HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science

Telecommunication Software and Multimedia Laboratory

Jonna Partanen

Using SPKI certificates for access control in Java 1.2

Supervisor: Professor Arto Karila
Instructor: Assistant Professor Pekka Nikander

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF MASTER'S THESIS**

Author:	Jonna Partanen		
Title:	Using SPKI certificates for access control in Java 1.2		
Date:	1998-08-12	**Number of pages:**	69
Faculty:	Information Technology		
Professorship:	Tik-110 Computer Networks		
Supervisor:	Professor Arto Karila		
Instructor:	Assistant Professor Pekka Nikander		

The new Java security architecture is enhanced to support fine-grained access control. In Java 1.2, classes are signed, and their access rights are locally configured, based on the signer's identity and the location where the class was loaded from. However, previous experience has shown that local configuration files do not scale well on the global Internet, but may create an administrative nightmare. A better alternative is needed.

In this Thesis, we describe the current Java 1.2 security architecture and propose distributed capability based policy management as an alternative to the local policy configuration files. We also discuss SPKI certificates and evaluate the possibility of using SPKI authorization certificates for delegating permissions to Java classes.

In addition to the local management, the current Java security policy has another shortcoming: it is static. The permissions in each protection domain reflect the policy configuration as it was when the domain was created, and not as it is at the moment when the permissions are checked. We suggest making the policy dynamic by deriving the permissions, when needed, directly from the SPKI certificates.

In this work we have created a prototype of the proposed system. The prototype includes all the crucial components, and forms a fully functional system. We also present performance and security analyses of our design. Finally, we conclude that SPKI certificates are suitable for distributing Java security management. Our prototype implementation has adequate performance and is more secure and scalable than the standard JDK implementation.

Keywords: access control, authorization, certificate, Java, SPKI

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Jonna Partanen		
Työn nimi:	SPKI-sertifikaattien käyttö Java 1.2:n pääsynvalvonnassa		
Päivämäärä:	1998-08-12	**Sivumäärä:**	69
Osasto:	Tietotekniikan osasto		
Professuuri:	Tik-110 Tietokoneverkot		
Työn valvoja:	Professori Arto Karila		
Työn ohjaaja:	Yliassistentti Pekka Nikander		

Javan uusi turva-arkkitehtuuri tukee hienojakoista pääsynvalvontaa. Java 1.2:ssa luokat allekirjoitetaan ja niiden pääsyoikeudet määritellään paikallisesti allekirjoittajan identiteetin ja latauspaikan perusteella. Aiempi kokemus on kuitenkin osoittanut, että paikalliset asetustiedostot eivät ole riittävän laajennettavia ratkaisuja maailmanlaajuista Internettiä ajatellen. Hallintaan on löydettävä parempi ratkaisu.

Tässä diplomityössä esitellään Javan turvallisuusarkkitehtuuri ja ehdotetaan hajautettua, kykenevyyksiin (capability) perustuvaa turvapolitiikan hallintaa vaihtoehdoksi paikallisten asetustiedostojen käytölle. Kuvailen myös SPKI-sertifikaatit ja arvioin mahdollisuutta käyttää niitä Java-oikeuksien delegointiin luokille.

Paikallisen hallinnan lisäksi nykyisessä Javan turvapolitiikassa on toinenkin ongelma: se on staattinen. Suojausalueiden (protection domain) ja sitä kautta luokkien oikeudet eivät kuvasta tarkistushetken tilannetta, vaan alueen luomishetkellä vallinnutta politiikkaa. Ehdotan turvapolitiikan muuttamista dynaamiseksi johtamalla oikeudet SPKI-sertifikaateista aina tarvittaessa.

Työssä olen toteuttanut prototyypin suunnitellusta järjestelmästä. Prototyyppi pitää sisällään kaikki keskeiset osat, ja muodostaa toimivan kokonaisuuden. Lisäksi esittelen toteutusta koskevan suorituskyky- ja turvallisuusanalyysin tulokset. Johtopäätöksenä on, että SPKI-sertifikaatteja voidaan soveltaa Javan turvapolitiikan hallinnan hajauttamiseen. Prototyyppini on teoreettisesti tarkasteltuna turvallisempi ja skaalautuvampi kuin JDKn toteutus, ja sen suorituskyky on riittävä.

Hakusanat: Java, pääsynvalvonta, sertifikaatti, SPKI, valtuutus

Preface

This Thesis was written as a part of the Telecommunications Software Security Architecture research project at the Telecommunications and Multimedia Laboratory at Helsinki University of Technology (HUT). Many people have contributed to the result.

I want to thank my professor, Arto Karila, for his support. He wanted me to keep a tight time schedule, and made me believe I could do it.

My special thanks to my supervisor, Pekka Nikander. He gave me ideas and helped me in developing them. He also proofread this thesis and gave valuable suggestions for improvement. Without him this work would have been a lot more of a struggle.

I also want to thank Nixu Oy for letting me take a leave of absence to do this research, and for letting me work in their premises until I got a desk of my own at HUT. The other members of the TeSSA project also deserve my thanks for their support.

Finally, I'd like to thank my family and especially my fiancé Camillo Särs for patience and advice. Thanks for enduring my moods and the fact that I have had so little time for you during this work.

Otaniemi 1998-08-12

Jonna Partanen

Table of Contents

Terms

Access control	Making sure that only legitimate subjects can access objects in the system.
ACL	Access Control List
Authorization	The act of granting a right to some entity.
BNF	Backus-Naur Form
Capability	A digital “ticket” that grants access to some resource.
Certificate	A digitally signed record that states the properties of some entity.
DAC	Discretionary Access Control
Denial of service	A situation where the system is unable to perform the actions requested by legitimate users, or some information is not available to the legitimate users when needed.
Issuer	A certificate field that specifies the entity that issued the certificate.
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
MAC	Mandatory Access Control *Also*: Message Authentication Code
Object	A resource in a computer system, for example a file, a directory, a memory address, a network connection, a process etc.
Reference monitor	A concept that illustrates the access control concept: each access to the objects of the system is requested from the reference monitor and it either allows or disallows the access.

Sandbox	A concept that illustrates the Java security policy of restricting untrusted applets' actions. Untrusted applets can play in the sandbox, but they cannot get out or affect entities outside.
Security policy	Defines the security requirements in a particular system, or in other words, specifies what actions each subject of the computer system is allowed to do to each object.
Trojan horse	A program that seems to do something useful, and thus gets the users to run it, but instead of or in addition to the announced purpose does something harmful.
X.509	An international standard for identity certificates and their usage, defined by ISO.
Subject	A certificate field that specifies the entity the certificate was issued to. *Also*: An active player, such as a user or a process, in a computer system.
Tag	A certificate field that apecifies the authorization given to the certificate subject.
Delegation	A certificate field that specifies whether the certificate subject is allowed to further delegate the rights given to it in the certificate.
Validity	A certificate field that indicates the conditions that must be met in order to the certificate to be valid. It is usually given as a time range.
S-expression	A LISP-like, parenthesized expression that is used to define SPKI certificates (among other things). A s-expression can contain a byte string or a set of other, simpler s-expressions.
SDSI	Simple Distributed Security Infrastructure
SPKI	Simple Public Key Infrastructure

System class	A class that is a part of the Java class library delivered with the JDK and the JRE.

1. Introduction

Security is commonly agreed to be an important property in computer systems, and more and more systems are designed and built to be secure. Security is not, however, something that one can design or implement once and then be done with. It needs continuous management. While poor design or implementation is sure to thwart the system security, poor management will definitely do the same. Thus, proper management of the system security needs to be made as easy as possible.

A part of making security management easy is getting rid of arbitrary, technical restrictions to the way it can be done. One such restriction is the requirement that the manager needs to update one or several files on every computer of the organization to change the way the organization-wide system behaves. Another example of restrictions is that security management or even parts of it cannot be assigned to the system administrator and other people within the organization as desired—or even to other organizations—without surrendering too much power over the system to everybody involved. These technical restrictions have often been created by the system security design or implementation.

Distributed management of system security means that, from an individual computer's point of view, the management can be done locally, but it can also be done somewhere else. From the system security manager's point of view, distributed management means that he or she can manage the security of a large organization from a centralized location, without having to walk to every device's console or to log in every computer that is being managed.

The goal of this study is to extend the Java 1.2 security model towards fully distributed management and deployment. The current Java security policy is defined using local configuration files in each computer and device that runs Java. We want to make these configuration files obsolete. Furthermore, we want to make the security policy dynamic, so that changes in the policy affect the access control decisions instantly. At the moment the policy is fairly static.

Another goal is to create a ground for future work on making several JVMs (Java Virtual Machines) to be able to create common distributed security domains that also allow secure channels to be created between the JVMs. This kind of distributed deployment is almost impossible without distributed security management.

In this work, we evaluate the suitability of SPKI (Simple Public Key Infrastructure) for implementing distributed security policy management features, described above, in Java 1.2. As a part of this goal, we have implemented a research prototype that builds on the earlier work by Ilari Lehti [Leh97].

1.1 Distributed Systems Security

In his Licenciate's Thesis, Nikander defines confidentiality, integrity and availability to be the basic dimensions of security [Nik97]. Data confidentiality means that only legitimate parties are able to read information. Data integrity means that only legitimate parties are able to modify information. All systems contain information that should be protected from unauthorized changes or disclosure to unwanted parts within or outside the system. System availability means the information is accessible when needed, or the system is able to perform the tasks the legitimate users ask it to do.

There are many different threats to confidentiality, integrity and availability. For example, a malicious Web applet might try to invade the naive user's privacy by reading information from the hard disk or by monitoring what the user types on the keyboard. As more and more computers are being equipped with microphones, it might even gain control of the microphone and thus find out what is being said near the computer. Even if parts of the disclosed information are not confidential alone, the aggregation of such information could be highly classified. The applet might also try to modify and delete files, or even format the hard disk. Further more, it could compromise the system availability by hogging so much memory or other resources that the computer is stalled, or even crashing the browser or the operating system. [McG97]

In the example above, access control, or restricting what the applet can do, is one of the most important means for archiving the goals of confidentiality, integrity and availability. Access control, in turn, needs authentication to know who the entity trying to do something is, authorization to know what the entity is allowed to do, and cryptography to make forging identity or authorization impossible in practice. The concepts of authorization, access control and cryptography are defined in more detail later in this chapter.

1.2 Security Policy and Access Control

All computer systems have objects or resources that must be protected from disclosure, unauthorized change, or exclusive reservation such that legitimate users cannot access the objects when needed (denial of service).

A computer system can be seen to include two types of entities. Users and processes acting on the users' behalf are called *subjects* of the computer system. The resources that may need to be protected are called *objects* of the computer system. The objects may be files, directories, peripheral devices or even processes. The subjects perform actions on the objects.

A security policy for the system defines which subjects should be allowed to access which objects and in what way. The policy can be expressed as a table called an access matrix, where the columns represent the subjects of the system, rows the objects of the system, and each intersection states the action the subject represented by the column is allowed to perform on the object represented by the row. An example security policy of a very small system with only three processes and four files might look like the one given in Figure 1.

	Process A	**Process B**	**Process C**
File 1	read, write	read	read
File 2	-	read, write, execute	-
File 3	read, execute	read, execute	-
File 4	read, write	read, write	-

Figure 1 A sample access matrix

The security policy definition itself does not help much. The policy needs to be enforced. Usually the system includes some mechanism to mediate access, so that the subjects must make a request to be able to perform an action on an object, and there is some mechanism to grant or deny the subjects' requests based on the security policy. This concept of mediator or filter is known as a reference monitor, and shown in Figure 2. It may be a separate module or the functionality may have been distributed throughout the system. In either case, the mechanisms that implement the reference monitor functionality perform the access control in the system.

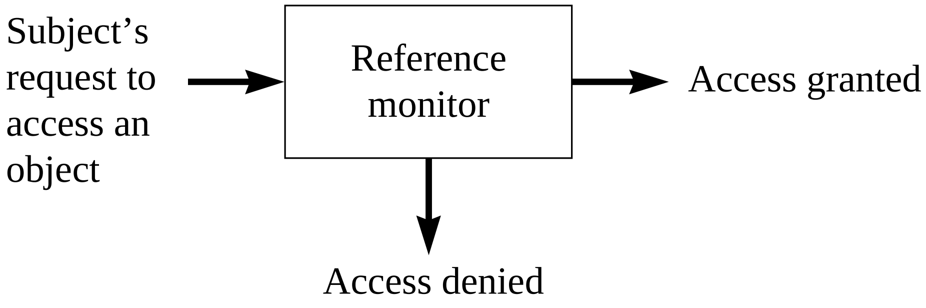


Figure 2 Reference monitor

There are many different ways to implement access control. One of the most well known mechanisms is the UNIX `rwxrwxrwx` protection bits that specify the read, write and execute access rights of a file[1] for the owner, group and others. Every time a process tries to access a file, these bits are inspected and the access is granted if the process or its group has the permission bit required.

Another well known and often used access control mechanism is called access control lists (ACLs). In this scheme every file or other protected resource maintains a list of subjects that have the right to access the file. Usually the list also specifies the type of access granted. This approach has more expressive power than the protection bits.

Capabilities are a similar approach to ACLs, but here the subjects have the list of objects they can access and not the other way around. In practice, the capabilities can be seen as separate objects or "tickets". To get a specific access to a protected resource, the subject must show it has the associated capability.

1.3 Authorization

The Webster's dictionary defines authorization as the act of authorizing, or the permission or power granted by an authority [MW94]. In a computer system, authorization is the action of giving a subject access rights to an object. Thus, defining the system security policy or a part of it is authorization.

[1] In UNIX all devices are also abstracted as files, so controlling access to files gives adequate protection.

All of the access control mechanisms mentioned above are so-called discretionary access control mechanisms (DAC). This means the users can authorize access to certain resources and thus affect the mediation performed on the system. This gives the users a great flexibility to authorize others to use their files as needed, but can also result in security problems if the users, for example, forget to protect critical resources. Usually the owner of an object is the one responsible for authorization. Often there are also some default access rights that are granted if the owner does not explicitly specify the authorization for some object.

In a mandatory access control mechanism (MAC), an administrator or a security officer does the authorization and the users cannot affect it. Although this may result in better security, it is most likely to deteriorate the system’s usability to the effect that a DAC mechanism is preferred in most environments.

1.4 Public Key Cryptosystems and Digital Signatures

Cryptosystems are numerical algorithms that convert normal data called plaintext into encrypted, unintelligible ciphertext, and possibly vice versa. A public key cryptosystem uses one piece of information, called *key*, as input to the encryption function, and another related key as an input to the decryption function. One of the keys is typically kept private and the other key is published (and therefore the name public key cryptosystem).

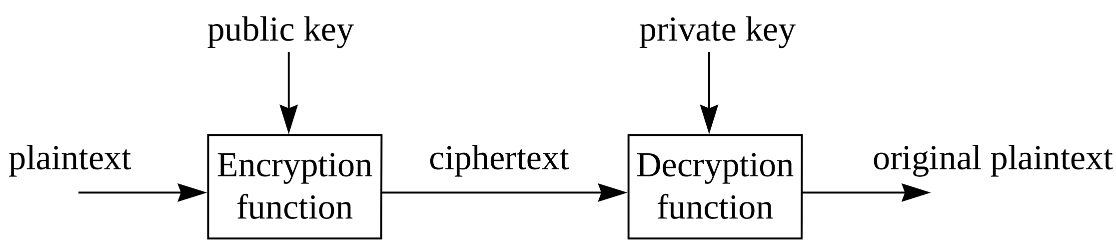


Figure 3 A public key cryptosystem

The system of two interrelated keys makes public key cryptosystems especially good for authentication purposes, as the holder of a private key K-, let us call her Alice, can encrypt a piece of data with the private key, and publish the data and the resulting ciphertext. Anyone can then use the corresponding public key K+ to verify that the information was indeed encrypted using the private key K-. Since the only person knowing the private key K- is Alice, the verifier knows that Alice indeed once encrypted the data. If real time authentication is needed, the verifier can generate some amount of random data, called a *challenge*, that the party that pretends to be Alice must encrypt. If the response contains the challenge encrypted with the private key K-, the verifier knows that the party really has access to Alice's private key, and can presume that it is Alice.

Current public key cryptosystems have one fundamental drawback: they are relatively slow. Therefore, they are usually used for encrypting and decrypting only relatively small amounts of data. If a party would like to authenticate a large amount of information, digital signatures could be used for diminishing the amount of data to be encrypted. Digital signatures are generated in two steps. First, a one-way function, called a hash, is applied to the original message. The output is a bit string of fixed length, usually a lot shorter than the original message, and its value depends on every bit of the original message so that even only slightly different messages result in completely different outputs. The hash functions are expected to be collision-free so that it is practically impossible to find two messages that would produce the same hash value. In the second step, the hash value is encrypted with the private key to produce the final signature.

The strength of a cryptosystem depends on the algorithm(s) used and the length of the key(s) [Sch94]. Cryptography is generally categorized into strong and weak cryptography. The difference between these two groups is basically that the former is believed to be unfeasible to break, while the latter can be broken if the breaker is willing to put enough effort to it. Of course, what is considered strong today may be weak in a couple of years as the technology evolves and computers become faster and cheaper, or even tomorrow if a weakness is found in the algorithm used. Throughout the rest of this Thesis we assume that all cryptography used is strong.

1.5 Capabilities and Certificates

In many systems with capability-based access control the ability to read and copy a capability makes an entity able to use the authorization granted in the capability. That is, possession of the ticket grants the authority. Consequently, the capabilities or the "tickets" must be protected from unauthorized reading. However, if obtaining a copy of the ticket would not yield to being able to use the ticket, this kind of protections would be unnecessary. Authorization certificates have this desirable characteristic: the authorization is given to a private key, which is only known to the authorized entity. The capability cannot be used without the key. Now the key must be held secret, but the certificate can be published and freely copied. This considerably diminishes the amount of data to be held secret. Using strong cryptography also makes creating false “tickets” very hard.

The strict requirement that capabilities may not be read by unauthorized subjects is hard to satisfy in distributed systems. Using authorization certificates as capabilities makes designing secure, capability-based distributed systems much easier. Now the certificates can be transmitted from one part of the system to another with almost no security measures, and the private keys can be stored locally where they are easier to protect. The private keys might even reside in a tamperproof part of the hardware and never exit it.

1.6 Mobile Code and Trojan Horses

Controlling user access to different resources has been studied for a long time and is fairly well understood. Unfortunately, this kind of access control alone cannot solve the problem of unauthorized access. One of the biggest sources of security problems today are programs called Trojan horses. A Trojan horse is a program that seems to do something useful, but instead or in addition does something harmful [Amo94]. Often this harmful thing helps a non-legitimate user to bypass the access control and thus gain access to the system or to protected resources. Traditional access control cannot help against this problem because the legitimate users run the Trojan horse program using their own access rights and denying these rights would prevent them from doing their job.

The fact that traditionally the system administrator has controlled the installation of most software has kept this problem from developing into a total catastrophe. However, this is rapidly changing with the new era of mobile code: anyone can download a program from the World Wide Web and run it on his computer, and people are doing just that. We now need to control the individual programs’ access to different resources as well as the users’ access. This need is generally known as the confinement problem [Lam73].

Access control in program level has an additional benefit: it can restrict damage caused by innocent programming errors as well as restricting what maliciously written programs can do. Thus, the principle of least privilege should apply also to programs and not only to users.

1.7 Organization of this Thesis

The rest of this Thesis is organized as follows.

Chapter 2 introduces the Java 1.2 Security Architecture. It explains how the access control is done and what are the benefits and shortcomings of this approach.

Chapter 3 presents a model called Simple Public Key Infrastructure (SPKI) and shows how it can be used to provide an authorization and access control framework.

Chapter 4 presents the design of the new access control system for Java 1.2.

Chapter 5 describes the prototype implementation. The use of design patterns is also discussed.

Chapter 6 gives an evaluation of the design and its implementation. It discusses the applicability of the approach in terms of security and performance. In addition, it suggests further improvements to the model.

Chapter 7 presents our conclusions and summarizes the work.

1.8 Notations Used

We use the following notations in this Thesis.

Class names are written with capital letter initials, for example: ClassName.

File names are written with Courier font, for example: `/tmp/myfile`.

Certificate and file format definitions are written with Helvetica font, for example: This is a definition.

Code examples are written with Courier font: `while (i < 10) i++;` .

The format of SPKI objects is a simplified form of S-expressions. An S-expression is a list enclosed in matching “(“ and “)”. In this Thesis, SPKI objects are defined in a familiar extension of BNF – with “|” meaning logical OR, “*” meaning closure (0 or more occurrences), “?” meaning optional (0 or 1 occurrence) and “+” meaning non-empty closure (1 or more occurrences). A quoted string represents those characters.

2. Java 1.2 Security Architecture

Java is a relatively new, object oriented programming language, but it is also more. It is an interpreted language: programs written in Java are compiled into an intermediate, platform-neutral format called *byte code*. This byte code can be executed in any system that has a Java interpreter. Today, the interpreters can be found in all major Web browsers, all major operating systems, and even PDAs and other embedded devices. Thus, Java can be used to create programs that can be run without modifications on all platforms, and this property makes it very popular for creating applications and Web applets. The popularity of Web applets, in turn, puts more weight on the security issues. This is mainly because most of the users browsing the Web do not even realize that while looking at a nice Web page, they are actually running a downloaded program on *their machine*, and that the program might try to harm the user. Furthermore, the vast majority of users do not want to think about the security, even if they realize they are running applets: they just want to surf—safely.

The Java Runtime Environment (JRE) seems to be the first widely accepted platform for running executable content, or mobile code. This is mostly because it has addressed the security issues of executing unknown code, i.e., the confinement problem, from the very beginning. The purpose of Java security is to protect the user form attacks that invade the user's privacy, modify the system, or hog resources so that legitimate users cannot use the machine. This is done by limiting what untrusted programs can do. The task is not simple, because nobody wants to run a program that is not allowed to do anything. Protecting the user means balancing between the risks and the benefits [McG97].

The JRE comprises two major components: the Java Virtual Machine (JVM) that interprets the byte code, and an execution environment that runs on the JVM and includes the Java class library, a set of basic classes that provide useful functionality [Arn96]. In addition to just interpreting the byte code, the JVM also performs a set of checks on the code before and while it is executed. The most important of these checks is type safety, ensuring that the data pretending to be of a specific type, such as an integer, really is what it pretends to be, and gets handled as that.

Java classes are divided in two groups. *System classes* are Java classes that belong to the standard Java library or to an extension of it. These classes are provided by Sun and other Java platform vendors. They form an important part of the underlying platform and are trusted to perform their job correctly and not to include any Trojan horses. The other group is application classes. These classes are a part of some applet or application and their origin may or may not be known. They are generally treated as untrusted, unless the security policy states otherwise.

2.1 Access Control in Java

Java access control traditionally starts from the assumption that unknown mobile code is like an unknown animal. It is potentially dangerous and must be kept in a cage until it has been surveyed and is known to be harmless. The cage keeps the animal from doing damage to its environment. In Java, the cage is the so-called sandbox: the classes inside the sandbox can execute harmless operations such as calculations and basic interaction with the user, but they cannot access any critical resources on the system, such as arbitrary files or network connections. In particular, they cannot modify the system in any way, and their means of collecting information about the user and sending them off to an outside party have been strictly restricted.

In Java 1.0, the distinction between known and unknown code was made based on where the code was loaded from. Code from local disk was treated as trusted and could do anything. Code loaded from the network was seen as unknown and was placed in the sandbox. However, this division turned out to be too restrictive and counterproductive, because it prevented users from loading straight from the network mobile code that needed to do something not allowed in the sandbox, even if the user knew that the code was all right. Instead, the user had to install the classes locally before he could run them.

Java 1.1 introduced the notion of signed classes. Now mobile code loaded from the network could be trusted if it was signed with the key of a trusted verifier. For example, administrators could now review code intended for use in their organizations and if the review disclosed nothing alarming they would sign the classes with their key. Now they could make the signed classes available on the network. All clients could then verify that the signature matches the trusted key of their administrator and use the code without having to restrict it to the sandbox. All unsigned applets were still considered untrusted and executed in the sandbox, equally restricted as before. Figure 4 shows the Java 1.1 security architecture.

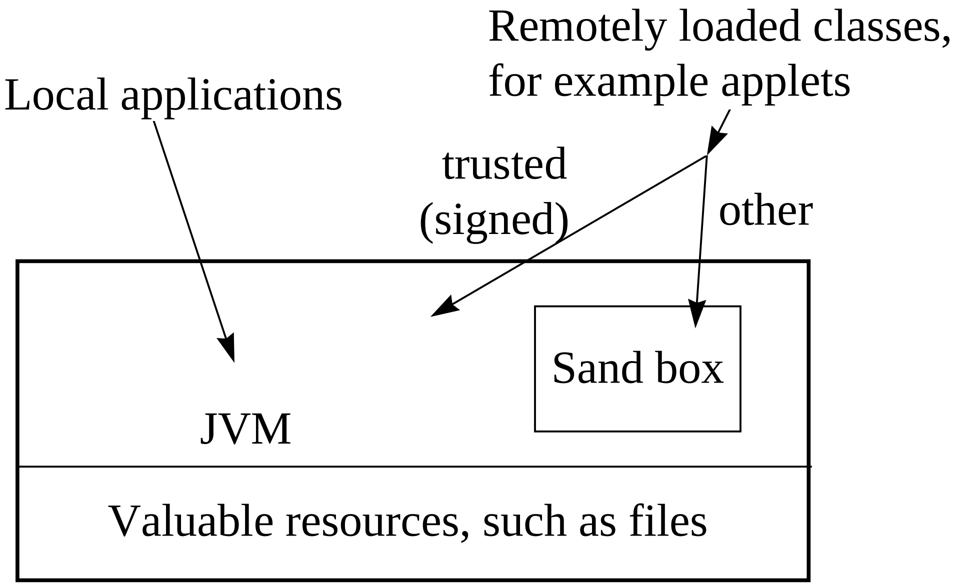


Figure 4 Java 1.1 Security Model

Unfortunately, this model was still too black-and-white. The ability to bypass access control if the code was stored on local disk created a security threat. In addition, reviewing all the source code for every possible threat was tedious and difficult to delegate outside the organization. The all-or-nothing permissions just were not enough. If the program could be given only those rights it needed to perform its job, verifying that it uses those rights correctly is much easier than verifying everything. Further more, it may not even be necessary if the organization does not see the resources affected as critical. Thus, with Java 1.2, Sun introduced a fine-grained, configurable security policy that mandates what access rights are granted to which programs [Gong98]. It applies to all classes except the system classes. This architecture is depicted in Figure 5.

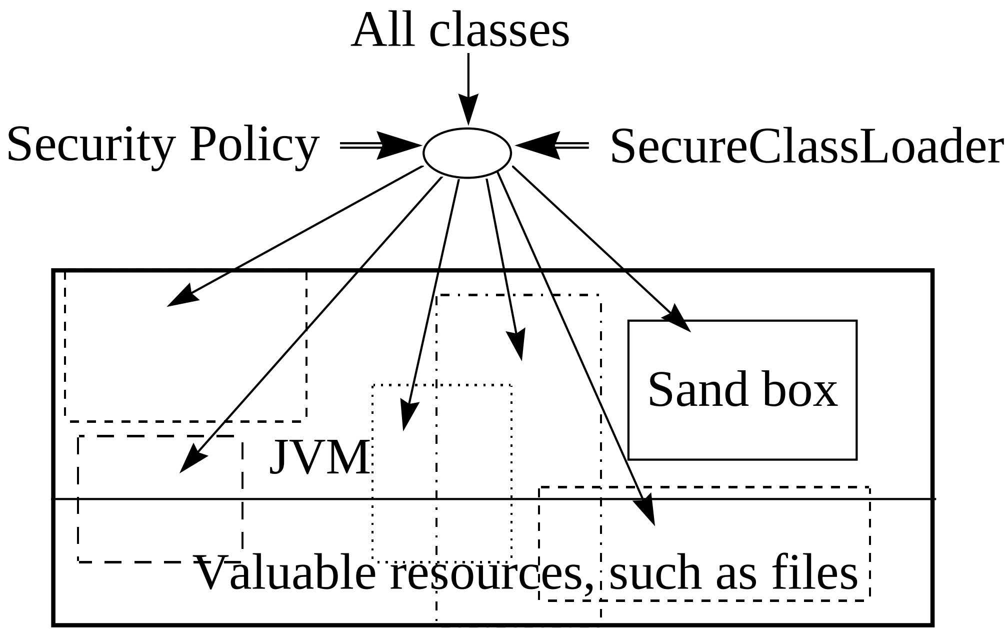


Figure 5 Java 1.2 Security Model

The rest of this chapter describes the Java 1.2 security architecture in more detail and introduces its different parts. A list of the parts is given in Table 1.

Class or classes	**The role of the class or classes**
Permission and its subclasses	The instances of these classes represent different “tickets” or access rights, i.e., permissions, in the JVM.
ProtectionDomain	The “glue” that connects the Permission objects to the classes.
SecureClassLoader and its subclasses	These classes take care of placing the classes into the protection domains.
Policy and its subclasses	These classes decide what Permission instances each class gets.
AccessController	This class checks that requesters of protected resources have the corresponding “tickets”.

Cryptographic keys, signatures and certificates	These classes and interfaces are used, among other things, to digitally sign classes and thus express that the signer trusts them. This trust, in turn, can be used by an administrator to define a security policy according to which the Policy makes its decisions.

Table 1 The central parts of the Java access control architecture

2.2 Permissions

Java 1.2 introduces a new type of classes, called permissions, that are used inside the Java Virtual Machine to represent access rights to protected resources. Each protected resource in the system has a corresponding Permission object. The Permission object can be seen as a capability or a “ticket” that grants access to the corresponding resource. There are typically many instances of a given Permission, possessed by and thus granting access to different classes.

Permissions are divided into several subtypes that extend the Permission class. Each resource type or category, such as files or network connections, has its own Permission subclass. Inside the category, different instances of the class correspond to different instances of the resource. Examples of the different Permission classes and their instances are given in Table 2. In addition, programmers may provide their own Permission subclasses if they create protected resources of their own.

Permission	Examples
java.awt.AWTPermission	p = new AWTPermission(“systemClipboard”); p = new AWTPermission(“eventQueue”);
java.io.FilePermission	p = new FilePermission(“myfile”, “read,write”); p = new FilePermission(“/usr/bin/*”, “execute”); p = new FilePermission(“<<ALL FILES>>”, “read”);
java.lang.RuntimePermission	p = new RuntimePermission(“createClassLoader”); p = new RuntimePermission(“Class.getProtectionDomain”);
java.net.NetPermission	p = new NetPermission(“Authenticator. requestPasswordAuthentication”);
java.net.SocketPermission	p = new SocketPermission(“www.company.com”, “accept”); p = new SocketPermission(“*.domain.com:80”, “connect”); p = new SocketPermission(“1.2.3.4:8000-9000”,”listen”);
java.security.AllPermission	p = new AllPermission();
java.util.PropertyPermission	p = new PropertyPermission(“user.*”, “read,write”);

Table 2 Examples of Java permissions

The Permission subtypes can be categorized in three groups. The rights granted by the permissions in group one are uniquely specified by the permission type or class as such and so the permissions on this group do not need any arguments. An example of this group is the AllPermission. The second group consists of permission classes whose type only tells which generic part of the system (such as the user interface) the right is associated with, but does not specify the exact action granted. Consequently, these permissions take as an argument a name that makes the permission fully qualified. An example of this group is the RuntimePermission. The third group takes two string arguments: a target that denotes the object and an action that specifies the type of access. For example, the FilePermission belongs to this group.

Some permissions are more generic than others. A single, general permission object may grant access to more than one instance of the controlled resource. For example, an instance of the FilePermission class may also grant access to all files in the system or a directory, instead of just a single file. A generic permission instance *implies* a number of more restricted permissions. Thus, for example, FilePermission("/tmp/*", "read,write") implies FilePermission("/tmp/foo.txt", "read"). Instances of the class AllPermission imply all other permissions.

There are also new classes for holding groups of Permission objects. PermissionCollection is an abstract superclass for these containers. Some subclasses of PermissionCollection can only hold permissions of one type, for example FilePermissions. Other subclasses, such as the Permissions class, can be used to hold any number of different types of Permission objects. In practice, the Permissions object holds a PermissionCollection object for each permission type that is stored in the collection.

2.3 Protection Domains

Just as in any capability-based access control system, the Java classes must be prevented from creating permissions for themselves and thus gaining unauthorized access. This is the task of ProtectionDomains [GS98].

Each class belongs to one and only one ProtectionDomain. Each ProtectionDomain has a PermissionCollection object that holds the permissions of that domain. Only these permissions can be used to gain access to resources. The classes cannot change their ProtectionDomain nor the PermissionCollection of the domain. Thus, the classes are free to create any Permission objects they like, but they cannot affect the access control decisions and gain unauthorized access. Figure 6 illustrates the ProtectionDomain concept.

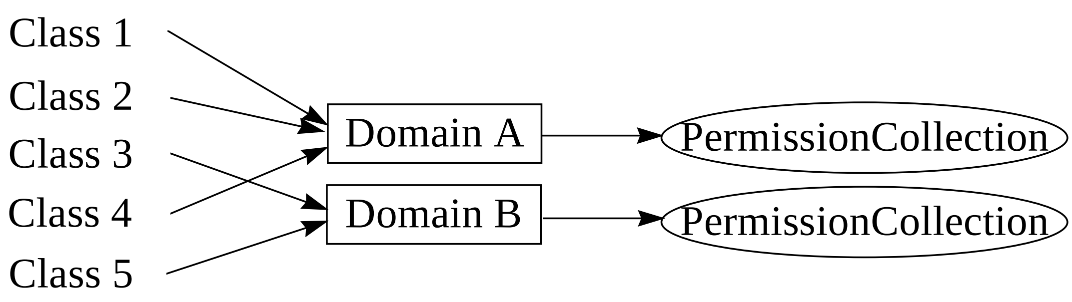


Figure 6 Protection domains

In the current Java 1.2 implementation, a protection domain of a class is uniquely identified by the CodeSource of the class. A CodeSource consists of the codebase or URL that the class was loaded from, and a set of cryptographic certificates that indicate the signatures the class has. Thus, if classes A and B have both been signed by person P, and A was loaded from URL1 and B from URL2, then A's CodeSource is (P's certificate, URL1) and B's CodeSource is (P's certificate, URL2). The classes are placed in the protection domains corresponding to their CodeSources. If a class is not signed, or if the signature cannot be verified, the class is placed into a protection domain that has an empty set of certificates.

All classes in the same protection domain get the same Permission objects. However, classes with identical permissions may belong to different protection domains, because many protection domains may happen to have a similar set of permissions assigned to them by the current security policy.

2.3.1 Placing classes in domains

Before a class can be run, its code must be loaded into memory and the class must be linked and initialized. Class loading refers to the process of finding the binary form of a class or interface type with a particular name, typically by retrieving a binary representation from a class file, and constructing, from that binary form, a Class object to represent the class or interface [Arn96].

Unlike many other programming languages, Java enables dynamic class loading, i.e., loading a class may be delayed until it is used for the first time. This way, classes that are never used are never loaded, even if the source code includes a reference to them. Dynamic class loading makes also late binding possible. New classes can be created and used on the fly. A class does not have to even exist at the time an application is launched, and the application is still able to load and use it later when it does exist.

The loading process is implemented by the class ClassLoader and its subclasses. Different subclasses of ClassLoader may implement different loading policies. A class loader may, for example, cache binary representations of classes and interfaces, or load a group of related classes together [JAVA98].

The SecureClassLoader is a new class loader that directly or indirectly loads all classes except the system classes. Indirect loading means that the SecureClassLoader is used to load a class loader, which in turn is used to load the class. The SecureClassLoader takes care of placing the loaded classes into protection domains. This is why all new class loaders should inherit the SecureClassLoader.

2.4 AccessController

The AccessController is the Java 1.2 incarnation of the reference monitor concept. That is, when a thread requests access to a protected resource such as a file, the AccessController object is asked whether the access is granted or not. To determine this, the AccessController checks the execution context to see if the caller has the Permission object corresponding to the resource. For example, if a class tries to read the file `/home/jhp/myfile`, its protection domain must have the FilePermission("/home/jhp/myfile", "read"), or some other permission that implies this permission.

Asking for access that requires some permission may be made by a method that was called by another class. The calling class may belong to a different protection domain. Since it is important that a class does not bypass the access control simply by calling another class that has more permissions, the AccessController also checks all the previous classes in the call chain. The general algorithm is that if class A calls class B, which in turn calls class C and so on, and finally class M tries to read a file, then the AccessController checks each class from M to A to see if they all have the required permission. If some class in the call chain does not have the permission, Access-Controller throws an exception. Otherwise it returns quietly, implicating that the request has been accepted.

There is one irregularity to the general access control algorithm. A class may ask the AccessController to mark it as “privileged” while performing a task. This marking creates an artificial bottom to the call stack. When the AccessController reaches a class marked privileged, it checks whether this class has the permission in question and then stops. The preceding callers are not checked. Figure 7 illustrates the access checking algorithm and the effect of marking a class as privileged. In case A all the relevant classes have the required permission, and the request is granted. In case B, the Class 2 does not have the permission and AccessControlException is thrown. In case C the Class 2 still does not have the permission, but since the Class 1 is marked privileged, the AccessController stops the check after checking Class 1 and the access is granted.

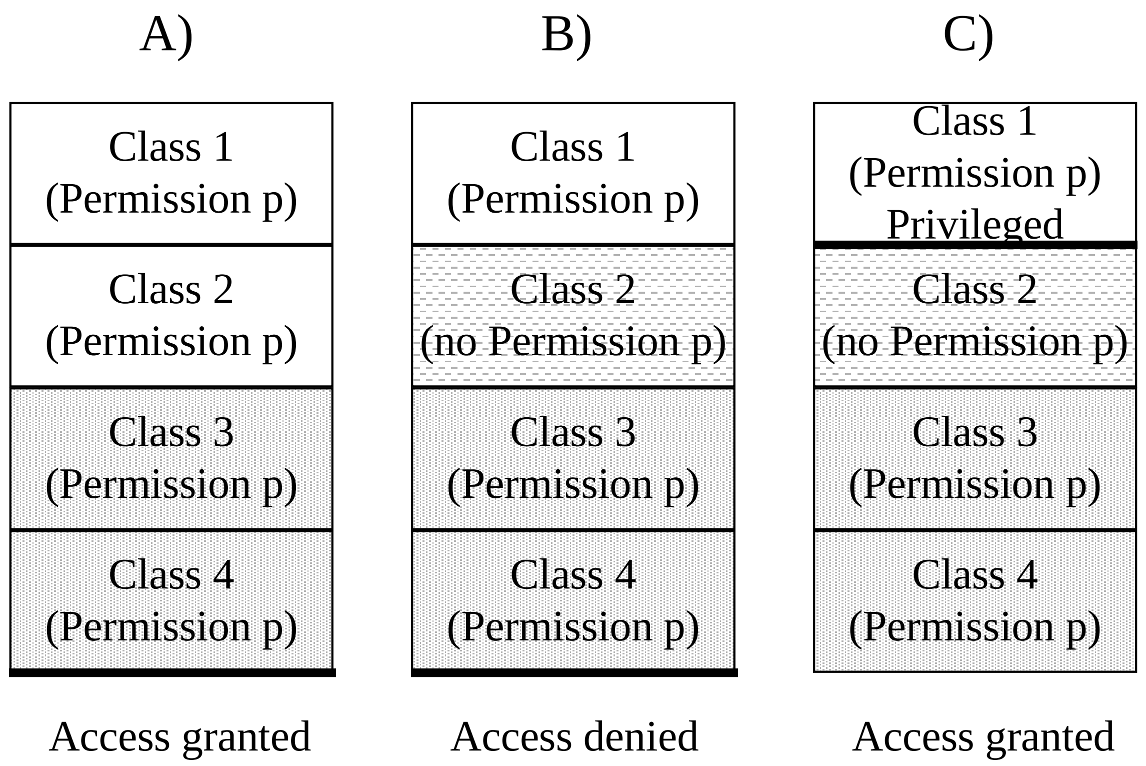


Figure 7 Examples of call stacks and corresponding access control decisions. The different colors represent different protection domains. The fat line shows where the AccessController thinks the stack bottom is.

To further ensure that the access control cannot be bypassed, any thread inherits its parent’s access control context. The AccessControlContext object contains all information relevant to making access control decisions. Thus, finally, the access control algorithm can be stated in pseudo code as follows [Gong98]:

```
// Check the current call stack

i = m;
while (i > 0) {
    if (caller i’s domain does not have the permission)
        throw AccessControlException
    else if (caller i is marked as privileged)
        return;
    i = i - 1;
}

// Check inherited AccessControlContext

inheritedContext.checkPermission(permission);
```

2.5 Security Policy

A security policy defines the rules that mandate which actions the actors in the system are allowed or disallowed to do [Amo94]. The Java security policy, implemented as a subclass of the class Policy, defines what permissions each protection domain gets. There is a clear separation of duties between the AccessController and a Policy object: the Policy defines the rules and the AccessController enforces them. In other words, the Policy gives you the tickets and AccessController checks them at the gate. This means we can change the policy according to which we distribute the permissions, without having to touch the AccessController.

A security policy can be static or dynamic. A static security policy is fixed: the permissions of a class cannot change once it is loaded into the JRE. However, the permissions can be different the next time the class is loaded, during another run of the JRE. Having a static security policy has some performance advantages. However, if the session is long the circumstances may change so much that a change in the security policy is needed. Furthermore, even if the sessions are short, a change in the policy may be so important that it must take effect immediately. Thus, dynamic security policy that can be changed “on the fly” is the preferred solution because it provides better security. The security policy in current Java 1.2 implementation is rather static. That is, it does have a “refresh” method, but that method must be called explicitly and it only affects the permissions granted after the refresh. The domains that got their permissions before the refresh still have the same permissions after it.

The class Policy is an abstract class. The actual implementation, which can be changed, defines how the security policy is managed. The default policy implementation of JDK 1.2 uses a set of configuration files to define the security policy. The files have two types of entries: an entry that specifies the location of the KeyStore where keys and certificates are stored, and entries that grant permissions. The entries that grant permissions have the following format:

```
grant [SignedBy "signer_names"] [, CodeBase "URL"] {
      permission permission_class_name [ "target_name" ]
             [, "action"] [, SignedBy "signer_names"];
      permission ...
   };
```

There is one configuration file for defining a system-wide security policy. Each user may additionally have their own policy file. All the definitions are additive, so permissions can only be granted, not taken away. If the policy files do not exist or their format is incorrect, the classes end up in the sandbox.

The policy configuration file is clearly a kind of an ACL. As all ACLs, it has the disadvantage that it must be maintained locally, i.e. the access right management cannot be easily distributed. If we want to make this management easier to distribute we must replace the configuration files with a capability-based policy definition.

2.6 Keys, Certificates and Certificate Management

As mentioned above, the classes are placed in the protection domains according to where they have been loaded from, and what keys they have been signed with. To be able to sign classes and verify the resulting signatures, Java includes a basic set of cryptographic functionality. The concepts of cryptographic keys, digital signatures and certificates are a central part of this functionality. The keys are used as input to the signature functions, and the certificates are used for telling the verifier the key that can be used to verify the signature, and whom the key belongs to.

The Java Cryptography Architecture (JCA) was designed to support implementation independence and interoperability. This is archieved with the so-called “provider” architecture. For each cryptographic service, such as a message digest or a signature, there is an abstract class or interface that defines the general behavior of the service.

The actual implementation of the functionality is provided by a "provider", a package or a set of packages that implement one or several cryptographic services. The user may have several providers installed, and new providers can be installed dynamically. When the user needs a particular service, she can request it without specifying the implementation and get the default provider, or she can request the service from a specific provider's implementation.

In the international version of Java 1.2 there are several general interfaces related to cryptographic keys. Key is the top-level interface for all keys, PublicKey is the interface for public Keys and PrivateKey for corresponding private keys. The KeyPair class is used to hold a (PublicKey, PrivateKey) pair. Classes implementing the KeyPairGenerator interface are used to generate new key pairs. The KeyFactory is an interface for classes that take care of converting keys to raw key material, called KeySpec, and vice versa. There are also more specific interfaces for RSA and DSA keys and their handling. The default provider that comes with the JDK and implements these services is called "SUN".

The java.security.cert.Certificate is an abstract superclass for identity certificates. It has several methods that are supposed to be common to all identity certificates: equals, getEncoded, getPublicKey, getType, hashValue, toString and two different verify methods. Although it was designed to be a superclass for identity certificates, with little imagination it is generalizable to authorization certificates as well.

Many uses for certificates require maintaining a list of certificates that have been revoked before their validity period is over. They are called Certificate Revocation Lists (CRL). JDK has an abstract superclass for CRLs, too.

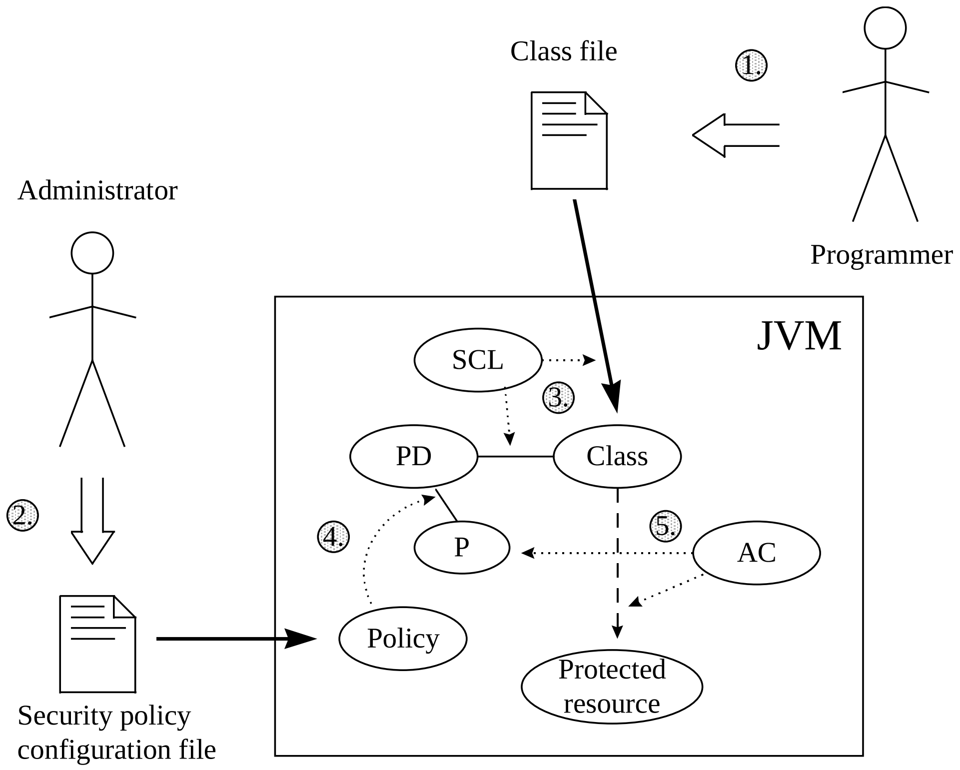
The Certificate class' only subclass in JDK 1.2 is the X.509Certificate class. It is also an abstract class, and serves as the API for X.509 certificate implementations. In JDK, X.509 certificates are used in creating and verifying the signatures of the classes. The related classes include the X509CRL class and the X509Extension interface.

Whereas all certificates have the getEncoded() method that returns the transfer representation of the certificate, they do not have a method that would create a new certificate from this encoded form. This is the job of a CertificateFactory object. Each certificate type has its own CertificateFactory subclass.

Key and certificate storage in Java is handled by a class called KeyStore. The KeyStore is an in-memory collection of keys and the corresponding certificates. Its subclasses may also implement permanent storage, for example on the local hard disk. The default implementation of the KeyStore functionality in JDK 1.2 is called JavaKeyStore. It stores the keys and certificates in a file. The file is encrypted with a pass phrase to protect the private keys. JDK also provides some tools for manipulating the file. However, anyone can provide his or her own implementation of the KeyStore functionality.

2.7 Summary of the Java Security Architecture

The Java Security Architecture can be summarized with the simplified sequence of events during the lifetime of a class. First, a programmer creates a class file, signs the file with his private key K and attaches his certificate, and makes the whole bunch available to the users (1.). Meanwhile, the system administrator decides that the programmer is trustworthy to write code that can access certain resources on his system, and adds the corresponding rule to the security policy configuration file (2.). When a user decides to run the code, the SecureClassLoader loads the class file into a class object, checks the signature, and places the class into a protection domain (3.). When the protection domain was created it got the permissions associated with it from the Policy object (4.). Now the class is ready to run. Later, any time when the class tries to access a protected resource, the AccessController checks whether the class has the permissions needed for the access (5.).



SCL = SecureClassLoader
AC = AccessController

PD = ProtectionDomain
P = Permissions

Figure 8 Summary of the Java Security Architecture

3. Authorization with SPKI

3.1 Certificate Types

The word “certificate” was first used in 1978 by Loren Kohnfelder to refer to a signed record holding a name and a public key [SPKI98b]. Since then, cryptographically binding a globally unique name to a key—and thus certifying an identity—has long been considered the only purpose of certificates. In fact, many security professionals still use the words “certificate” and “identity certificate” as synonyms. However, another type of certificates called authorization certificates has been proposed, first independently in the SDSI [SDSI96] and PolicyMaker [Bla96] prototype systems and the SPKI initiative[SPKI98a].

If we combine these two views, we get a new definition for the word “certificate”. A certificate essentially is a digitally signed record stating something about the entity that the certificate was issued to. To be more precise, certificates may certify identity, they may state some other information about the keyholder, or they may certify authorization. The implicit meaning of any certificate is “I, the issuer of this certificate, attest to the information stated here”.

Certificates can be categorized in two general groups according to the information they carry. The first group is used to transmit some identifying information about the certificate subject, or in other words, what the issuer knows about the subject. The second group is used to express authorization and delegation, or trust in short. As trust is not knowledge but belief, the two groups have distinct characteristics.

3.1.1 Identity certificates

As stated above, identity certificates bind a name to a key. There are several different standards for identity certificates. These include The X.509, PGP, SDSI and SPKI. The SDSI system and the SPKI proposed standard have a similar approach and are being combined to a single standard.

The X.509 is the oldest of the standards mentioned here. It assumes that each user has a globally unique ID of the X.501 Name format. X.509 certificates bind these IDs to cryptographic keys [X.509]. X.509 certificates are issued to users by official Certificate Authorities.

In the PGP system, each user generates a key pair that is associated to their globally unique ID, usually of the format (Full Name, EmailAddress) [PGP95]. The keys are stored in key records. PGP users can sign each other's key records, if they are sure the records have not been tampered with since they have been created. When encountering a new key, the user can now use the signatures to deduce if she can trust the key to be valid or not.

The SPKI/SDSI identity certificates differ from the more traditional examples in that they use local, linked name spaces instead of a global one. The motivation for such a choice of name space is that as the number of entities with an identity grows larger and larger, the minimum length of the global name also grows. When every human, computer, organization and piece of software need their own global name, the names will become so long that humans are likely to have trouble remembering them (many humans have trouble remembering the names of their acquaintances, and those names are short). When humans are not sure, they tend to guess. This may introduce serious security flaws in systems that depend on global names.

Global names are often based on the entity's common name and organization or home place in order to be easier to remember. However, this creates the problem that a person may change her name but her identity should not change because of that. Similarly, the person may move or change the job but this should not force the person to change her identity.

In SPKI, the identity certificates are mainly used to facilitate the policy management by humans since it is difficult for humans to remember long, arbitrary bit strings such as cryptographic keys. The SPKI identity certificates do not have delegation and tag fields (see Section 3.2). Semantically, their delegation is always "true" and the authorization is always "*", so that the name certificate passes to the key everything that is granted to the name. If there is more than one name certificate for a given name, and these certificates have different subjects, then that name is a group. More specifically, all name certificates define groups, many of which have only one member.

SPKI certificates that state something about the keyholder are called donation certificates. They are actually not identity certificates, since they may not contain a name at all. They do not look like identity certificates, either. A donation certificate might certify that the keyholder of key X is called Jonna Partanen, lives in Espoo and is a female, or that the keyholder of key Y has paid his taxes, or anything else. Donation certificates do not transfer authorization and do not have a delegation field. An example of a donation certificate is given in Appendix B.

3.1.2 Authorization certificates

When used in an access control situation, the identity certificates create an artificial gap between the information that is certified, i.e. the identity, and the real question, i.e. "what is this entity allowed to do?" [Bla96]. This is also the case in JDK 1.2: the classes are signed and the certificates tell the identities of the signers, but the security manager is left to guess two important things in order to be able to define the security policy, namely:

- What permissions does this piece of software need in order to perform its tasks? Or, why was the piece of software signed in the first place?
- Can we trust it to perform those tasks correctly and not to misuse its permissions?

To overcome these problems, we need authorization certificates. Just as identity certificates bind names to keys, and thus certify identity, authorization certificates bind capabilities to keys, and thus certify authorization.

Different proposals for authorization certificates include SDSI [SDSI96], PolicyMaker [Bla96] and SPKI [SPKI98b]. SPKI has adopted some of the SDSI and PolicyMaker ideas, and is being standardized by the IETF. Authorization certificates have issuer, subject and validity information just as identity certificates. The main differences are the possible authorization and delegation fields. The authorization field specifies what rights are delegated to the subject of the certificate. The delegation field, if the certificate type has one, specifies whether the subject has the right to further delegate the rights given in the authorization field of the certificate.

SPKI authorization certificates are a fairly good example of authorization certificates in general. The five security related fields of a SPKI authorization certificate form a five-tuple <I, S, D, A, V> that is often used to briefly describe the contents of a certificate. In short, the meaning of a certificate defined by such a five-tuple can be stated as follows: The issuer I of the certificate authorizes the subject S to do A whenever the conditions specified in V are met. If delegation D is true, the subject S is also allowed to further delegate the right A. Of course, the issuer has to have the right to authorize the right A in the first place by controlling the resource in question or some other means, in order to the subject really be able to use the authorization.

Let us consider an example. Alice has an account X in the bank B. The bank wants to give Alice on-line access to her account. Consequently, the bank creates the SPKI certificate <K_B, K_A, Yes, Access to the account X, Always>, and gives it to Alice. Alice can now use the certificate to access her account, but to do so, she has to prove that she holds the private key corresponding to the public key K_A. Since the delegation bit of the certificate is set, Alice can also give her brother Carl the right to access her account to pay her bills while she is on vacation. She does not want Carl to be able to delegate the right to anyone else. To do this, Alice produces Carl the certificate <K_A, K_C, No, Access to account X for paying bills, During the vacation>. Carl, too, has to show that he controls the private key corresponding to the public key K_C before the bank will accept the certificates and let him pay the bills.

3.2 SPKI Certificates

SPKI certificates are defined with s-expressions. S-expressions are LISP-like parenthesized expressions with the limitations that empty lists are not allowed and the first element in any S-expression must be a string, called the "type" of the expression.

The body of a SPKI authorization certificate is defined to have the following fields:

<cert>:: "(" "cert" <version>? <cert-display>? <issuer> <issuer-loc>? <subject> <subject-loc>? <deleg>? <tag> <valid>? <comment>? ")" ;

The most important fields, also included in the five-tuples that describe the certificate contents, are:

- <issuer>, **I —** the principal that issued the certificate. SPKI principals are keys, expressed as a public key or a hash of the public key.

- <subject>, **S** — the subject of the certificate, i.e., the entity that the certificate was issued to, typically a public key, a hash of a public key, a name, or a hash of an object. It may, however, also be a threshold or the flesh and blood (or silicon and metal) holder of a key.

- <deleg>, **D** — the delegation field, if present, indicates that the subject can delegate the right A to someone else.

- <tag>, **A** — the authorization (also called tag) specifies the actual rights given in the certificate. The format is user-defined, i.e., every application that uses SPKI certificates is responsible of defining what kind of authorization fields it understands.

- <valid>, **V** — gives the conditions for the validity of the certificate. Typically, this field consists of "not-before" and "not-after" dates, but it may also be a one-time validity or some other on-line test. The validity field is also optional, and if missing, is interpreted to mean "always".

Table 3 gives the purposes of the other, less important fields. They are all optional.

Field	**Purpose**
version	The version number of the certificate format. A missing version field is interpreted to mean "0". A certificate with an unrecognized version number should be ignored.
cert-display	Gives a display hint for the entire certificate. It can be used to aid the user interface in displaying the certificate.
issuer-loc	Also called issuer-info. If present, it provides URLs of the location(s) of the certificate(s) by which the issuer derives the authority to pass along the authorization in the present <cert>. The SPKI definition expects the prover (the calling client) to track down such other certificates and provide them to the verifier (the called server), and the issuer-info is allowed in the certificate to simplify that process for the prover.
subject-loc	Also called subject-info. It provides the URL of the location of

	information about the subject. For example, if the subject is a hash of a key, this might provide the location of the key being hashed. If the subject is a SDSI name, it might give the location of a SDSI name certificate server.
comment	Natural language message to the possible human reader of the certificate. Ignored by the software that handles the certificates.

Table 3 The certificate fields that are not security-critical

The body is not the only part of the whole certificate. In fact, the certificate body without a signature is useless and can not be used to gain access, since without the signature, anyone could forge such a “certificate”. The <signature> object is defined as follows:

<signature>:: "(" "signature" <hash> <principal> <sig-val> ")" ;

The <hash> contains a hash value computed of the object being signed, <principal> contains the signer’s public key or a hash of it, and the <sig-val> includes the actual signature bytes.

Complete certificates are defined as a sequence that has at least the <cert> and <signature> parts. A signature object is typically used for a certificate body and follows that <cert> object in a <sequence>, but this is not its only use. One can also sign objects other than certificate bodies; for example, one can form the signature of a file.

The other possible parts in a <sequence> are public keys, CRLs, delta-CRLs, revalidations and opcodes. The revalidation is used to define a new validity period for a certificate or to provide the result of an on-line test. At present the only opcode defined is “hash”, meaning that the previous item in the sequence is to be hashed by the given algorithm and saved, indexed by that hash value. Presumably, that item (certificate body or public key, for example) is referred to by the hash value in some subsequent object. [SPKI98a]

3.3 Certification Loops

The X.509 framework postulates that everyone will get a certificate from an official Certification Authority (CA). The CAs are supposed to be organized into a global tree so that X.509 certificates form chains from the root node through intermediate nodes to the leafs in the hierarchy [X.509]. Unfortunately, this approach may not reflect the real world trust relationships. In order to trust in a certificate, the user has to trust in all nodes of the hierarchy. Most of the current root CAs are American companies such as VeriSign and Netscape. A user may not need to be very paranoid in order to not trust any one of these CAs, especially since their certification practice statements generally deny all liability. However, since the users cannot easily become their own CAs, they are forced to use certificates they do not really trust.

The SPKI incentive has another approach. It proposes that the only root of a certificate chain should be the source of trust, i.e., the Verifier [SPKI98b]. When used for authorization, the verifier is the owner of the resource to be protected, and thus has the most interest in choosing to whom to create the certificates in the first place. In the case of identification, the entity trying to identify another should be the one specifying whom it accepts as an introducer, as is the case in PGP. Thus, the issuer of the first certificate in any valid certificate chain is always the one who has the most to loose.

The fact that there is a certificate chain delegating some rights to a subject is by no means sufficient for making a positive access control decision. The subject must also authenticate itself, i.e., prove that it indeed is the subject mentioned in the last certificate of the chain[2]. If the subject of the last certificate is a public key, the authentication is done with a challenge-response protocol that assures the verifier that the other entity really controls the corresponding private key. If the subject is a hash of an object, the verifier performs the authentication by comparing the hash in the certificate with the hash of the object trying to use the certificate.

The on-line authentication closes the certificate chain into a loop, starting from and ending to the verifier. We illustrate this concept of certificate loops with a typical

[2] Even this may not be enough. A local security policy is free to impose further restrictions, e.g. like PolicyMaker.

transaction scenario described by Lehti and Nikander [LN98]. The scenario takes place when a user wants to access a service, such as a banking service or a company's extranet, for example. First, the user wants to authenticate the service in order to know he is not interacting with a bogus service. Thus, the user has authorized (or now authorizes) a Certificate Authority that the user trusts, potentially because the user has a contract with this CA, to identify entities. This CA may in turn delegate identifying services to another CA and may or may not allow further delegation. Finally, a CA who has the right to identify services writes a donation certificate to the service, specifying its public key and name, location, contact information or any other information the user needs to be sure of the service's identity. The user's software verifies that the certificates form a valid chain. The chain is closed to a loop when the user contacts the service and the service proves with an on-line authentication protocol that it possesses the private key corresponding to the public key in the certificate. The user is now assured he is communicating with the desired service.

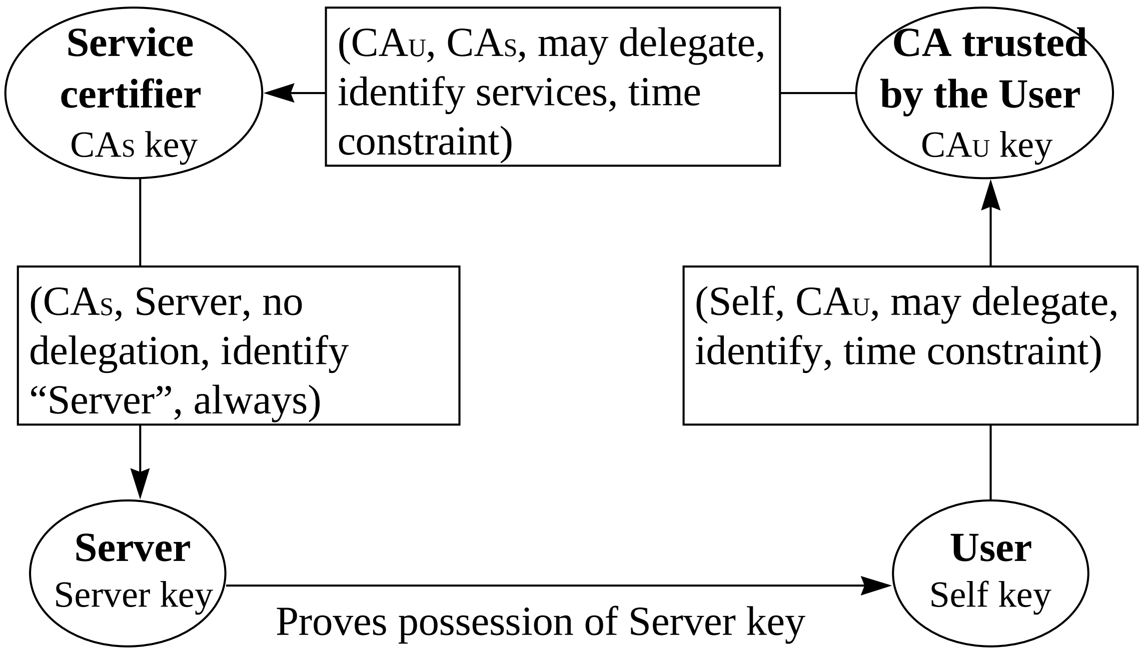


Figure 9 Identification loop: User authenticates the Service

The server, in turn, wants to know if the user has the right to access the service. The service delegates its access right management to an administrator or a Policy Manager. The PM may again further delegate the right to authorize access to the service. In the case of cooperating companies that want to provide extranet access to each other's employees, the first PM may be the administrator of the server. He may delegate the authorization to the security officer of the company, who may further delegate it to the security officers of the other companies. Finally, these security officers would delegate the right to the individual employees of their companies. Access rights and validity periods of the certificates may be restricted along the delegation path. The service's software verifies that the certificates form a valid chain, and when the user requests access, he also proves possession of the private part of the key to close the certificate loop. This on-line authentication step may be performed simultaneously with the on-line authentication identifying the service to the user, described above.

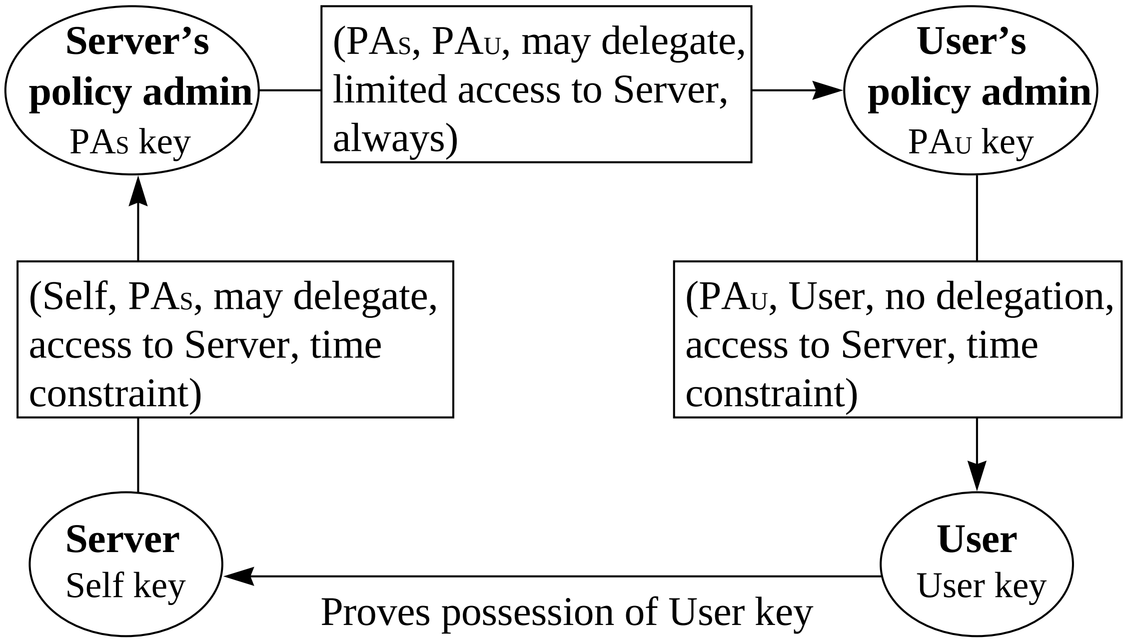


Figure 10 Authorization loop: Service checks that the User is authorized to use it

Traditionally the service would also have been interested in the identity of the user to decide if the user has the right to access the service. Now it only needs to read the authorization field. Furthermore, using authorization certificates has some additional advantages. It makes delegation easier. An even more important reason is the possibility for anonymous access. The server only needs to know that the holder of a certain key is authorized to use the service. It does not have to know the name of the keyholder.

3.4 Policy Management

Recent history has shown on many occasions that local configuration scales badly to the global Internet. Instead, a system that has been designed to be fully distributed, i.e., both deployed and managed in a distributed way, scales extremely well. A prime example of this is the Domain Name System (DNS): it was taken into use when the static hosts file grew too large to manage, and technically it has not needed any major modifications ever since. [NP98]

According to our initial analysis, a system that is based on signed capabilities scales well. SPKI is such a system. It allows all kinds of rights, including administration, to be delegated in a controlled way, allowing administration to be distributed within organisations and between organisations. As parts of security management can be delegated to other people, the burden of the organisation security officer is diminished.

Using a distributed directory service for certificate storage and retrieval also eliminates the need to make the same changes several times, once for each local configuration. This could also be archieved by using some kind of remote configuration mechanism, such as Sun Microsystems Network Information Service (NIS). However, using certificates significantly reduces the security requirements for the directory service if compared to a remote configuration mechanism, since certificates are much harder to forge than regular configuration files and thus do not need any additional protection from integrity threats. Their confidentiality may still need protection.

4. The Design of the New Access Control System

4.1 Design Goals and Requirements

The central idea of our design is to make the Java policy management distributed instead of local. We wanted to see if this could be archieved with the help of SPKI. Changing the local configuration every time a change in the security policy is needed is possible as long as we only have a few different applications with different access rights. However, as the number of new applications and corresponding policy configuration changes grows, it becomes increasingly tedious. In a large organization, it may be nearly impossible. We expect a distributed system to scale better.

Another design goal is to make the Java security policy dynamic. The default policy implementation is static (unless explicitly refreshed). By making the policy dynamic, we expect to improve the system security in situations where the runtime sessions are long, where a static policy might reflect a month-old situation instead of the current one.

The architecture should be extendible. Thus, we also wanted the architecture to make use of the flexibility and extendibility of the Java 1.2 Security Architecture. The “Java way” includes adopting the provider architecture and making the components that build the functionality exchangeable, so that other authors may provide their own implementation of the functionality and add new functionality.

Our implementation should be compatible with applications made for standard JRE 1.2. In fact, our modifications should be transparent to other than security related parts of the system, and normal applications should not need to be aware of them. Consequently, it should be possible to implement it as a regular Java extension.

As with all management tools, whether managing security or something else, usability is an important aspect of the design. However, investigating the usability issues in policy management or trust management is beyond the scope of this study. There has not been very much research on the topic, and thus the principles of creating user interfaces for this kind of software have not been established. Thus, our design should include a user interface that can be easily improved or changed to be more user friendly, according to later usability research. The user interface should be designed according to general usability guidelines for designing user interfaces.

4.2 Functional Specification

Since we want to use SPKI certificates to express security policy, we need a Java implementation of the SPKI certificates and related classes. This implementation should conform to the SPKI specifications. Furthermore, it must include such a tag definition that can express Java security policy, i.e., grant Java permissions. As our design requirements mandate, our SPKI certificate class hierarchy should fit in the Java cryptography architecture. This can be done by making our SPKICertificate class extend the java.security.cert.Certificate class and creating subclasses of the related classes such as the CertificateFactory. In addition, the implementation should be separated from the general interface for SPKI certificates, so that other programmers may provide their own implementation of the SPKI certificates.

In order to sign the certificates and verify the signatures we need a key storage. The key repository must also protect the private signing keys from unauthorized access. Java already includes a KeyStore class that fulfills our functional requirements for a key storage.

The SPKI certificates themselves need to be stored in a certificate repository. The repository must include functions to add new certificates to storage, to retrieve certificates that fulfill some search criteria, and to remove certificates from the storage. The minimum search criteria that must be implemented are search by certificate issuer and search by certificate subject. The Java KeyStore can also be used to store certificates, but it lacks the search criteria required from certificate storage. The repository should be an abstract class and the actual implementation used should be configurable through a configuration file.

The new Java security policy management must use SPKI certificates for expressing the policy. This requires the policy implementation to be able to map the SPKI certificate tags into Java permissions. It must also be able to use the certificate repository to retrieve the certificates relevant to the policy decision at hand, and to verify that the retrieved certificates form a valid chain. The policy implementation may also create Chain Reduction Certificates of the verified certificate chains and cache them for future use.

The purpose of the user interface is to help the policy administrators to create and sign the SPKI certificates that express their security policy. The user interface must guide the user through the process of making valid certificates. The interface should preferably be form-based and must allow easy editing of the certificate fields. The user interface module must also store the created certificates into the certificate repository and help the user to export and import certificates from and to the repository. Finally, the user interface should allow the user to view the certificates stored in the repository. We do not think that the user interface should help the user to create and manage keys, because the Java KeyStore already has a user interface of its own.

4.3 System Architecture

Compared to the Java security architecture summarized in Section 2, our system architecture is fairly similar.

The SPKICertificate is an abstract superclass for all SPKI certificates, and extends the java.security.cert.Certificate class. It includes the SPKI certificate fields and the signature, and provides methods for other classes to be able to read the certificate field contents and to verify the signature.

In Java 1.2, each ProtectionDomain may be decorated with a set of certificates. The certificates are used to define the CodeSource of the domain, which in turn is used to determine what permissions the domain gets. In our prototype, these certificates are SPKI certificates. When the programmer creates an application, he places the application classes in a jar file and adds SPKI certificates that issue to the classes the permissions they need in order to function correctly.

To be able to retrieve the SPKI certificates from the jar files, verify them, and place them in the ProtectionDomains' certificate collection, we have to extend the functionality of the JarVerifier and to provide our own "SignatureFileVerifier", called SPKIVerifier.

Since the jar files can not include all certificates from all sites that might want to run the applications, our architecture also needs a certificate repository to store the certificates local to each organization. We have created a general CertificateRepository class that serves as an abstract superclass for all certificate repositories. The

SPKICertificateRepository extends this superclass and provides an interface to repositories that store SPKICertificates and implement the search criteria required to find certificates to do SPKI chain reduction. The SPKICertificateRepository is also abstract to allow different implementations. Finally, a LocalSPKIRepository class provides the default implementation for the SPKICertificateRepository.

The SPKIPolicy class extends the java.security.Policy class. It uses the SPKICertificate objects retrieved from the class' ProtectionDomain and the SPKICertificateRepository to make the security policy decisions. The SPKIPolicy object searches the repository for valid certificate chains from the Self key of the policy object to the hash of the class and gives the class the permissions delegated through the chain.

Finally, to make the security policy dynamic, we provide our own PermissionCollection subclass called DynamicPermissions. In contrary to the other PermissionCollections, it does not contain a set of Permission objects at all. Instead, its implies() method asks the Policy whether it has the permission in question at the moment when the method was invoked.

Figure 11 summarizes the system architecture and its main differences to standard JDK. The programmer must now decorate the jar file with SPKI certificates, instead of just signing the classes (1). The system administrator, too, now handles SPKI certificates instead of the old configuration file (2). When the class is loaded, the jar file is searched for SPKI certificates and if found, they determine the class' ProtectionDomain (3). When initialized, the ProtectionDomain gets a DynamicPermissions object instead of a static PermissionCollection. When the AccessController asks if the class has a particular permission, the DynamicPermissions in turn asks the same question from the SPKIPolicy (4).

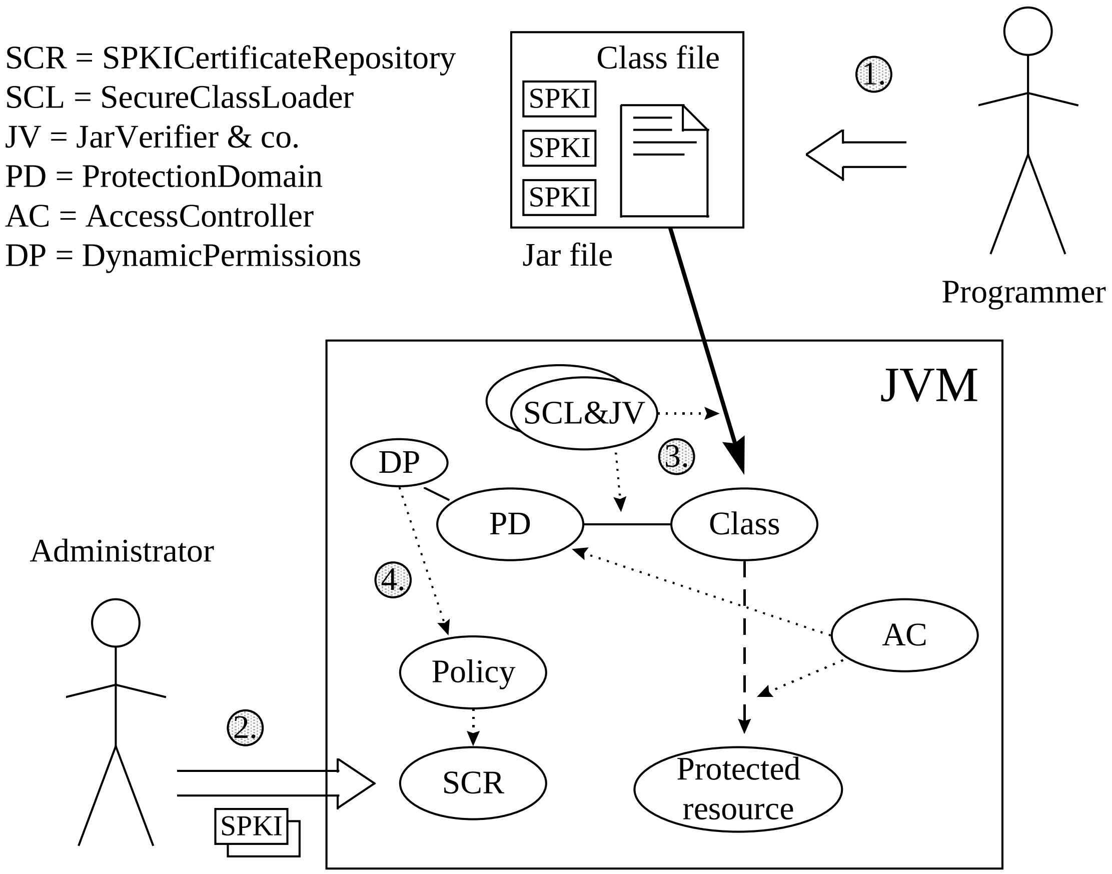


Figure 11 System architecture

4.4 Expressing Java Permissions with SPKI Certificates

The SPKI certificate definition [SPKI98a] specifies the tag field as follows[3]:

<tag>:: "(" "tag" "(*)" ")" | "(" "tag" <tag-expr> ")" ;

<tag-expr>:: <simple-tag> | <tag-set> | <tag-string> ;

The form (tag(*)) means unlimited authorization, i.e. all permissions. When translated to Java permissions it becomes the java.security.AllPermission.

[3] See Appendix A for a full definition of SPKI certificate format.

As we are interested in expressing Java permissions as tags, we extend the SPKI tag definition as follows:

```
<simple-tag>:: <java-permission-tag> ;
<java-permission-tag>:: "(" "java-permission" <permission-type> <permission-target>?
                              <permission-action>? ")" ;
<permission-type>:: "(" "type" <bytes> ")" ;
<permission-target>:: "(" "target" <bytes> ")" ;
<permission-action>:: "(" "action" <bytes> ")" ;
```

The “type” gives the full class name of the permission class in question. This may be a permission type included in JDK or any other class, as long as it is a subclass of the class java.security.Permission. If the constructor of the permission specified by the type takes a target as an argument, that string is given in the “target” field of the certificate. Likewise, if the constructor of the permission takes an action as an argument, it is given in the “action” field of the certificate. The target and action strings are passed to the constructor as-is, because we cannot expect to be able to parse the arguments of all kinds of permissions, as any programmer can define her own types of permissions.

The other fields of SPKI certificates are used as specified in the SPKI definition [SPKI98a]. Appendix B shows examples of SPKI certificates used by our prototype.

4.5 Certificate Storage, Retrieval and Chain Reduction

The basic principles for chain reduction are described in the current SPKI drafts [SPKI98a] [SPKI98b]. Two certificates form a valid chain if the subject of the first certificate is the same as the issuer on the second certificate, both certificates are valid, the delegation field in the first certificate is true, and the tag of one of the certificates is a subset of the tag in the other certificate. If the result was formed into a Chain Reduction Certificate (CRC), it would have the issuer of the first certificate as its issuer, the subject of the second certificate as its subject, the intersection of the validity fields as its validity, the delegation field of the second certificate as its delegation, and the intersection of the tags as its tag.

The SPKI drafts propose that the Prover is responsible of presenting a valid certificate chain to the verifier at the time of access request or authentication [SPKI98b]. This approach effectively moves the burden of certificate storage, retrieval and part of the chain reduction from the server, i.e., the Verifier, to the client software. The server is only left to verify that the chain presented is a valid one. This approach may be suitable to controlling user access, since the user is likely to know which certificates it has been issued and may even be able to store these certificates on a smart card or in some other practical way. However, *mobile code downloaded from the Web cannot know if it has been issued local certificates or not*, and it certainly cannot possess all these certificates from each site it may be downloaded to. Thus, this approach is doomed to fail in our architecture and we do not pursue it any further.

Another possibility is presented by Viljanen and Nikander in [NV98]. They suggest storing the certificates in DNS and present an algorithm for finding the certificates needed for chain reduction. Other proposals for certificate storage and retrieval include using X.500 public directories with the DAP/LDAP protocol for retrieval, and the MIT PGP directory. These solutions have the problem of being identity-oriented and we feel that their retrieval modes would need some modifications to accommodate authorization certificates as well.

Although storing the certificates in DNS seems like a promising solution, investigating this possibility is beyond the scope of this study and we cannot state anything definitive about it. Since none of the storage alternatives seems to be established as the best solution and better solutions may yet be proposed, we have decided to make the certificate storage an interchangeable part of the architecture. We provide such an interface to the certificate repository that the system can use it without having to know whatever the actual implementation providing the functionality is.

Aura has analyzed different chain reduction algorithms, and presents a proof that the chain reduction is a solvable decision technique [Aura97] [Aura98]. The simplest case of chain reduction is the one where all the subjects are keys, i.e. we do not have to consider names, groups or thresholds. In this simple case, checking if a valid chain exists can be done with a simple depth-first graph search algorithm.

Aura proposes starting the reduction from the rear, i.e. the last certificate of the chain. The subject of this certificate is the Prover. This approach has the advantage that we are less likely to work in vain. The Self key has almost always issued at least one valid certificate, but none may be issued to the Prover or the Prover's certificates may not be valid. If this is the case, then the algorithm stops at the beginning and does not attempt to reduce the chain in vain. An additional advantage of starting from the rear is that handling of groups and thresholds becomes easy and efficient. When reaching a group or a threshold, we only need to check if we have already encountered the necessary amount of certificates to follow and if not, we can stop the checking.

However, as Aura concludes, a depth-first algorithm is not very efficient. It is apparently not optimal for finding short paths in graphs that have mostly short paths but may also have a few very long paths that in the worst case do not even lead to the Self key, such as we expect to encounter in our architecture. A breadth-first search would be more efficient in this case. As it, too, has its drawbacks, Aura proposes yet another alternative. The search can be made more efficient with the meet-in-the-middle algorithm. It starts from the rear, but if it has not found a valid chain after reducing a certain number of certificates it starts from the beginning of the would-be chain and works towards the center. It reduces up to some limit of certificates and if the ends still have not met, it concludes that a valid chain does not exist.

When the algorithm encounters a subject that is a name, it must be reduced to a key. This reduction is also fairly straight forward according to the SDSI local namespace scheme. We start from the first name in the chain. It belongs to the authorization certificate's issuer's name space, so we can easily locate the name certificate with the correct name and issuer, and find out the corresponding principal. The next name certificate we need to locate is issued by this principal to the next name in the chain. This way we work our way to the last name in the chain, belonging to the last linked name space, and find the certificate that binds that name to a key. This key is the real subject of the authorization certificate. However, since SPKI supports late binding of names, it is possible that at some point we do not find a certificate that would bind the name to a key. In that case, we give up: the certificates do not form a valid chain.

4.6 Configuration

JDK uses a configuration file to specify several security parameters of the system. The file is `lib/security/java.security` in the JDK directory. This file contains, among others, entries that specify the full class names of the current implementations for the Policy, the KeyStore and the X.509 certificates.

We decided to follow the example and defined two new entries to this configuration file. The first entry specifies the implementation to be used for providing the SPKICertificate functionality, and the second defines the SPKICertificateRepository implementation to be used. The defaults for these properties are the following:

```
spkicertificate.provider=fi.hut.tcm.spki.cert.SPKICert
certstorage.provider=fi.hut.tcm.spki.cert.LocalSPKIRepository
```

In addition, to make the JRE to use our Policy implementation, the "policy.provider" property in the configuration file must be changed from java.security.PolicyFile to our implementation:

```
policy.provider=fi.hut.tcm.spki.policy.SPKIPolicy
```

The Java launcher also includes a command line option that can be used to set or override the security properties when starting an application. The option is -D, followed immediately with the property definition as it would appear in the configuration file. An example command line could look like this:

```
$ java -Dpolicy.provider=fi.hut.tcm.spki.policy.SPKIPolicy \
com.company.MyApplication
```

4.7 Modifications to the Java Class Library

As stated above, we want to decorate the Jar files with SPKI certificates instead of signature files, and we want these certificates to determine the classes' protection domain. The class that parses the Jar file and initiates the verifying of the signature files is the java.util.jar.JarVerifier. Unfortunately, the signature file types that this class understands (.SF, .DSA and .RSA) are hard coded in its source code. Adding new types of "signature files" to be verified cannot be done without tampering with this class. Thus, we were forced to add to the JarVerifier the signature file type ".SPKI", and to make it invoke our own SPKIVerifier when such a file is encountered.

Although not as necessary as the modification of the JarVerifier, we would also like to modify the java.security.cert.Certificate class. As mentioned before, this class was designed to be a superclass for identity certificates. As we would like to use it as a superclass for authorization certificates as well, we feel that it should reflect the common features of all certificates, instead of just the common features of identity certificates.

Identity certificates and authorization certificates have several common features. They are both signed records, so they have a signature. They have a transfer representation, an encoded form common to all implementations. They also have an issuer, and the issuer has a key pair to be able to sign the certificate. Furthermore, the certificates have a subject, i.e., some entity that the certificate was issued to. However, the type and properties of this subject may vary. Finally, they have validity information, usually in the form of “not before” and “not after” dates.

The java.security.cert.Certificate class has, as mentioned before, several methods. As any object, it has the methods equals, hashCode and toString. These methods are obviously common to identity and authorization certificates. The getEncoded method returns the encoded transfer representation of the certificate, and is also common to all certificate types. Likewise, the getType and verify methods have the same intuitive meaning and serve an important purpose in both identity and authorization certificates. The last method, getPublicKey, is more problematic. In identity certificates, it is used to get the subject’s public key from the certificate. In authorization certificates, the subject may not even have a public key. However, in all certificates the issuer has to have a public and private key to be able to sign the certificate. Thus, the most logical key to be returned by the getPublicKey method in authorization certificates would be the issuer’s key that can be used to verify the signature. Nevertheless, having the same method to return completely different parts of the certificate, depending on which type of certificate is in question, is obviously not a good idea.

We propose that the getPublicKey method in the Certificate class be changed to getSubjectKey. Furthermore, since not all certificates’ subjects do have a public key, the method should throw a NoSuchFieldException if the public key does not exist. We also suggest that a method called getIssuerKey would be added to the Certificate class. This method would return the public key that can be used to verify the certificate’s signature.

5. Implementation

Our prototype implementation comprises two general parts. The first part is to continue previous work by Lehti, to provide a working implementation of SPKI certificates that is compatible with the Java 1.2 cryptography architecture. The other part is to generate the extensions needed for the Java security architecture in order to use the SPKI certificates for security policy definition and management. The rest of this chapter describes our implementation in more detail.

5.1 Prototype Class Model

Translating the system architecture as described in Chapter 4 into a class model is quite straightforward.

In his Master's Thesis [Leh97], Ilari Lehti has described a class model for SPKI certificates. As the SPKI drafts have evolved, the class model is slightly outdated. A principal can no longer be a secret key, and the issue time field has been obsoleted. We adopt the basic model with these changes, and further extend it to fit in as a subclass for Java's Certificate class as specified in Chapter 4.

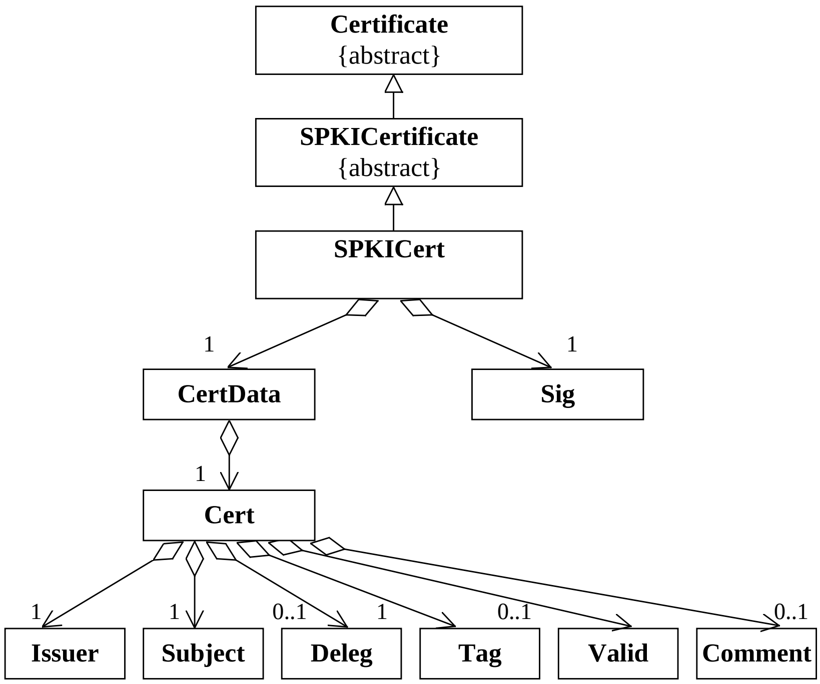


Figure 12 Top level view of the SPKI certificate model

The tags are further defined to have the following class structure:

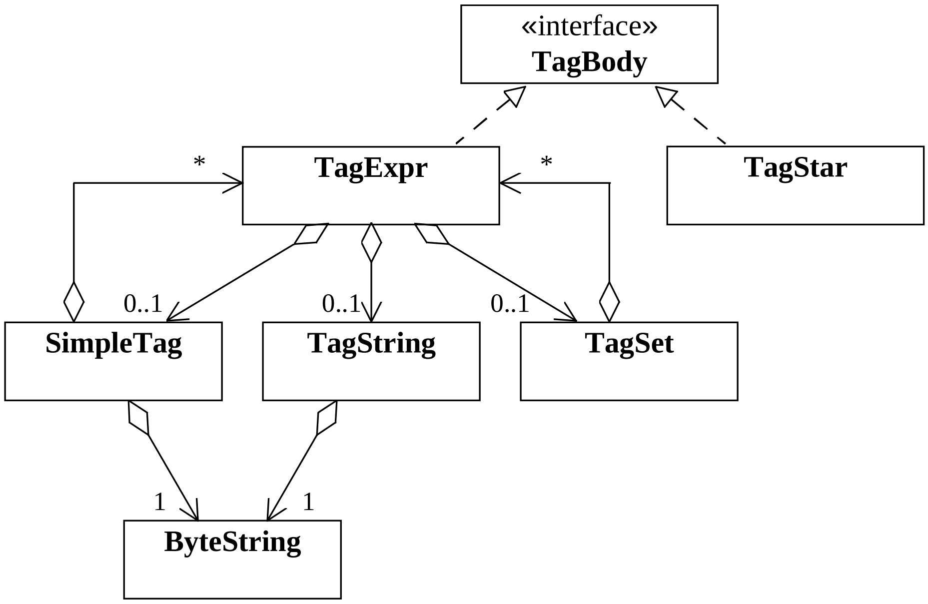


Figure 13 SPKI tag classes

As described in Chapter 4, we have extended the tag definition to be more suitable for expressing Java permissions. The new extensions to the tag definition are shown in the class diagram in Figure 14.

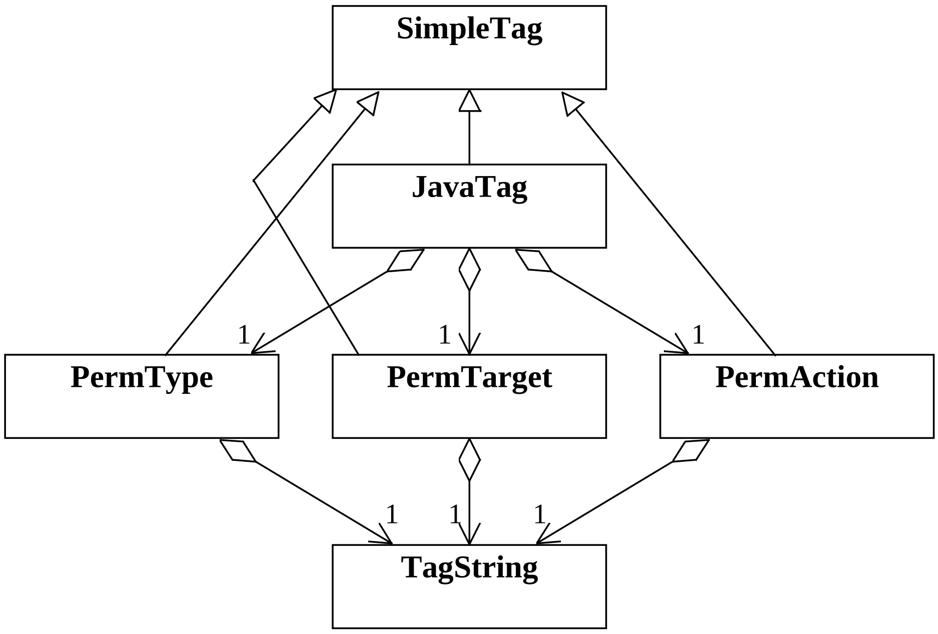


Figure 14 Class model of Java tags

Our prototype includes our own Policy class called SPKIPolicy. The class diagram in Figure 15 shows the SPKIPolicy class and the repository classes it uses for storing keys and certificates.

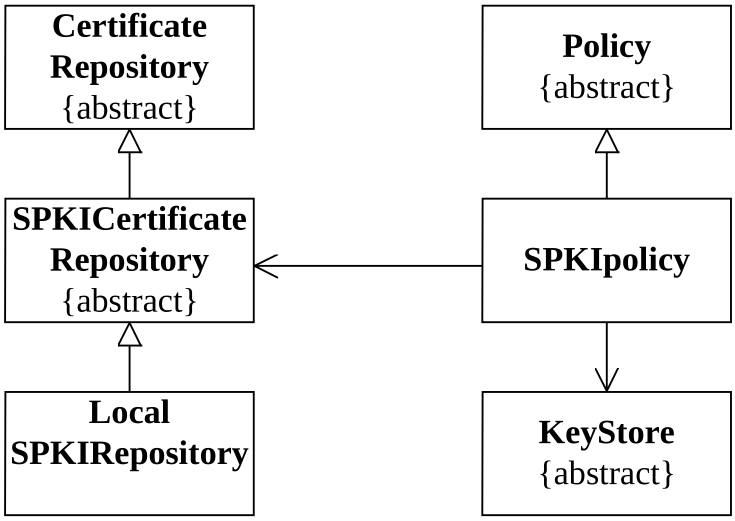


Figure 15 Policy class model

Finally, to make the Java security policy dynamic instead of static, we have created our own PermissionCollection subclass, which fetches its permissions from the SPKIPolicy object when the ProtectionDomain's permissions are checked. The new Dynamic-Permissions class and related classes are shown in Figure 16.

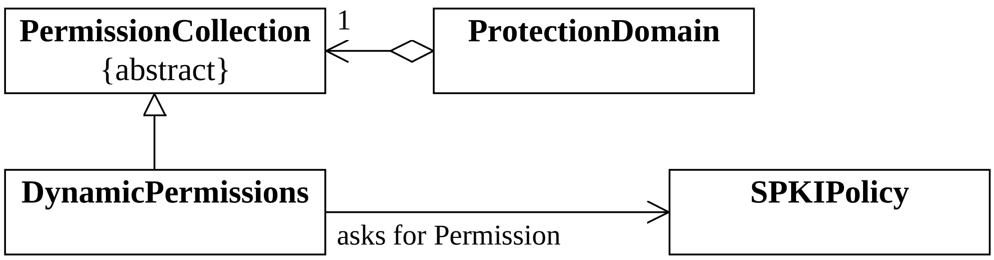


Figure 16 DynamicPermissions class

5.2 Prototype Functionality

Our prototype does not implement all of the functionality specified in Chapter 4. We decided to build only the central parts necessary for evaluating the concept.

Our prototype implementation of the certificate storage is not distributed. It stores the certificates in a file on local hard disk. Otherwise, it performs all the tasks specified in the previous chapter: adding and removing certificates, and fetching certificates by issuer and subject.

The SPKI implementation is built according to the minimum requirements. The subject can only be a hash value or a principal, i.e., a public key or a hash of a public key. No name certificates or threshold subjects are handled. The validity field may only consist of not-before and not-after dates, and thus may not include any on-line tests, as their handling has not been defined in sufficient detail in the SPKI drafts. No revalidation is implemented. Finally, of the fields that are not related to security, only the version and comment fields are implemented. The comment field does naturally not affect the certificate reduction. Finally, no Chain Reduction Certificates are created.

When a class is loaded, the jar file is searched for SPKI certificates in addition to normal signatures. To make the Java Runtime read SPKI certificates from the jar files and put them to the protection domains we had to create a class of our own that handles the SPKI file verification. In addition, we had to slightly modify the java.util.jar.Jar-Verifier to make it look for SPKI certificate files and invoke our SPKI verifier.

The SPKIPolicy implementation uses SPKICertificate objects, instead of X509Certificate objects and codebases, to determine what permissions the protection domains get. When the ProtectionDomain is initialized, and gets its Permission-Collection object from the SPKIPolicy, it actually gets an instance of the Dynamic-Permissions class. When the access rights of the ProtectionDomain are checked by invoking the implies() method in the DynamicPermissions object, the implies method in turn calls a getPermission() method in the SPKIPolicy object. This method takes as arguments the SPKICertificates the ProtectionDomain has, and the Permission object that the implies method was asked to check. It uses the protection domain’s SPKI certificates and SPKI certificates found in the certificate repository as it tries to reduce them to a valid chain that would delegate the permission in question to the protection domain. It returns a boolean value, depending on whether the reduction was successful or not. The implies() method relays this answer further to the AccessController.

The user interface of the prototype is a very simple command line tool. It asks the user for the KeyStore location and password, then asks for the certificate field contents, creates the certificate, signs it, and finally asks for the CertificateRepository location and places the certificate in the repository. The tool includes command line options for all the information it needs, so that the tool can also be used non-interactively. The `-help` option prints brief instructions for using the tool.

```
$  java fi.hut.tcm.spki.UI –help
Usage: java fi.hut.tcm.spki.UI [options]
Options:
[-storepass <passwd>]
[-keypass <keypasswd>]
[-manifest <outputfilename>]
[-issuer <keyalias>]
[-subject <keyalias>]
[-nodeleg] | [-deleg]
[-type <full permission class name>]
[-target <targetstring>]
[-action <actionstring>]
```

5.3 Use of Design Patterns

Gamma defines design patterns to be simple and elegant solutions to specific problems in object-oriented software design [Gam95]. However, often these solutions are not the first ones that spring to mind when encountering a design problem. As the first solution usually is not as good as the well documented and tested design pattern, it is considered a good design practice to use the patterns where applicable. In his book, he described the most widely known and commonly accepted design patterns and their use [Gam95].

During the design of our class model, we have consciously tried to identify the design patterns applicable in our model. We have identified and used four patterns: Composite, Visitor, Singleton and Abstract Factory.

The Composite pattern is a structural pattern that is defined to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly. In our design, the Composite pattern is used extensively in the SPKI certificate package. The certificate parts, CertData and Signature are tree formations of the different field types, which in turn may be composites of other entities.

The Visitor pattern is a behavioral pattern that represents an operation to be performed on the elements of an object structure. It lets the designer define a new operation without changing the classes of the structure. We have used this pattern to implement operations that need to go through the SPKICertificate object structure, such as creating the transfer representation of the certificate or printing it.

The Singleton pattern is a very simple, creational design pattern that ensures a class only has one instance, and provides a global point of access to it. Just as Java designers have used this pattern to ensure that in the JVM there is only one instance of the AccessController, Policy and KeyStore, we have used it to ensure that there is only one SPKICertificateRepository.

The "provider" architecture in Java follows the Abstract Factory pattern. This pattern provides an interface for creating families of related or dependent objects without specifying their concrete class. We have used this pattern in the certificate repositories, so that a user can request an instance of the abstract interface class, without having to worry about the class that actually implements the functionality.

6. Evaluation and Consideration

6.1 Performance Measurements

We measured the performance of the prototype to investigate the applicability of the model. As a static security policy obviously has some performance advantages compared to a dynamic policy, we wanted to compare the prototype with the default implementation provided with JDK to see if the difference was acceptable. Since the main performance changes to the default JDK implementation occur in class loading and permission checking, these two tasks are the ones we measured. A simple test program measured the time to load ten classes that belonged to different protection domains and the time to perform ten thousand access control checks.

For the test, we created the same security policy both in the form of a policy configuration file and a set of SPKI certificates. The average length of SPKI certificate loops was three certificates. We also created ten similar classes and signed them with different keys. The following table shows the classes and the permissions assigned in the policy.

Class	**Permissions granted in the security policy**
TestClass1	java.io.FilePermission("<<ALL FILES >>", "read")
TestClass2	java.io.FilePermission("/tmp/myfile.txt", "read,write") java.util.PropertyPermission("user.*", "read")
TestClass3	java.lang.RuntimePermission("Class.getProtectionDomain") java.util.PropertyPermission("*", "read") java.net.SocketPermission("www.tcm.hut.fi", "connect")
TestClass4	java.net.SocketPermission("*.tcm.hut.fi:80", "connect") java.lang.RuntimePermission("exit")
TestClass5	java.util.PropertyPermission("*", "read") java.io.FilePermission("/-", "read,execute")
TestClass6	java.io.FilePermission("/temp/-", "read,write,execute")

TestClass7	java.util.PropertyPermission("user.*", "read") java.util.PropertyPermission("java.*", "read") java.net.SocketPermission("*.tcm.hut.fi:80", "connect")
TestClass8	java.lang.RuntimePermission("getPolicy") java.lang.RuntimePermission("setPolicy")
TestClass9	java.security.AllPermission()
TestClass10	— (gets default permissions, i.e. ends up in the sand box)

Table 4 The test situation

Originally, we expected the class loading to get slightly slower or stay the same, as we would not need to figure out what permissions new protection domains would get, but would instead need to resolve the certificates from the jar files. As class loading is fairly well optimized in the JDK, it was also possible that no change in the performance would be noticed. As to the permission checking, we expected the access right check to be slower, since we not only verify whether the class' permissions imply the permission needed, but also resolve what permissions the class has at the moment.

The measurement performed during quiet hours and it was repeated 10 times to get a reasonable average of the performance. The test environment was Solaris 2.6 running on Ultra1 hardware. The JDK version used was JDK1.2beta4. The results are given in the following table.

	JDK	Our Prototype
Time to load 10 classes that belong to different protection domains	1690 ms	4990 ms
Time to resolve 10000 access rights	38900 ms	39200 ms

Table 5 Test results

Our prototype is not optimized in any means; it does some unnecessary work. At the moment it handles the SPKI certificates of the classes to be loaded *in addition* to the regular signatures and not *instead* of them, although they are not used for anything in our system. This is a part of why the class loading in our prototype takes so much more time than in the default configuration. Another reason is that our test jar files only had one regular signature, but several SPKI certificates attached to them.

The results show that our system is more than three times slower in class loading and only slightly slower in access checking. Given that our prototype has not been optimized, we feel that the performance is quite satisfactory. Especially the speed of access right checking in our prototype surprised us.

6.2 Security Analysis

In Section 2.5 we noted that a dynamic security policy has clear security advantages when compared to a static policy, as changes to the policy take effect immediately. Revoking permissions when misuse is detected should always have an immediate influence to stop further damage, and this is the case in our architecture. In this aspect, we claim that our design is more secure than the default architecture in JDK.

Another source for security problems in JDK 1.2 is the use of source URL in addition to the X.509 certificates in defining which permission domain the classes belong to, and thus what permissions they get. To us, using URLs seems like a bad idea from the security point of view. The URLs consist of the domain name of the source machine and an arbitrary string. Until secure DNS is deployed, the domain name or any other DNS information cannot be used for security purposes because it is too easy to falsify. In our design, the URL has no relevance in determining what permissions the classes get, so the security hazard of false domain names is avoided.

Furthermore, our design changes the integrity requirements to the different parts of the system. In JDK, protecting the policy configuration files of both the system and the individual users is essential, as any applet or application that gains write access to these files can give itself any permissions it desires. In our design, the certificates do not need any particular protection from unauthorized change, since their signatures already prevent any unauthorized changes to go unnoticed. Naturally, tampering with the certificates may result in denial of service, as certificates whose signatures cannot be verified are ignored. Although this is also a serious threat, it is not as fatal as the possibility for malicious programs to gain arbitrary access rights in the system. In our design, the private signature keys need to be carefully protected from unauthorized use. This is no different from the requirements in the JDK, so conclude that our system is more resistant against integrity attacks.

However, our prototype implementation has not undergone thorough peer review or extensive security testing, and thus it is possible if not even probable that the implementation includes bugs that lessen the security of our design.

6.3 Suitability of SPKI Style Authorization for Java Access Control

As stated in Chapter 2, the Java access control is inherently capability-based, as it uses Permission objects as tickets represent access rights inside the JVM. Conceptually, extending this practice outside the JVM by using authorization certificates is not a great leap.

With the ever-growing use of Java applets and applications for real work and not just making Web pages more attractive, the security issues become more and more important. Among the most important of these issues is security policy management: it should be easy and efficient. The current management solution, the ACL type configuration file does not scale well. According to our analysis, basing the policy management on authorization certificates is an easily scalable solution, provided that storing and locating the certificates can be done in a distributed and effective manner. There are already several distributed directory systems that could be used to fulfill this requirement: an excellent example, although not the only one, is the DNS system. Consequently, using SPKI certificates seem to be a good way to solve the scalability problem.

As the performance measurements and the security analysis suggest, using SPKI certificates to define the Java security policy is clearly applicable, and may be even more attracting than the way Java policy management is done today.

6.4 Suggestions for Improvement and Further Work

The most important deficiency of our design and implementation is the lack of a proper distributed directory service for certificate storage and retrieval. Without the distributed certificate repository, our policy configuration is just as local as the original configuration file that we wanted to replace. The different alternatives deserve a thorough evaluation.

Another important research topic that we could not include in this study is evaluating the usability issues in authorization certificate based policy management. The user interface of our prototype is rather crude and probably not very user friendly, and would definitely benefit of further development. To be able to create really easy-to-use policy management software we should be aware of the concepts and assumptions of the users, how they see their management tasks, and what are the typical use scenarios. This is still an unexplored field.

As mentioned in Chapter 1, one of the goals of this study was to create a ground for future extensions to JDK that enable fully distributed security domains between several JVMs and secure communication channels between them. Now that distributed security management for Java, the basic requirement for such a development, has been established, we expect the research on distributed security domains to proceed.

7. Conclusions

In this Thesis we have investigated possibilities for implementing distributed policy management for Java 1.2 and shown that it can be done effectively and securely using SPKI certificates.

In Chapter 2, we have described the Java 1.2 security architecture and its shortcomings. JDK 1.2 uses a capability-based approach for representing the access rights inside the JVM, but this otherwise relatively beautiful design is impaired by using a local configuration file to define what capabilities each class gets. Chapter 3 presents different certificate types in general and SPKI certificates in particular, and discusses some policy management issues.

The central idea of our design is to replace the local policy configuration files with distributed policy management, implemented with the help of SPKI certificates. In our design, explained in Chapter 4, the programmer decorates the jar files with SPKI certificates that directly indicate what permissions the classes would like to have. These certificates determine the classes' protection domain, but the actual permissions the classes have at each time point depend on the valid certificate chains leading to the classes at that time.

The actual prototype implementation, described in Chapter 5, consists of a generic SPKI certificate package that extends the java.security.cert.Certificate class, the custom policy manager, a dynamic PermissionCollection class, and the minor modifications needed to the library proper. For the purpose of this prototype, we only implemented a local certificate repository. The class hierarchy of the SPKI package is fairly complex, due to the number of fields in the certificate and the great variety of possible contents to the fields. However, the classes themselves are relatively simple and straightforward. Although the prototype is not optimized in any way, its performance is clearly adequate, especially in the permission checking.

During this work, we made some observations about the Java security architecture and its extendibility. Java security architecture was built to be extendible and to support implementation independence, and many parts of the system show extraordinarily good judgement. However, in spite of these fine design principles, not all parts of the JDK are well thought and easily extendible. The JarVerifier that reads and interprets jar files and their signatures is inflexible. It is impossible to extend its functionality without replacing or modifying the class in the JDK library. This, obviously, is not a desirable feature if we want to distribute our authorization certificate packages as a regular extension to JDK. As Java 1.2 is still in beta phase at the time of this writing, we can hope that this shortcoming gets fixed in the final version.

The basic Java 1.2 cryptographic architecture is a rather good starting point for adding new types of certificates. However, the Certificate class needs some minor changes to be suitable as a superclass for authorization certificates. The getPublicKey method does not have an intuitive meaning in authorization certificates, or at least the interpretation is not the same as for identity certificates. We propose renaming this method to getSubjectKey and letting it throw a NoSuchFieldException if the certificate's subject is not a public key or does not have a (known) public key. In addition, since every certificate has an issuer, and every issuer must have a public key to sign the certificate, we suggest adding a getIssuerKey method to the Certificate class. We see the goals of extendibility and implementation independence as worth pursuing, and have tried to further advance them in our design by following the Java "provider architecture" example and separating the general API for SPKI certificates and the actual implementation from each other.

In the future, more research is needed to investigate the different alternatives for creating a distributed certificate repository and to study the usability aspects of using authorization certificates in security policy management.

References

[Amo94] E. Amoroso. Fundamentals of Computer Security Technology. Prentice-Hall, 1994.

[Arn96] K. Arnold, J. Gosling. The Java Programming Language. Addison-Wesley, 1996.

[Aura98] T. Aura. "On the Structure of Delegation Networks". *Proceedings of the 11th Computer Security Foundations Workshop*, IEEE Computer Society Press, June 1998.

[Aura97] T. Aura. "Comparison of Graph-Search Algorithms for Authorization Verification in Delegation". *Proceedings of the 2nd Nordic Workshop on Secure Computer Systems*, Helsinki, 1997.

[Bla96] M. Blaze, J. Feigenbaum, J. Lacy. "Decentralized Trust Management". *Proceedings of IEEE Conference on Security and Privacy*, Oakland, CA, May 1996.

[SPKI98a] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylönen. *Simple Public Key Certificate.* Internet Draft, work in progress, March 1998. Expires 18 September 1998.

[SPKI98b] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylönen. *SPKI Certificate Theory.* Internet Draft, work in progress, March 1998. Expires 16 September 1998.

[SPKI98c] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylönen. *SPKI Examples.* Internet Draft, work in progress, March 1998. Expires 15 September 1998.

[Gong98] L. Gong. *Java™ Security Architecture (JDK 1.2)*, DRAFT DOCUMENT (Revision 0.9), `<URL:http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html>`, Sun Microsystems, June 1998.

[GS98] L. Gong, R. Schemers. “Implementing Protection Domains in the Java Development Kit 1.2”, *Proceedings of the 1998 Network and Distributed System Security Symposium,* Internet Society, March 1998.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Har88] N. Hardy. “The Confused Deputy” [online revised]. *Operating System Review*, vol. 22 #4, October 1988, pp 36-38 [referenced 1998-07-02]. `<URL:http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html>`

[JAVA98] “Java™ Platform 1.2 Beta 4 API Specification” [online]. `<URL:http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html>` [referenced 1998-08-03]. Sun Microsystems, 1998.

[Lam73] B. Lampson. “A Note on the Confinement Problem” [online]. *Communications of the ACM,* vol 16 #10, January 1973 [referenced 1998-07-02]. `<URL:http://www.cis.upenn.edu/~KeyKOS/Confinement.html>`

[Lan89] C. Landau. “Security in a Secure Capability-Based System” [online], *Operating Systems Review,* October 1989, pp 2-4 [referenced 1998-07-02]. `<URL:http://www.cis.upenn.edu/~KeyKOS/Security.html>`

[Leh97] I. Lehti. SPKI-based Access Control Server. Master’s Thesis, Helsinki University of Technology, Helsinki, 1997.

[LN98] I. Lehti, P. Nikander. “Certifying Trust”. *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) ’98,* Springer-Verlag, February 1998.

[McG97] G. McGraw, E. Felten. Java Security: Hostile Applets, Holes, and Antidotes. John Wiley & Sons, New York, 1997.

[MW94] Merriam-Webster's Collegiate Dictionary [online], Tenth Edition. Merriam-Webster, Inc., 1994. `<URL:http://www.eb.co.uk/>`

[Nik97] P. Nikander. Modelling of Cryptographic Protocols - A Concurrency Perspective. Licenciate's Thesis, Helsinki University of Technology, 1997.

[NP98] P. Nikander, J. Partanen. "Distributed Policy Management for Java 1.2", unpublished manuscript, submitted to NDSS'99.

[NV98] P. Nikander, L. Viljanen. "Storing and Retrieving Internet Certificates", unpublished manuscript, submitted to NordSec'98.

[PGP95] P. Zimmermann. The Official PGP Users Guide. MIT Press, 1995.

[Sch94] B. Schneier. Applied Cryptography, algorithms, protocols and source code in C. Second Edition, John Wiley & Sons, 1994.

[SDSI96] R. Rivest, B. Lampson. "SDSI – A Simple Distributed Security Infrastructure". *Proceedings of the 1996 Usenix Security Symposium*, 1996.

[Tho84] K. Thompson. Reflections on Trusting Trust. *Communcations of the ACM*, vol 27 #8, 1984.

[X.509] ITU-T Recommendation X.509, *Information Technology – Open Systems Interconnection – The Directory: Authentication Framework*. November 1993.

Appendix A: SPKI Certificate Definition

This definition was copied from the SPKI Certificate Structure draft [SPKI98a].

Top Level Objects

The list of BNF rules that follows is sorted alphabetically, not grouped by kind of definition. The top level objects defined are:

<5-tuple>: an object defined for documentation purposes only. The actual contents of a 5-tuple are implementation dependent.

<acl>: an object for local use, which might be implementation dependent. An ACL is not expected to be communicated from machine to machine.

<crl>, <delta-crl> and <reval>: objects returned from on-line tests.

<sequence>: the object carrying keys, certificates and on-line test results from Prover to verifier.

Alphabetical List of BNF Rules

<5-tuple>:: <issuer5> <subject5> <deleg5> <tag-body5> <valid5> ;

<acl-entry>:: "(" "entry" <subj-obj> <deleg>? <tag> <valid>? <comment>? ")" ;

<acl>:: "(" "acl" <version>? <acl-entry>* ")" ;

<byte-string>:: <bytes> | <display-type> <bytes> ;

<bytes>:: <decimal> ":" {binary byte string of that length} ;

<cert-display>:: "(" "display" <byte-string> ")" ;

<cert>:: "(" "cert" <version>? <cert-display>? <issuer> <issuer-loc>? <subject> <subject-loc>? <deleg>? <tag> <valid>? <comment>? ")" ;

<comment>:: "(" "comment" <byte-string> ")" ;

<crl>:: "(" "crl" <version>? <hash-list> <valid-basic> ")" ;

<date>:: <byte-string> ;

<ddigit>:: "0" | <nzddigit> ;

<decimal>:: <nzddigit> <ddigit>* | "0" ;

<deleg5>:: "t" | "f" ;

<deleg>:: "(" "propagate" ")" ;

<delta-crl>:: "(" "delta-crl" <version>? <hash-of-crl> <hash-list> <valid-basic> ")" ;

<display-type>:: "[" <bytes> "]" ;

<fq-name>:: "(" "name" <principal> <names> ")" ;

<general-op>:: "(" "do" <byte-string> <s-part>* ")" ;

<gte>:: "g" | "ge" ;

<hash-alg-name>:: "md5" | "sha1" | <uri> ;

<hash-list>:: "(" "canceled" <hash>* ")" ;

<hash-of-crl>:: <hash> ;

<hash-of-key>:: <hash> ;

<hash-op>:: "(" "do" "hash" <hash-alg-name> ")" ;

<hash-value>:: <byte-string> ;

<hash>:: "(" "hash" <hash-alg-name> <hash-value> <uris> ")" ;

<issuer-loc>:: "(" "issuer-info" <uris> ")" ;

<issuer-name>:: "(" "issuer" "(" "name" <principal> <byte-string> ")" ")" ;

<issuer5>:: <key5> | "self" ;

<issuer>:: "(" "issuer" <principal> ")" ;

<k-val>:: <byte-string> ;

<key5>:: <pub-key> ;

<keyholder-obj>:: <principal> | <name> ;

<keyholder>:: "(" "keyholder" <keyholder-obj> ")" ;

<low-lim>:: <gte> <byte-string> ;

<lte>:: "l" | "le" ;

<n-val>:: <byte-string> ;

<name-cert>:: "(" "cert" <version>? <cert-display>? <issuer-name> <subject> <valid> <comment>? ")" ;

<name>:: <relative-name> | <fq-name> ;

<names>:: <byte-string>+ ;

<not-after>:: "(" "not-after" <date> ")" ;

<not-before>:: "(" "not-before" <date> ")" ;

<nzddigit>:: "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

<obj-hash>:: "(" "object-hash" <hash> ")" ;

<one-valid>:: "(" "one-time" <byte-string> ")" ;

<online-test>:: "(" "online" <online-type> <uris> <principal> <s-part>* ")" ;

<online-type>:: "crl" | "reval" | "one-time" ;

<op>:: <hash-op> | <general-op> ;

<principal>:: <pub-key> | <hash-of-key> ;

<pub-key>:: "(" "public-key" <pub-sig-alg-id> <s-expr>* <uris> ")" ;

<pub-sig-alg-id>:: "rsa-pkcs1-md5" | "rsa-pkcs1-sha1" | "rsa-pkcs1" | "dsa-sha1" | <uri> ;

<range-ordering>:: "alpha" | "numeric" | "time" | "binary" | "date" ;

<relative-name>:: "(" "name" <names> ")" ;

<reval-body>:: <one-valid> | <valid-basic> ;

<reval>:: "(" "reval" <version>? <subj-hash> <reval-body> ")" ;

<s-expr>:: "(" <byte-string> <s-part>* ")" ;

<s-part>:: <byte-string> | <s-expr> ;

<seq-ent>:: <cert> | <name-cert> | <pub-key> | <signature> | <op> | <reval> | <crl> | <delta-crl> ;

<sequence>:: "(" "sequence" <seq-ent>* ")" ;

<sig-val>:: <s-part> ;

<signature>:: "(" "signature" <hash> <principal> <sig-val> ")" ;

<simple-tag>:: "(" <byte-string> <tag-expr>* ")" ;

<subj-hash>:: "(" "cert" <hash> ")" ;

<subj-obj>:: <principal> | <name> | <obj-hash> | <keyholder> | <subj-thresh> ;

<subj-thresh>:: "(" "k-of-n" <k-val> <n-val> <subj-obj>* ")" ;

<subject-loc>:: "(" "subject-info" <uris> ")" ;

<subject5>:: <key5> | <fq-name5> | <obj-hash> | <keyholder> | <subj-thresh> ;

<subject>:: "(" "subject" <subj-obj> ")" ;

<tag-body5>:: <tag-expr> | "null" ;

<tag-expr>:: <simple-tag> | <tag-set> | <tag-string> ;

<tag-prefix>:: "(" "*" "prefix" <byte-string> ")" ;

<tag-range>:: "(" "*" "range" <range-ordering> <low-lim>? <up-lim>? ")" ;

<tag-set>:: "(" "*" "set" <tag-expr>* ")" ;

<tag-star>:: "(" "tag" "(*)" ")" ;

<tag-string>:: <byte-string> | <tag-range> | <tag-prefix> ;

<tag>:: <tag-star> | "(" "tag" <tag-expr> ")" ;

<up-lim>:: <lte> <byte-string> ;

<uri>:: <byte-string> ;

<uris>:: "(" "uri" <uri>* ")" ;

<valid-basic>:: <not-before>? <not-after>? ;

<valid5>:: <valid-basic> | "null" | "now" ;

<valid>:: <valid-basic> <online-test>* ;

<version>:: "(" "version" <byte-string> ")" ;

Appendix B: Certificate Examples

In this appendix, we show a couple of examples of SPKI certificates. They are shown in advanced or canonical format, because the official transfer format, BASE64 encoded canonical form, is not appropriate for illustrating the use of these certificates.

Authorization Certificates

The most interesting part of an authorization certificate is the tag field. The following table shows some examples of different Java Permissions and the corresponding tag field definitions in advanced form.

Java Permission	**Tag field**
java.lang.RuntimePermission("Class.getProtectionDomain")	(tag (type java.lang.RuntimePermission) (target Class.getProtectionDomain))
java.net.SocketPermission("www.domain.com:80", "connect")	(tag (type java.net.SocketPermission) (target www.domain.com:80) (action connect))
java.net.NetPermission("Authenticator. requestPasswordAuthentication")	(tag (type java.lang.RuntimePermission) (target "Authenticator. requestPasswordAuthentication"))
java.io.FilePermission("<<ALL FILES>>", "read")	(tag (type java.io.FilePermission) (target <<ALL FILES>>) (action read))
java.awt.AWTPermission("systemClipboard")	(tag (type java.awt.AWTPermission) (target systemClipboard))

The following example shows a full SPKI authorization certificate in canonical format. This certificate delegates the subject key the Java FilePermission that grants execute access to files in the directory `/home/jhp/bin` and its subdirectories. The subject does not have the right to delegate the access further, and the certificate is valid forever.

```
(8:sequence
(4:cert
(7:version1:)
(6:issuer(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBuDCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYUAAoGBAOXVb4qEGdhiiJ8ffpHjwz7yhU7FiUmG3MUh
2GhA3tXZvgRWqIoJbSHIrI6CfkYmZH4/m7TQLeNWrUKcT74icEgedPwhGFpBtQpXkffNhopnxZHW
+Eg/4BgxPGWPXWXtZaIu0B8paAOh8xqeO1nGn0MfL49aLokrwkqn4GlP/UXo
|)))
(7:subject(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBtzCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQAAoGAP5EtSn7Yv8Oa0Q5xaHULXJhw6dSHzNeYp4s0
X47gpIzJF4yR0hL70QBBsI5UN15vmkALf3amfDjsnr5WCJ68ffe6Zzrqy4XF9mKUD4LPePvpB5fl
9bDsNxfMA6agAxy6arg+9pYu68OgpUT+NJr1wUWVB70rOTs0NfB8+l5v/vU=
|)))
(3:tag(15:java-permission
(4:type22:java.io.FilePermission)
(6:target15:/home/jhp/bin/-)(6:action7:execute))))
(9:signature(4:hash4:sha131:|QmaS4ZGK7ZSQjCMQeysu2GxWWB8=
|)(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBuDCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYUAAoGBAOXVb4qEGdhiiJ8ffpHjwz7yhU7FiUmG3MUh
2GhA3tXZvgRWqIoJbSHIrI6CfkYmZH4/m7TQLeNWrUKcT74icEgedPwhGFpBtQpXkffNhopnxZHW
+Eg/4BgxPGWPXWXtZaIu0B8paAOh8xqeO1nGn0MfL49aLokrwkqn4GlP/UXo
|))67:|MC0CFQCAHfQXlPC741htSv5aXTO2mSqORAIUdAdtHs0juf0NMTPGM/ar5WgZ9wo=
|))
```


The next example shows an authorization certificate that transfers all rights, including the right to delegate them further, from the issuer to the subject. This certificate is also valid forever. The certificate is presented in canonical format.


```
(8:sequence(4:cert(7:version1:0)(6:issuer(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBtzCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
```

```
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQAAoGAMPvUPIjkK+JOc8+Ng63Mf3lirEZXtuwBgP6Y
3M0ix+eOwK9NoTZ3BU8XWbQ+WWsLRBcc+tV97G2IhWkYuWmohsf7de6Mj/wSiSYwX1jQreRwocYu
236YzO3XvciftArZQhHwecVZA1zTKrbuyC+NSuCIMhFriskW0kfHmgJ23yY=
|)))(7:subject(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBuDCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYUAAoGBAOXVb4qEGdhiiJ8ffpHjwz7yhU7FiUmG3MUh
2GhA3tXZvgRWqIoJbSHIrI6CfkYmZH4/m7TQLeNWrUKcT74icEgedPwhGFpBtQpXkffNhopnxZHW
+Eg/4BgxPGWPXWXtZaIu0B8paAOh8xqeO1nGn0MfL49aLokrwkqn4GlP/UXo
|)))(9:propagate)(3:tag(1:*)))
(9:signature(4:hash4:sha131:|I3xYGkxijsCtpm9s3B/XlGYGUHM=
|)(10:public-key8:dsa-sha1
(7:KeySpec602:|MIIBtzCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9
EAMMeP4C2USZp
RV1AIlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fn
xqimFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCouuE
C/BYHPUCgYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJ
FnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImo
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoDgYQAAoGAMPvUPIjkK+JOc8+Ng63Mf3lirEZXtuwBgP6Y
3M0ix+eOwK9NoTZ3BU8XWbQ+WWsLRBcc+tV97G2IhWkYuWmohsf7de6Mj/wSiSYwX1jQreRwocYu
236YzO3XvciftArZQhHwecVZA1zTKrbuyC+NSuCIMhFriskW0kfHmgJ23yY=
|))67:|MCwCFCwmdgJXhatkAhlEGmAlHR+n37wwAhQcuE7lM766V/jk2iBgNRCQEIpB1w==
|))
```

Name and Donation Certificates

The following example of SPKI name certificates was copied from [SPKI98c]. It is shown in both advanced format and BASE64 encoded transfer format.

```
(cert
    (issuer
     (name (hash md5 |Txoz1GxK/uBvJbx3prIhEw==|) fred))
    (subject (hash md5 |Z5pxCD64YwgS1IY4Rh61oA==|))
    (not-after "2001-01-01_00:00:00"))
```

```
{KDQ6Y2VydCg2Omlzc3Vlcig0Om5hbWUoNDpoYXNoMzptZDUxNjpPGjPUbEr
+4G8lvHemsiETKTQ6ZnJlZCkpKDc6c3ViamVjdCg0Omhhc2gzOm1kNTE2Ome
acQg+uGMIEtSGOEYetaApKSg5Om5vdC1hZnRlcjE5OjIwMDEtMDEtMDFfMDA
6MDA6MDApKQ==}
```

The following example shows a full SPKI donation certificate, also called auto certificate because its issuer and subject are the same [SPKI98c]. It is presented in advanced format. This donation certificate is issued by Carl Ellison to himself, and attests that the keyholder of Carl's key is called Carl Ellison and lives in Baltimore. The certificate has expired on the 15th of April, 1998.

```
(sequence
    (public-key
     (rsa-pkcs1-md5
      (e #11#)
      (n
       |ALNdAXftavTBG2zHV7BEV59gntNlxtJYqfWIi2kTcFIgIPSjKlHleyi9s
       5dDcQbVNMzjRjF+z8TrICEn9Msy0vXB00WYRtw/7aH2WAZx+x8erOWR+yn
       1CTRLS/68IWB6Wc1x8hiPycMbiICAbSYjHC/ghq2mwCZO7VQXJENzYr45|
       )))
    (do hash md5)
    (cert
     (issuer (hash md5 |+gbUgUltGysNgewRwu/3hQ==|))
     (subject
      (keyholder (hash md5 |+gbUgUltGysNgewRwu/3hQ==|)))
     (tag
      (* set
       (name "Carl M. Ellison")
       (street "207 Grindall St.")
       (city "Baltimore MD")
       (zip "21230-4103")))
     (not-after "1998-04-15_00:00:00"))
    (signature
     (hash md5 |54LeOBILOUpskE5xRTSmmA==|)
     (hash md5 |+gbUgUltGysNgewRwu/3hQ==|)
     |HU6ptoaEd7v4rTKBiRrpJBqDKWX9fBfLY/MeHyJRryS8iA34+nixf+8Yh/
     buBin9xgcu1lIZ3Gu9UPLnu5bSbiJGDXwKlOuhTRG+lolZWHaAd5YnqmV9h
     Khws7UM4KoenAhfouKshc8Wgb3RmMepi6t80Arcc6vIuAF4PCP+zxc=|)
    )
   )
```