenticated, they are merely impersonated. The immediate client to our kernel may or may not faithfully relay who *it* authenticated before it turned to us.

For every principal in this list, our Placeless kernel creates a statement describing the requested action. For example, if the immediate client was authenticated as having key K, and the requested operation is to read a document, then the statement describing the requested action is

$$\mathit{GlobalName}(K) : \mathit{ReadPermission}$$

This statement is added to the list of statements describing the local policy (see Section 4.2), and to the statements derived from certificates also presented by the client. Then, an inference engine tries to deduce the statement

$$\mathit{Self} : \mathit{ReadPermission}.$$

If that succeeds for every principal in the list, access is allowed, otherwise it is denied.

3.3.6. An Example. Let us go through an example to illustrate the concepts of policies and proving of authorization. We will abbreviate as follows: *BindingPermission* will become *Bind*, *DelegationPermission* will become *Delegate*,

$$
\begin{array}{rlr}
(1)(5)-(Cont) & Bob : Bind(K_L, Lab) & (9) \\
(1)(6)-(Cont) & Bob : Bind(Lab's\ Alice, secretary) & (10) \\
(1)(7)-(Cont) & Bob : Delegate(secretary, Read) & (11) \\
(9)-(Link) & Self : Bind(K_L, Bob's\ Lab) & (12) \\
(10)-(Link) & Self : Bind(Bob's\ Lab's\ Alice, Bob's\ secretary) & (13) \\
(8)(12)-(Cont) & Bob's\ Lab : Bind(K_A, Alice) & (14) \\
(14)-(Link) & Self : Bind(K_A, Bob's\ Lab's\ Alice) & (15) \\
(3)(11)-(Del) & Self : Delegate(Bob's\ secretary, Read) & (16) \\
(13)(15)-(Trans) & Self : Bind(K_A, Bob's\ secretary) & (17) \\
(4)(17)-(Cont) & Bob's\ secretary : Read & (18) \\
(16)(18)-(Del) & Self : Read &
\end{array}
$$

Figure 3. Proof for the example in Section 3.3.6

ReadPermission will become *Read*, and we will drop the *LocalName* and *GlobalName* terms around keys and names - K_X is a GlobalName, and names such as Bob, Alice, etc. are LocalNames.

In this example, we know the key of our boss, Bob, and have made an appropriate entry in our policy:

$$Self : Bind(K_B, Bob) \tag{1}$$

We have also given Bob ReadPermission, and have given him permission to delegate that permission:

$$Self : Delegate(Bob, Read) \tag{2}$$

$$Self : Delegate(Bob, Delegate(AnyPrincipal, Read)) \tag{3}$$

A client has connected to our kernel. During the SSL handshake, we have learned that the key of that client is K_A. The client is trying to read one of our documents. This can be expressed as

$$K_A : Read. \tag{4}$$

In addition, the client presents certificates. These are Delegate- or BindingPermissions signed by principals. We interpret them as statements made by the key that signed them. The first one said that the principal with key K_B (Bob) has bound the name "Lab" to the key K_L:

$$K_B : Bind(K_L, Lab) \tag{5}$$

Bob also tells us that he has bound the name "secretary" to the principal called "Alice" in the name space of the principal he calls "Lab"

$$K_B : Bind(Lab's\ Alice, secretary) \tag{6}$$

and that he has delegated his ReadPermission to the principal he calls secretary:

$$K_B : Delegate(secretary, Read) \tag{7}$$

The last certificate our client presents to us is signed by the key K_L, and states that in K_L's name space, the name "Alice" is bound to the key K_A:

$$K_L : Bind(K_A, Alice). \tag{8}$$

From statements (1) through (8) we will have to prove that $Self : Read$ holds.

Figure 3 shows the proof. On the left, we show which statements and inference rules are needed to deduce the new statements, which are numbered on the right.

4. The Implementation

We have implemented our access control system for Placeless using the Java 2 Platform JDK. We use the IAIK [11] cryptographic provider and SSL implementation. We provide classes for the various types of principals and permissions, and we implemented an inference engine for our access control logic. We also implemented tools that hide most of the underlying mechanisms and should make it easy for Placeless users to manage their day-to-day security settings.

4.1. Running RMI over SSL

Since the release of the Java 2 Platform, it has been relatively easy to change the transport mechanism underlying RMI. We chose to use SSL to gain both privacy and authenticated communication. In Placeless, the SSL layer always requires client authentication and uses a specific cipher suite[4] since there is no need for cipher suite negotiation. In SSL, the handshake between client and server can fail if either client or server do not authorize the other party to connect. In Placeless, we always allow two parties to establish an SSL connection with each other. We note the identity of the other side of the communication link and defer access control decisions to higher layers in the system.

When an RMI client wants to talk to an RMI server, it uses *RMI stubs*, which are objects that impersonate remote (server) Java objects in the local VM and forward calls to the (remote) VM. If the client does not have the stub code available locally, it can download it from the server. In the case of a non-standard transport mechanism (such as SSL), the client might also have to download code that implements the non-standard transport mechanism. Note that in the case of SSL this presents a problem. The client needs a trustworthy SSL implementation so that (a) it can trust the identity of the other end of the connection, and (b) the client believes that nonces and session keys are properly generated. However, in certain situations it might acquire the code for that SSL implementation from the very party it is trying to authenticate. This is clearly a security hole. In our current implementation, we just disallow dynamic downloading of stub code. This, however, disables a whole set of features for distributed Java programs, so that a more satisfying solution is yet to be found. Unfortunately, the Java 2 RMI implementation does not provide hooks to control the downloading of stub classes via RMI.

The SSL implementation we used provides new `Socket` classes, which can be used just like normal TCP sockets, but implement SSL functionality. An `SSLSocket` provides a method to query the identity (*i.e.* the X.509 certificate) of the other party of an SSL connection. This feature is lost in the "higher" RMI layer – an RMI server object has no way of knowing through which specific SSL connection a call to one of its methods has been invoked. This, however, is necessary to establish the identity of the party originating the call and to perform access control decisions. There is no easy way to add this functionality. After all, RMI is designed to *hide* the transport layer from upper layers, since it is supposed to be transport layer independent. It turns out, though, that in the current implementation of RMI the thread in which a communications socket is created for an RMI connection is the same thread in which calls to the server object associated with this connections are executed. There is a new thread for every RMI connection. While this may be questionable in terms of performance, it gave us a handle to track identities of SSL peers into the RMI layer: When an SSL socket is created, it notes which thread created it. Later, when an RMI server object executes a call that was initiated by a remote party, it can again check which thread is currently executing, and lookup which `SSLSocket` is associated with that thread. This way, RMI server objects can learn the identity (an X.509 certificate) of the parties invoking calls on them.

[4] 1024-bit RSA for key exchange, 128-bit RC4 for encryption, and SHA for MACs.

4.2. Policies

Calls to a remote Placeless kernel are handled by RMI server objects. They pass the identity of the caller, together with certificates the caller provides[5] and a statement describing the intended action, to a reference monitor. That reference monitor, just like the Java `SecurityManager`, then makes a decision whether to proceed with the call or not. The decision will be based on the *policy* associated with the object on which the proposed action is to be performed.

As noted earlier, a policy is a set of statements in our access control logic. All statements are of the form

$$\begin{array}{ll} \mathit{Self} & : \ \mathit{BindingPermission}(\ldots,\ldots) \\ \text{or} & \\ \mathit{Self} & : \ \mathit{DelegationPermission}(\ldots,\ldots) \end{array}$$

We store the *BindingPermission* statements in the user's *name space*. The *DelegationPermission* statements (also sometimes referred to by themselves as the "policy" as opposed to the name space) are stored with the documents. This is similar to simple access control lists, which are also stored with the document they are protecting. There is a possible level of indirection for Placeless policies, though: A document may, instead of specifying its own policy, specify the name of a policy. That policy is then looked up in the user's *policy document* (a collection of policies). If a document does not specify a policy for its access control, the policy called "default" from the user's policy document is used.

Sometimes, certain actions do not pertain to documents. For example, which policy should we consult when we are about to create a new document? The document does not exist yet, so it cannot specify a policy governing its creation. For these types of actions we consult the policy called "space" from the user's policy document. It is up to the reference monitor to decide which policy to apply. In our implementation, the reference monitor will first decide whether the proposed action calls for the space policy or for a document-specific policy. If the action needs

[5] These are signed statements of our our access control logic. A client might, for example, provide certificates to prove that someone delegated privileges to him.

a document-specific policy, it tries to fetch the policy from the document. This can either be an actual policy (a set of statements), or a name. In the latter case, the reference monitor resolves the name against the list of policies stored in the user's policy document. If there is no policy with the given name, the "default" policy is used.

The reference monitor will then add the *BindingPermission* statements from the user's name space, together with the statements extracted from the caller's certificates and the statement describing the intended action, to the policy and engage the inference engine to try to prove that access should be granted (see Section 3.3.5).

4.3. Certificates

Certificates are signed BindingPermissions or DelegationPermissions. A client presents certificates with his request. Certificates are interpreted as statements made by the key that signed them. In our implementation, we cache certificates on the kernel side so that for a given SSL connection, a client has to present its certificates only once. Our implementation does not support certificate revocation lists. Instead, our certificates expire. The inference engine will not take into account statements gained from expired certificates.

4.4. Inference Engine

As explained in Section 3.3.5, the inference engine has to find a derivation of a statement of the form

$$\mathit{Self} : \mathit{Permission}$$

from the set of statements gained from the policy, name space, certificates, and the requested action. We implemented a simple forward-chaining inference engine: We keep generating new statements from already existing ones. Since the set of derivable statements is finite, this process will terminate, and our inference engine is sound and complete with respect to our logic.

4.5. Tools

We implemented several tools for users to handle our access control system.

Key creation: We use the java keytool to create 1024-bit RSA keys and self-signed X.509 certificates. The X.509 certificates are not used for any purpose except to exchange public keys in the SSL handshake.

Key export and import: There is a tool to export the user's public key to a file. Other users can then import that public key and map it to local names in their name spaces.

Name space manager: The name space manager is a tool that allows users to create statements of the kind

$$\mathit{Self} \quad : \quad \mathit{BindingPermission}(\ldots, \ldots).$$

for their own local name space. They can bind local names to keys, ReferencePrincipals, and other local names.

Policy tool: The policy tool allows users to make statements of the kind

$$\mathit{Self} \quad : \quad \mathit{DelegationPermission}(\ldots, \ldots).$$

and store them in named policies. In the current implementation, the kinds of supported statements are restricted. Permissions can only be granted to LocalNames, and we do not support the granting of DelegationPermissions or BindingPermissions.

The policy tool can also be used to create new policies, delete policies, and remove statements from a policy. There is also a tool that populates the "default" and "space" policy with reasonable default values.

Certificate tool: The certificate tool allows users to import certificates, *i.e.* signed statements of the form

$$\begin{array}{ccl} \mathsf{SomeKey} & : & \mathit{BindingPermission}(\ldots, \ldots) \\ & \text{or} & \\ \mathsf{SomeKey} & : & \mathit{DelegationPermission}(\ldots, \ldots) \end{array}$$

into their certificate collection. The certificate tool can also be used to export certificates of the form

$$\mathsf{UserKey} \quad : \quad \mathit{DelegationPermission}(\ldots, \ldots).$$

This is equivalent to delegating permissions to third parties.

The above-mentioned name space manager can be used to export certificates of the form

$$\mathsf{UserKey} \quad : \quad \mathit{BindingPermission}(\ldots, \ldots)$$

This will allow third parties to make statements about principals known in our name space.

5. Experience with Access Control Logics

We wanted to base our system on an access control logic because we wanted to be able to develop a formal model of the complex relationships between privilege delegation and linked local name spaces. This way, it is easier to reason about certain properties of the system.

Abadi, Halpern and van der Meyden [1, 10] have tried to formalize SDSI-style linked local name spaces and point out that there is no obviously right way to do so. While they can prove that their logics are sound with respect to a certain semantics, the choice of semantics is arbitrary. One cannot prove that a certain logic will not be surprising, *i.e.* allow intuitively undesirable conclusions. There has not been a published attempt to formalize the whole SPKI framework, which would combine the linked local name spaces with an access control framework. For our work, we attempted to do that, although on a smaller scale (*e.g.* we do not have the notion of a threshold principal).

We encountered similar "surprises" with our logic. For example, in an early version we had the following delegation rule:

$$\frac{\begin{array}{l} \mathit{Self} : \mathit{DelegationPermission}(P, X) \\ \quad X' : P' \\ \quad X' \subseteq X \end{array}}{\mathit{Self} : \mathit{DelegationPermission}(X', P \sqcap P')}$$

If we try to apply this rule to our example in Section 3.3.6, instead of statement (16) (see Figure 3) we would get:

$$\mathit{Self} : \mathit{Delegate}(\mathit{secretary}, \mathit{Read}).$$

This means that Bob issued a certificate that made us believe *our* secretary has ReadPermission. This is not desirable.

The logic, as presented here, has another problem: If we, instead of giving Bob ReadPermission, had given all managers ReadPermission (and the permission to delegate it):

$$\mathit{Self} : \mathit{Delegate}(\mathit{Managers}, \mathit{Delegate}(\mathit{AnyPrincipal}, \mathit{Read}))$$

and had said that Bob is a manager:

$$\mathit{Self} : \mathit{Bind}(\mathit{Bob}, \mathit{Managers})$$

then we could conclude that any manager's secretary has ReadPermission:

$$\mathit{Self} : \mathit{Bind}(\mathit{Managers}'s\ \mathit{secretary}, \mathit{Read}),$$

which is also not desirable.

The following modified containment rule rectifies this problem:

$$\frac{\begin{array}{l} \mathit{Self} : \mathit{BindingPermission}(X, Y) \\ \mathit{Self} : \mathit{DelegationPermission}(Y, P) \end{array}}{\mathit{Self} : \mathit{DelegationPermission}(X, P)}$$

The containment rule presented in Section 3.3.4 gives *BindingPermission* a "speaks-for" flavor: A member of a group can speak for the group. This is the kind of groups we find in ABLP logic [2]. However, in [10] the authors point out that a subset-relationship might be more appropriate to model name bindings. The modified containment rule presented above follows that approach: If we have delegated a permission to a group, we have delegated it to every member.

But now we no longer can prove the example in Section 3.3.6. We need an additional inference rule, for example Halpern and van der Mayden's *Monotonicity*:

$$\forall Z. \quad \frac{\mathit{Self} : \mathit{Bind}(X, Y)}{\mathit{Self} : \mathit{Bind}(X' s\ Z, Y' s\ Z)}$$

The universal quantifier greatly expands the search space, though it remains finite. However, this is not trivially implemented in our forward-chaining inference engine (see Section 4.4).

To find a logic that meets intuition about what is secure and what is not, and at the same time is efficiently implementable, remains an open problem. We think that the logic presented here is a good starting point, but expect some evolution towards a logic that will be less prone to "surprises," and hopefully will have enjoyed the scrutiny of the computer security research community.

6. Future Work

As we have just seen, this work has led to some interesting ongoing research. The most pressing and interesting question is whether our access control logic is "good". While it is relatively easy to prove that our logic is sound (*e.g.* by proving all our inference rules as theorems in a logic known to be sound, for example Felten and Appel's logic [3]), it is harder to see whether it will "surprise", *i.e.* allow conclusions that we intuitively would consider insecure (see [1] and [10] for some example on "surprising" conclusions found in earlier attempts to formalize SDSI). We tried to learn our lesson from previous attempts (see Section 5), but it is not clear if the logic as presented here is the final word.

Our current implementation of the inference engine is rather inefficient. On one hand, we are looking for a more effective inference strategy than unguided forward chaining. On the other hand, we need to implement a better caching mechanism that allows quick lookups of frequently asked access control queries.

Other topics that we are currently working on include:

Certificate dissemination Clients have to present certificates to the kernels to support their requests. How do clients find out which certificates are relevant for the requested action? Currently, clients supply all certificates they collected to kernels. Even though they are cached at the kernel side, this still is a performance problem. Some work has already been done in this area [5].

Securing the name sever Placeless uses a name server similar to the RMI registry to locate kernels. Currently, the name server does not follow a security policy when registering kernels under their respective names. It could therefore be possible for a rogue kernel to impersonate another kernel to a client. Clients could in theory test the identity of the kernel they are connected to (after all, they have authenticated it over an SSL connection), but in practice hardly ever do so. We need to secure the Placeless name server.

Legacy authentication Recall the WWW servers in Figure 1. When they authenticate a browser, they do not necessarily use strong SSL authentication, which yields the public key of the browser. They might use a different kind of authentication that just yields a user name (for example, if they use password-based authentication). How do we translate that user name into a Placeless principal? In our current implementation, we assume that the WWW server knows (say, by convention) the local name under which the authenticated user is known to the kernel the server is connected to. It will therefore impersonate a *LocalName* principal when it connects to the server. This only works if the Web server and the kernel(s) it talks to agree on the names of principals. We are currently looking for a better solution for this problem of legacy authentication.

Parameterized permissions In the current system, every Placeless document (conceptually) has its own policy. Therefore, most of our permission classes (*e.g.* ReadPermission, WritePermission) do not have any parameters. For example, if we add this statement to the policy of document Foo

$$\mathit{Self} : \mathit{DelegationPermission}(\mathit{Bob}, \mathit{ReadPermission})$$

then it is implicitly clear that the ReadPermission pertains to the document Foo. However, if Bob now wants to delegate his ReadPermission for Foo, he has no way of expressing this. He can only delegate his parameterless ReadPermission, effectively delegating his privilege to read any document he is allowed to. If our permissions took parameters (like the standard Java permission classes do), Bob could delegate his permission to read specific documents to third parties.

7. Conclusion

We have designed and implemented a distributed access control system for the Placeless Documents System. In addition to strong authentication, we also provide communication confidentiality and integrity.

Borrowing ideas from SDSI, our system can be used across administrative domains and does not require centralized certification authorities. At the same time, it is flexible enough to allow the establishment of certification authorities, should a group of users feel so inclined (all they need to do is to delegate a *BindingPermission(AnyPrincipal, AnyPrincipal)* to certification authorities they trust).

Policies are expressed as a set of statements of the kind

$$\mathit{Self} : \mathit{DelegationPermission}(\mathit{Principal}, \mathit{Permission})$$

and

$$\mathit{Self} : \mathit{BindingPermission}(\mathit{Principal}, \mathit{Principal}).$$

The permissions can take arbitrary arguments, and can therefore be used to express a wide range of security policies. For example, we could have an `IntervalPermission` class that would take three arguments: A not-before-date *A*, a not-after-date *B*, and a permission *P*. Then, we could define that

$$\mathit{IntervalPermission}(A, B, P)$$

implies permission *Q* if and only if *P* implies *Q* and the current time is between *A* and *B*. This way, we can limit arbitrary privileges to certain times. Other arrangements, and permission classes implementing them, are possible.

Central to our system is a simple access control logic that tries to capture the spirit of SDSI, SPKI and ABLP-style systems while naturally fitting within the Java framework. We were trying to strike a balance between overly inflexible systems like simple access control lists (which do not allow for delegation, for example, and usually cannot span administrative domains) on one side, and very general, but difficult-to-use systems like proof carrying authentication [3] (which usually put the burden of rather complex proofs on the requester of an operation). In our system, the requester of an operation only needs to supply a sufficient set of certificates, while the inference engine on the serving side constructs the proof on its own.

We find the SDSI approach of linked local name spaces very appealing, but encountered known subtleties in its formalization, especially when we tried to combine this with SPKI-like delegation statements (see also [10] and [1] for a discussion on the formalization of SDSI).

While work remains to be done, we believe a system like the one described in this paper provides a flexible framework for access control in the Placeless Documents System, and for distributed Java applications in general.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, October 1998.

[2] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, November 1999.

[4] P. Dourish, K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, and J. Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 1999. scheduled for publication.

[5] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[6] C. M. Ellison. Establishing identity without certification authorities. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, July 1996.

[7] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, September 1999. RFC2693.

[8] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol Version 3.0*. IETF - Transport Layer Security Working Group, The Internet Society, November 1996. Internet Draft (work in progress).

[9] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[10] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 111–122, Mordano, Italy, June 1999.

[11] Institute for Applied Information Processing and Communications, Graz University of Technology. *IAIK JCE*, 1999. `http://jcewww.iaik.tu-graz.ac.at/index.htm`.