Adding SPKI Certificates to JDK 1.2

Jonna Partanen

jonna.partanen@hut.fi
Helsinki University of Technology

Pekka Nikander

pekka.nikander@hut.fi
Helsinki University of Technology

Abstract

The Java Development Kit (JDK) has included the concepts of cryptographic keys, signatures and certificates since version 1.0, and they have been improved and extended in JDK 1.2. However, the certificate interfaces still only cover identity certificates. As more and more security software makes use of authorization certificates, we feel that the concept of an authorization certificate and its implementation need to be added to the Java Security API.

In this paper, we analyze the certificate model of the JDK 1.2. We also describe an extension to the JDK 1.2 cryptography architecture, providing support for authorization certificates in general and SPKI certificates in particular.

In the future, we intend to use the extensions described in this paper to customize the JDK 1.2 policy management to be easier to distribute. In particular, we are going to replace the identity based security management that is configured through a configuration file, with a capability based alternative that uses SPKI certificates.

1 Introduction

The word "certificate" was first used in 1978 by Loren Kohnfelder to refer to a digitally signed record holding a name and a public key [5]. Since then, certificates were long considered to be a equivalent to identity certificates, serving only the purpose of cryptographically binding a globally unique name to a key and thus certifying an identity. In fact, many computer security professionals still use the words "certificate" and "identity certificate" as synonyms.

However, if we take a more general view to the matter, a certificate essentially is a digitally signed record that states some information about the entity the certificate was issued to. The entity may be a public key, a name, a person, a piece of software, or anything else as long as it can be stated in the certificate in a meaningful way. The information may be the entity's name, address, favourite colour, what the entity is allowed to do, or something else.

The Java runtime environment seems to be the first widely accepted architecture for running downloadable content. This is mostly because it has addressed the security issues from the very beginning. The core of the Java security is built on top of a set of cryptographic services such as digital signatures and certificates, provided by the cryptography API. However, this API also seems to be based on the misbelief that identity certificates are the only kind of certificates, or at least the only kind of practical use.

The rest of this paper is organized as follows. In the next section, we present the concept of authorization certificate and one concrete type of authorization certificates, the SPKI (Simple Public Key Infrastructure) certificates. In Section 3, we describe the Certificate class and other related classes of JDK 1.2. In Section 4, we show how the JDK Certificate class can be modified to be a superclass for authorization certificates as well as identity certificates, and how the JDK API could support SPKI certificates. Section 5 describes how we implemented SPKI certificates in JDK 1.2. Finally, in Section 6, we present our conclusions from this work.

2 Authorization Certificates

Authorization certificates, or signed credentials, are signed statements of authorization. In other words, whereas identity certificates bind a name to a cryptographic key, authorization certificates bind a capability to the key. The capability is then used directly to gain the access in question, instead of looking up the name in some access control list. Thus, an unnecessary step of indirection is removed from the access control procedure. Furthermore, as an authorization certificate does not necessarily need to carry any human understandable information about the identity of the subject, i.e., a name, authorization certificates can also be used in situations where anonymity is desired. The concept of authorization certificates was first independently described in the SDSI [11] and PolicyMaker [3] prototype systems and the SPKI [4] initiative.

2.1 SPKI Certificates

SPKI certificates are currently being standardized by the IETF. In the SPKI framework, all principals are keys. Delegations are made to a key, not to a keyholder as in identity certificates. Thus, SPKI certificates are conceptually closer to capabilities than to identity certificates. However, there is also a difference between capabilities and SPKI certificates. In traditional capability based systems, the capability is a secret ticket that grants authority to anyone possessing it, so the capabilities need to be strictly controlled. The SPKI certificate, however, only grants authority to the key specified in the certificate. Thus, it does not need to be treated as secret and to be strictly controlled.

Format. A SPKI certificate consists of several fields, the five most important being: issuer, subject, delegation, authorization and validity. They are often described as five-tuples <I, S, D, A, V>, where:

- **I** is the public key or a hash of the public key of the certificate's Issuer.
- **S** is the Subject of the certificate, typically a public key, a secure hash of a public key, or a secure hash of some other object. It may also be a name denoting a public key or a group of them.
- **D** is the delegation bit, indicating whether the subject has the right to further delegate the rights granted in this certificate.
- **A** is the authorization field (also called tag) that denotes the actual rights granted by the Issuer to the Subject. This field may also consist of other information.
- **V** is the validity field of the certificate. Typically, it is a time range, but it may also indicate "one-time" validity.

Let us consider an example. Alice has an account X in the bank B. The bank wants to grant Alice the on-line access to her account. Consequently, the bank creates a SPKI certificate <K_B , K_A , *Yes, access to 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 pay her bills while she is on vacation. However, she does not want Carl to be able to delegate the right to anyone else. To do this, Alice produces Carl the following certificate: <K_A , K_C , *No, access to account X for paying bills, the time of the vacation*>. These two certificates form a chain that delegate the access right to Carl. Carl, too, has to show that he holds the private key corresponding to K_C before the bank will accept the certificate chain and let him pay the bills.

Certificate reduction. Two SPKI certificates, <I_1 , S_1 , D_1 , A_1 , V_1 > and <I_2 , S_2 , D_2 , A_2 , V_2>, form a valid chain that can be reduced iff:

- $S_1 = I_2$
- D_1 = true

The certificate resulting from the reduction is <I, S, D, A, V>, where I is I_1, S is S_2, D is D_2, A is the intersection of A_1 and A_2, and V is the intersection of V_1 and V_2.

The forming of delegation chains and chain reduction of the corresponding certificates are key properties of SPKI. Reduction certificates can be used to improve chain reduction performance by shortening the chains to be verified. [4] [5]

Name certificates. In addition to the normal SPKI certificates that define authorization, the SPKI definition includes a form of identity certificates called name certificates. They bind names to keys, allowing late binding of symbols into keys [5] and easier management by humans. That is, the administrator can create a certificate by specifying the subject as a name instead of a key, and an other certificate binds the name to a key. The latter certificate does not have to exist at the time the first certificate is created, but can be created later on, if needed. If a certificate uses a name that has not yet been bound to a key, or the name certificate specifying the binding is not found, the certificate is considered invalid.

3 JDK 1.2 Cryptography Architecture

The Java Cryptography Architecture (JCA) has been built around the design principles of implementation independence and interoperability, and algorithm independence and extensibility. The aim is to let users of the API to use cryptographic concepts, such as digital signatures and message digests, without having to think about the implementations or even the algorithms being used to implement these concepts. [13]

The Java documentation explains that algorithm independence is achieved by defining types of cryptographic services called "engines", and defining classes that provide the functionality of these cryptographic engines. The `Signature` and `MessageDigest` classes are supposed to be examples of such an engine, but they both require that the algorithm is specified when the class is instantiated. Apparently the design principle of algorithm independence was too hard to follow in practise.

Implementation independence is achieved using a "provider"-based architecture. The term "provider" refers to a package or set of packages that implement one or more cryptographic services. The user may have several providers installed, and new providers can be installed dynamically. When the user needs a particular cryptographic 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 practise, this means that the architecture has several levels. The upmost level consists of general interfaces and abstract classes: `Certificate`, `Key`, `PublicKey`, `PrivateKey`, `Signature`, `MessageDigest` and so on. The second level consists of interfaces and abstract classes for different subtypes of the general concepts, specifying different algorithms. For example, the `Certificate` class is extended by `X509Certificate` class and the `PublicKey` interface is extended by `RSAPublicKey` and `DSAPublicKey` interfaces. Finally, the actual implementation is supplied by one or more providers that include the actual functionality. The default provider that comes with JDK and implements most of the concepts included in the architecture is named "SUN".

3.1 Certificates in JDK 1.2

The `java.security.cert.Certificate` class is an abstract superclass for identity certificates. It has several methods that are supposed to be common to all identity certificates, namely: equals, getEncoded, getPublicKey, getType, hashCode, toString, and two verify methods.

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 also includes an abstract class for this purpose.

The `Certificate`'s only subclass in JDK 1.2 is the `X509Certificate` class. It is also abstract and serves as the API for all X.509 certificate implementations. The related classes include the `X509CRL` class for certificate revocation lists and the `X509Extension` interface for the X.509v3 extensions.

Whereas all certificates have the getEncoded method that returns the transfer representation of the certificate, they do not include a method that would create a new certificate from this encoded format. This is the job of a `CertificateFactory`.

In JDK 1.2 the certificates play an important role in access control. Each class may have one or several certificates attached to it, and these certificates are used in defining what access rights the classes get. The certificates are given to the class in the class loading phase by the `JarVerifier`. The `JarVerifier` parses the jar file, verifies the signatures, and places the corresponding certificates to the class' set of certificates.

3.2 Certificate Storage

Certificates are generally long lived objects that are stored in files or databases and transfered between different computer systems. JDK provides a certificate storage functionality as a part of key management. In JDK, the `KeyStore` class represents a storage facility that can contain keys and certificates. The actual implementation can be chosen from some of the providers available, just as one would choose an implementation of a signature algorithm, for example. JDK includes one implementation of the KeyStore: the `JavaKeyStore` class that stores the objects in an encrypted file using a proprietary file format.

The KeyStore is also somewhat oriented towards identity certificates and specially X.509, but can fairly easily be used to store other types of certificates as well. Although the default implementation uses files, providing an implementation that uses a distributed database is equally possible.

4 Extending the Certificate Class

Identity certificates and authorization certificates have several common features. They are both signed records, so they have a signature. They also have a transfer representation, an encoded form common to all implementations. In addition, they 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. Like all objects, 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 is changed to getSubjectKey. Furthermore, since all certificates' subjects do not 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.

We also propose that, since most if not all certificate types have some kind of validity information, a validity check method should be added to the Certificate class. It could be named checkValidity according to the example set by the `X509Certificate` class, or its name could be simply isValid. The method would return a boolean value depending on whether the certificate is valid at the moment when the checking is done. If some certificate type, that we are unaware of, does not have any validity information at all, its implementation of this method should always return "true".

4.1 The Superclass for SPKI Certificates

Just as all X.509 certificate implementations in Java have a common, abstract superclass called `X509Certificate`, all the SPKI certificate implementations should have an abstract superclass that extends the `Certificate` class. We have called this class `SPKICertificate`. The superclasses for certificate implementations are shown in Figure 1.

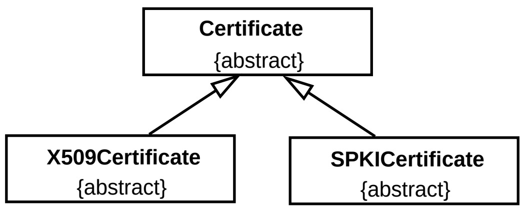

Figure 1: The certificate superclass hierarchy

The `SPKICertificate` specifies the methods that any SPKI certificate implementation must have (in addition to the methods specified by the `Certificate` class). These methods are derived from the SPKI specification according to

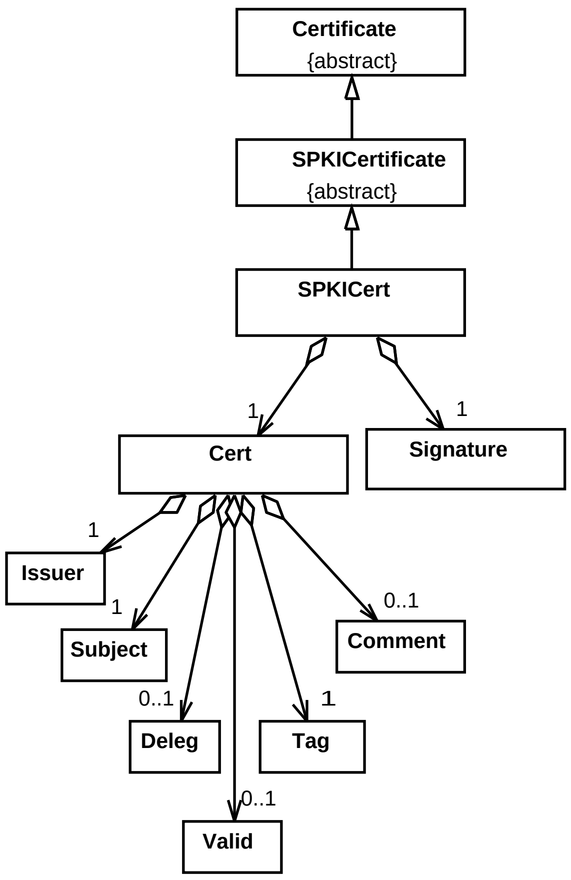

Figure 2: SPKI certificate object structure

the principle that there must be at least one access method for each field of the certificate: the method for viewing the field contents. We have named these methods getVersion, getIssuer, getSubject, getDelegation, getTag, getValidity, getComment, getIssuerInfo and getSubjectInfo.

5 Implementing SPKI Certificates

The core of our implementation are the classes `SPKICertificate` and `SPKICert`. The `SPKICertificate` is the abstract superclass for all SPKI certificates. The `SPKICert` class is our actual implementation. It is a subclass of the `SPKICertificate` class.

The simplified UML diagram of the SPKI certificate class hierarchy is shown in Figure 2. Three components of the `Cert` class have been left out of the diagram for clarity: these are the Version, the IssuerInfo and the SubjectInfo. They are optional, but recommended fields of the SPKI certificates.

The components of the `Cert` class are themselves complex hierarchies of relatively simple components. For example, according to the SPKI definition, the Tag object consists of a TagBody, which in turn can be either a TagStar-object or a TagExpression. The TagStar is used to denote any authorization, or "everything". The TagExpression is a recursive definition of a more restricted authorization, allowing the specification of capability groups, capability ranges and individual capabilities. An UML diagram of the Tag object hierarchy is shown in Figure 3. Again, the lowest levels of the hierarchy have been left out from the diagram.

Other classes in our prototype include the `SPKIProvider` and `SPKICertificateFactory`. They are described in the rest of this chapter.

5.1 Creating a Provider for SPKI

As we explained in Section 3, Java uses a class called `Provider` to find the classes implementing particular services. To register our implementation of the SPKI functionality, we must create a provider of our own. Our provider is called `SPKIProvider`. It extends the `java.security.Provider` class, and specifies the names of the classes implementing the functionality for handling SPKI certificates, namely, the `SPKICertificateFactory` class.

The `SPKICertificateFactory` is an engine class that is used to create `SPKICertificate` objects from the canonical s-expressions, the encoding format for SPKI certificates. It extends the `CertificateFactorySpi` class. A utility class, SPKIParserVisitor is used in pars-

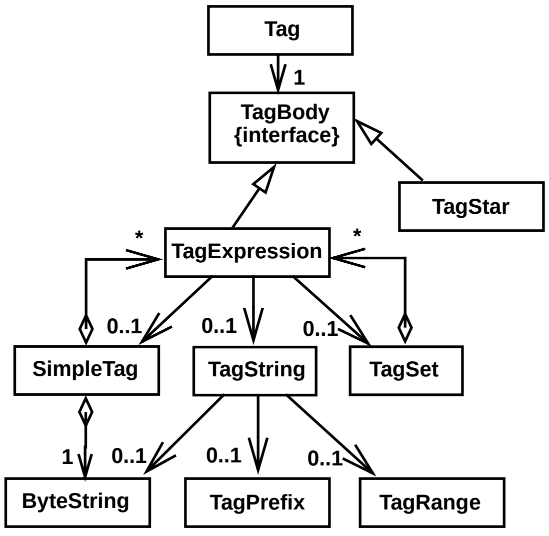

Figure 3: The Tag object structure

ing the canonical format into the SPKI object structure.

Because we are going to use the SPKI certificates to do fully distributed Java policy management, our provider should also include a distributed KeyStore implementation. At the time of this writing, this functionality is still under construction.

5.2 JarVerifier

As mentioned in Section 3, the classes in a jarfile may be signed. When such classes are loaded into the JVM, the `JarVerifier` verifies the signatures and if the check reveals no problems, decorates the loaded classes with corresponding X.509 certificates. These certificates in turn are used to determine what permissions the class should get.

This approach is suitable for the traditional access control that is based on identity certificates and ACLs. However, if we want to use authorization certificates for determining what access rights the classes have, the regular signatures could and should be replaced with authorization certificates to avoid unnecessary steps in class loading and access control.

Unfortunately, the `JarVerifier` has not been designed to be extendable. It seems to exist solely for the X.509 architecture, and even the signature algorithms that can be verified have been hard coded in the implementation. Thus we cannot even replace the standard signature algorithms of the Java library with some other algorithm, let alone replace them with a SPKI certificate, unless we replace the whole class. This is a major weakness in the otherwise relatively well designed and easily extendable architecture.

6 Conclusions

The basic JDK 1.2 cryptographic architecture is a fairly good starting point for adding new types of certificates. We see the goals of extendibility and implementation independence as worth pursuing, and have tried to further advance them in our design.

However, 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 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.

The `Certificate` class needs some minor changes to be suitable as a superclass for authorization certificates. The getPublicKey method has no intuitive meaning in authorization certificates, or at least the interpretation is not the same as in 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. 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.

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 straight forward.

In the future we are going to use the authorization certificate infrastructure created in this research to implement distributed access control management in JDK 1.2. [9]

References

[1] E. Amoroso, *Fundamentals of Computer Security Technology*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

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

[3] M. Blaze, J. Feigmenbaum, and J. Lacy, "Decentralized trust management", *Proceedings of the 1996 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 1996.

[4] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas and T. Ylönen, *Simple Public Key Certificate*, Internet-Draft `draft-ietf-spki-cert-structure-05.txt`, work in progress, Internet Engineering Task Force, March 1998.

[5] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas and T. Ylönen, *SPKI Certificate Theory*, Internet-Draft `draft-ietf-spki-cert-theory-02.txt`, work in progress, Internet Engineering Task Force, March 1998.

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

[7] I. Lehti, *SPKI-based Access Control Server*, Master's Thesis, Helsinki University of Technology, January 1998.

[8] I. Lehti and P. Nikander, "Certifying trust", *Proceedings of the Practice and Theory in Public Key Cryptography (PKC) '98*, Yokohama, Japan, Springer-Verlag, February 1998.

[9] P. Nikander and J. Partanen, "Distributed Policy Management for JDK 1.2", Proceedings of the 1999 Network and Distributed System Security Symposium, San Diego, CA, Internet Society, Reston, VA, February 1999.

[10] J. Partanen, *Using SPKI certificates for Access Control in Java 1.2,* Master's Thesis, Helsinki University of Technology, August 1998.

[11] R. L. Rivest and B. Lampson, "SDSI -- a simple distributed security infrastructure", *Proceedings of the 1996 Usenix Security Symposium*, 1996.

[12] *ITU-T Recommendation X.509 (1997 E): Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, ITU-T, June 1997.

[13] *Java™ Cryptography Architecture API Specification & Reference,* [on-line, referenced 28 July 1998], `http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html`, Sun Microsystems, June 1998.