Public Key Infrastructure
PKI Access Service

Confidential, Draft
RFC 29

15 Pages

Abstract

This RFC describes how services running on an OSGi compliant gateway can access a Public Key Infrastructure (PKI). Hence, this document specifies the interface a service can use to query a PKI, and it specifies how an object implementing this interface can be requested from the framework. This document is NOT concerned with specifying the logistics of setting up and maintaining a PKI.

0 Document Information

0.1 Table of Contents

0.2 Status

This document specifies an interface to access a Public Key Infrastructure for the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

All Page Within This Box

0.3 Acknowledgement

Preliminary versions of this document have been discussed at various SEG meetings. The input from the members of the SEG has greatly influenced this document.

Some text is taken over from the PKI RFP.

0.4 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.5 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Jun 18 2001	Initial draft based on discussions in the Leuven and Berlin meetings of the SEG. Frank Piessens, Security Expert Group member, OSGi.
2	Jul 10 2001	Revision based on comments from the SEG meeting in Copenhagen Frank Piessens, Security Expert Group member, OSGi

1 Introduction

A Public Key Infrastructure (PKI) enables a large number of security services. It is a supporting infrastructure making it possible to set up secure communication, to provide for non-repudiation of origin, to digitally sign bundles, etc…

Hence, it is expected that the PKI will be accessed by a number of different OSGi services. Therefore, it is important to standardize how the PKI can be accessed. The goal of this document is to specify a service for accessing the PKI.

All Page Within This Box

2 Motivation and Rationale

It is clear that a number of OSGi services will rely on a Public Key Infrastructure. Examples include a secure messaging service, a secure storage service, or a https-service. These services will need to be able to lookup and validate certificates. Access to the PKI could be arranged in a number of ways:

1. A first option is to let all services do their own thing. No standardized PKI Access service is defined, every service implements its own certificate lookup and validation code. The advantages of this option include: easy reuse of existing code since there is no requirement to code against a standard and no standardization effort. The main disadvantages are: every service developer relying on the PKI needs to either reinvent the wheel (i.e. reimplement the lookup and validation) or resort to code copying, and updating is more difficult (e.g. to support a new certificate format, every service using the PKI needs to be updated).

2. A second option is to disallow services to do their own PKI access, and force them to rely on a standardized PKI Access service. The disadvantages of this approach include: existing code has to be rewritten to the new standard, and it is difficult to enforce (verifying whether a service does its own validation and lookup of certificates requires manual code inspection). The advantages are: easy updating and easy support for different PKI standards (by having different implementations of the PKI Access service), and no need for different copies of the same code in different bundles.

3. Finally, a third option is a combination of the two above: allow services to implement their own PKI access (e.g. because an https implementation already exists), but standardize a PKI Access service that new service implementations can code against. The main advantage of this approach is its flexibility. It tries to get the best of both preceding options. The main disadvantage is that the possibility exists that the standard is ignored: since services are allowed to do their own thing, it is well possible that this third option in practice evolves in the first option.

In this document, the third option is followed: a PKI Access service is proposed that standardizes lookup and validation of certificates, but it is stipulated that this service should expose the standard Java cryptographic and certification path related packages, allowing existing implementations that run on the Java platform to be run without modifications. Exposing these packages from a single service reduces the amount of code that needs to be present in different bundles: even if a service implements its own PKI access, it does not need to put these packages in its bundle. Moreover the cryptographic packages have many applications other than validating certificates.

3 Technical Discussion

3.1 Introduction

To understand the purpose of the service proposed in this document, it is important to understand the context in which the service will be used. In this introduction, the different PKI-related software interfaces on a gateway are described.

The PKI-related interfaces on an OSGi gateway are structured as follows: four layers of abstraction, each resulting in some PKI-related API(s) are identified. Each subsequent layer builds on the functionality offered by the previous layers. In resource-constrained environments, some (parts) of these layers may be absent.

The four “layers” are not layers in the strictest sense of the word: their interaction can be summarized as follows (cfr. Figure 1): Layer 1 (the crypto layer) contains basic cryptographic primitives, and is separated in two sublayers. The services provided by layer 1 are used by layer 2, but may also be used by any service. Layer 2 (the certification path layer) uses layer 1 for signature checking, certificate parsing, etc… but may also use other API's for example the Java Naming and Directory Interface API. Layer 3 (the pki access layer) is a thin layer on top of layer 2. Layer 4 is not one monolithic layer, but consists of all the services using the PKI. A secure communication service for instance, will use the pki access layer to verify certificates, and will use the cryptographic primitives from the crypto layer to encrypt communicated data.

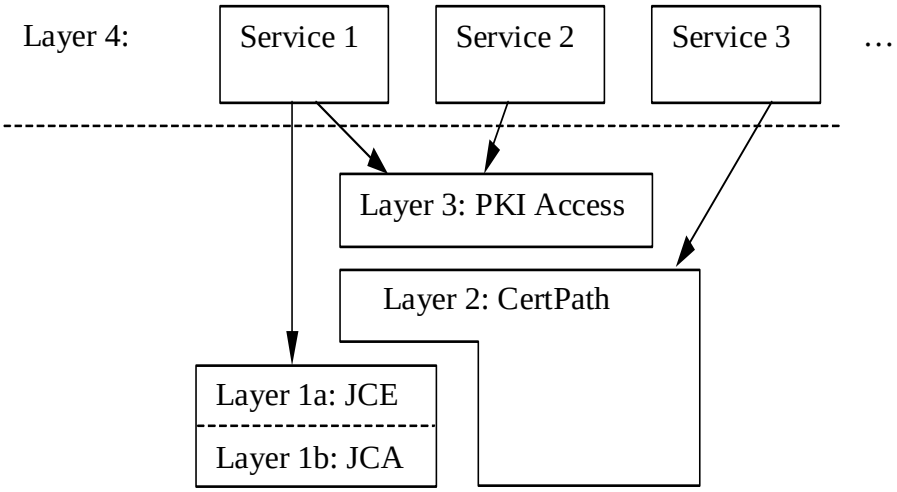

Figure 1 Structure of the PKI software API's

We shortly discuss each of the layers.

Layer 1: The Java Crypto Architecture and Extensions

The first layer consists of the JCA (by default present in the Java 2 platform) and the JCE (available as an installable extension to Java 2). The JCA must be present on an OSGi gateway offering the PKI Access service.

The JCE is optional and can be left out. If the JCE is not present, the PKI can only be used for digital signatures or integrity control, and not for encryption.

The JCA/JCE provides for the basic cryptographic building blocks, and elementary certificate operations. This includes, among others: secure random number generation, MAC's, digital signatures, symmetric and asymmetric encryption, key agreement, secure storage of keys and root certificates, and parsing of certificates.

The JCA/JCE API is documented by Sun. It is a provider based API, making it easy to put new functionality under the existing API. The API is documented in [2],[3].

Layer 2: The Certification Path API

The second layer consists of the certification path API standardized under the Java Community Process, and part of JDK 1.4.

The certification path layer is responsible for resolving trust issues (building and verifying certification paths to trust anchors), and for lookup of certificates. To achieve this, this layer uses the JCE/JCA API, e.g. to do signature verification. The draft certification path API documentation is available from the Java Community Process ([4]).

Layer 2 is optional and can be left out in a resource-constrained environment.

Layer 3: The PKI Access service

The PKI-Access layer is on OSGi service offering an extremely simple API to those services that want to use the PKI in a simple default way. It is this PKI Access service that is specified in this document.

On platforms that carry an implementation of the Certification Path API, layer 3 will be a thin façade over the Certification Path API: it will package a compatible CertStore, CertPathBuilder and CertValidator, and provide the necessary glue between these.

Layer 4: Services using the PKI

The highest layer contains all services that use the PKI to offer some higher-level service to application programmers. A typical example is the (secure) messaging service.

3.2 Design choices

In this section, we document a number of design decisions made during the development of this RFC. For each decision, we summarize the alternatives, and motivate why the choice was made as it is.

3.2.1 The Certificate class

The objects returned by, and verified by the PKI Access service (i.e. the objects representing certificates) could be typed in at least two ways:

1. A new interface OSGiCertificate could be defined, providing get-methods for all fields typically present in certificates used in the OSGi environment. The disadvantage of this approach is that existing java classes (e.g. X509Certificate) could not be reused, since they do not implement this new interface.

2. The existing java.security.cert.Certificate (abstract) class is used as the type. The disadvantage is that only very limited information about the contents of the certificate is available via the public methods of this class (in essence only the public key).

All Page Within This Box

The current design favors choice 2, because interoperability with existing Java code is considered important.

3.2.2 Engine classes versus interfaces

The JCA/JCE and CertPath API's are provider based API's allowing for multiple implementations of the API through the use of so-called engine classes. Most OSGi service API's on the other hand allow for multiple implementations through the use of interfaces.

An important design decision is which of these styles will be used for the PKI Access service: it relies on JCA/JCE and CertPath, so similarity to these API's would be reasonable. On the other hand, PKI Access is an OSGi service and hence should follow the OSGi conventions.

In the current design, the PKI Access service follows the OSGi style.

3.2.3 Naming

Certificates bind an entity to a public key. The entity is identified in the certificate through a name. An important design decision is how these names will be structured. Since the only existing naming scheme (federated addresses) are not very suitable as names for entities in certificates, the current design represents names just as Strings.

3.2.4 Certificate Filters

When looking up a certificate, the key search criterion is the name of the entity whose public key one is looking for. It may be useful to specify additional search criteria, e.g. in the case an entity has more than one key. Hence, the API for the methods that look up certificates could be structured in a number of ways:

- Just pass in a filter specifying all search criteria, including the name. This filter could be either an LDAP filter as used in the OSGi framework, or a custom-made filter.
- Pass in the name separately, and pass in a filter with additional search criteria as a second argument. This filter could again be either an LDAP filter or a custom-made filter.

In the current revision, the choice is made to pass in the name separately. The motivation for this choice is that the name parameter is a critical parameter from a security point of view, while the other search parameters are less critical. It does not make sense to do a wildcard search on the name, while it may make sense to do a wildcard search on the other criteria. Hence the name is kept as a separate parameter.

The choice for an LDAP filter or a custom-made filter is harder: the LDAP filter is more heavyweight, but is present in the framework anyway (it may still induce runtime inefficiencies though), a custom made filter can be made very light weight (just a properties-like object), but of course is PKI Access specific. In this revision, the custom made filter is still used.

3.3 API specification

3.3.1 org.osgi.nursery.pki Interface PKIAccess

public interface **PKIAccess**

PKIAccess is an interface for accessing the PKI. PKIAccess objects implementing different strategies for certificate lookup and validation can be requested from the framework.

All Page Within This Box

PKIAccess is typically a thin facade over the Certification Path API, making some sensible default choices for algorithms to do validation and lookup of certificates. The PKIAccess API is supposed to be used by applications requiring only default access to the PKI. Applications can have more expert-level access by calling the Certification Path API directly.

Keep in mind that calls to the specified methods may result in interactions with external entities. For example, a client may wish to establish whether it can trust a given `Certificate`. This may require that a certification path is constructed to a trust anchor. Construction of the certification path may, in turn, require retrieval of one or more certificates from external repositories. This is, however, hidden from the client - the client just calls `checkValidity(java.lang.String, java.security.cert.Certificate)`.

Version:
0.5

Method Summary

`void`	**`checkValidity`**`(java.lang.String name, java.security.cert.Certificate certificate)` Checks the validity of a certificate according to the strategy implemented by this PKIAccess object.
`void`	**`checkValidity`**`(java.lang.String name, java.security.cert.Certificate[] certificates)` Checks the validity of a chain of certificates according to the strategy implemented by this PKIAccess object.
`java.security.cert.Certificate[]`	**`getCertificates`**`(java.lang.String name)` Find certificates for the entity of the given name.
`java.security.cert.Certificate[]`	**`getCertificates`**`(java.lang.String name, CertificateFilter filter)` Find certificates for the entity of the given name, that satisfy the criteria specified in the CertificateFilter.

Method Detail

3.3.1.1 checkValidity

```
public void checkValidity(java.lang.String name,
                          java.security.cert.Certificate certificate)
                   throws java.security.cert.CertificateException
```

Checks the validity of a certificate according to the strategy implemented by this PKIAccess object.

Parameters:
`name` - the OSGi name of the owner of the certificate to be verified.
`certificate` - the `Certificate` to be verified.

Returns:
silently if the implementation can verify the validity of the given certificate. This method does a deep verify, including signature checking, certification path checking, expiration checking, revocation checking, ... The concrete strategy used for verification depends on the specific service object. The PKI Access service documents some aspects of its internal strategy by defining a number of service properties (that

All Page Within This Box

can be queried via a ServiceReference).
The name of the owner of the certificate is passed in as a parameter because the Java 2 abstract class `java.security.cert.Certificate` does not provide a method for getting the owner of a certificate. Without this name parameter the caller of this method would have no means to check if the certificate actually belongs to the named entity.
If the verification fails, a `CertificateException` is thrown indicating the cause of the failure. If the exception thrown is a `TransientCertificateException`, it might make sense to retry the validation again later.
If the PKI Access object does not recognise the certificate format of the passed certificate, it will throw an IllegalArgumentException.

3.3.1.2 checkValidity

```
public void checkValidity(java.lang.String name,
                          java.security.cert.Certificate[] certificate)
                   throws java.security.cert.CertificateException
```

Checks the validity of a chain of certificates according to the strategy implemented by this PKIAccess object.
This method allows for a push-model of certificate distribution: a service gets an appropriate certificate chain from its peer, and passes the entire chain to the PKI Access service. The PKI Access service can then validate the chain without having to lookup certificates.
Parameters:
`name` - the OSGi name of the owner of the certificate to be verified.
`certificates` - the certificate chain to be verified.
Returns:
silently if the implementation can verify the validity of the given certificate chain. This method does a deep verify, including signature checking, certification path checking, expiration checking, revocation checking, ... The concrete strategy used for verification depends on the specific service object. The PKI Access service documents some aspects of its internal strategy by defining a number of service properties (that can be queried via a ServiceReference).
The name of the owner of the certificate is passed in as a parameter because the Java 2 abstract class `java.security.cert.Certificate` does not provide a method for getting the owner of a certificate. Without this name parameter the caller of this method would have no means to check if the certificate actually belongs to the named entity.
If the verification fails, a `CertificateException` is thrown indicating the cause of the failure. If the exception thrown is a `TransientCertificateException`, it might make sense to retry the validation again later.
If the PKI Access object does not recognise the certificate format of the passed certificates, it will throw an IllegalArgumentException.

3.3.1.3 getCertificates

```
public java.security.cert.Certificate[] getCertificates(java.lang.String name,
                                                         CertificateFilter filter)
```

Find certificates for the entity of the given name, that satisfy the criteria specified in the CertificateFilter.
Parameters:
`name` - the OSGi name of the entity
`filter` - a CertificateFilter specifying the kind of certificate one is looking for. If this parameter is null, no filtering is done and all certificates found for the entity with the given name are returned.
Returns:

All Page Within This Box

an array of certificates for the named entity, satisfying the filtering criteria. There is no guarantee for completeness: the result represents a best effort search.
Returns null if no certificates can be found.

3.3.1.4 getCertificates

`public java.security.cert.Certificate[] getCertificates(java.lang.String name)`

Find certificates for the entity of the given name.
Equivalent to `getCertificates(name,null)`.

3.3.2 org.osgi.nursery.pki Class CertificateFilter

```
java.lang.Object
  |
  +-org.osgi.nursery.pki.CertificateFilter
```

---

public class **CertificateFilter**
extends java.lang.Object

CertificateFilter objects represent constraints on certificates. They are are used during lookup of certificates, to specify the kind of certificate one is looking for. Constraints can be put on the kind of algorithm the certificate key is usable for, the key strength, the key usage specifications, etc.
A number of frequently used filters could be defined as constants in this class. For instance, a STRONGRSA filter could look for certificates containing an RSA key of sufficiently large size.
Note: This class does not contain a `satisfiesFilter(Certificate)` method, because that would put the burden of knowing all diffferent certificate standards in this class. It seems more logical to put the knowledge about certificate standards under PKIAccess (because each PKIAccess object is typically made for one particular certificate standard). Hence, this class is just a simple Properties-like class.

Constructor Summary
CertificateFilter() Creates an empty filter.

Method Summary	
CertificateFilter	**addConstraint**(java.lang.String name, java.lang.Object value) Adds a constraint to the filter.
java.util.Map	**getConstraints**() Returns all the constraints in this filter as a Map.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

3.3.2.1 CertificateFilter

`public` **`CertificateFilter`**`()`

Creates an empty filter. An empty filter puts no constraints on certificates.

Method Detail

3.3.2.2 addConstraint

```
public CertificateFilter addConstraint(java.lang.String name,
                                       java.lang.Object value)
```

Adds a constraint to the filter. As a result, the filter becomes more stringent: only certificates passing the old filter, and the new constraint will pass the new filter.
Currently known names of constraints are "algorithm", "keyusage" and "keystrength".
For an "algorithm" constraint, the possible values are Strings like "RSA", "DSA", "ECDSA", ...
For a "keystrength" constraint, the possible values are Integers. The constraint specifies a minimal key strength (e.g. bitlength of the key).
For a "keyusage" constraint, the possible values are Strings like "DigitalSignature", "DataEncryption", "CodeSigning", "NonRepudiation", "EntityAuthentication", ...

Returns:
the modified CertificateFilter.

3.3.2.3 getConstraints

`public java.util.Map` **`getConstraints`**`()`

Returns all the constraints in this filter as a Map.

3.3.3 org.osgi.nursery.pki Class TransientCertificateException

```
java.lang.Object
  |
  +-java.lang.Throwable
        |
        +-java.lang.Exception
              |
              +-java.security.GeneralSecurityException
                    |
                    +-java.security.cert.CertificateException
                          |
                          +-org.osgi.nursery.pki.TransientCertificateException
```

All Implemented Interfaces:
java.io.Serializable

---

public class **TransientCertificateException**
extends java.security.cert.CertificateException

TransientCertificateExceptions represent a certificate validation exception that might be transient. Examples might be: a CRL distribution point that is unreachable, or an OCSP responder that does not

All Page Within This Box

reply within a specified timeout. If validation of a certificate throws a TransientCertificateException, it is sensible to try the validation again later.

See Also:

Serialized Form

Constructor Summary
`TransientCertificateException``(java.lang.String message)` Create a TransientCertificateException with the given message.

Methods inherited from class java.lang.Throwable
`fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

3.3.3.1 TransientCertificateException

`public` **`TransientCertificateException`**`(java.lang.String message)`

Create a TransientCertificateException with the given message.

3.4 Requesting a PKI Access object from the framework

PKI Access objects can range from quite simple (e.g. implementing a certificate store in a local file, employing a simple trust model and doing no revocation checking) to quite complex (e.g. implementing a powerful certificate lookup mechanism through LDAP, supporting a complex trust model and doing revocation checking based on CRL's).

When requesting a PKI Access service from the framework, it is important to be able to specify the kind of PKI Access service one requires. Therefore, implementations of the PKI Access service are required to specify the behavior of their implementation through a number of properties of the service. The following property keys should be defined:

- "certificateFormat", property indicating what format of certificates the PKI Access object understands (possible formats could be "X509v3", "WAP", …)
- "revocationChecking", indicating the revocation checking mechanism used (e.g. "none", "OCSP", "CRL",…) (see section 4.2 for recommendations on the choice of a revocation mechanism)
- "lookupMethod", indicating how the service looks up certificates (e.g. "LDAP", "localfile", "none", …)

Other property keys could be present, e.g. if the PKI Access service is implementing a standardized certificate validation algorithm (like the PKIX algorithm), a “standardName” property could be present.

4 Security Considerations

The PKI Access service is of major importance for the security of a gateway. To make sure that only secure PKI Access objects are made available, two aspects are of great importance: correct initial provisioning of the gateway, and the choice of a strong certificate validation algorithm (i.e. a good implementation of the checkValidity method).

4.1 Initial provisioning and the creation of the first PKI Access object

The very first PKI Access object on a gateway must rely on some trusted root certificate(s) to be able to do any certificate validation. Root certificates for later PKI Access objects could be validated through the first PKI Access object (e.g. because the two corresponding root CA’s have cross certified each other).

Hence, the creation of the first PKI Access object is dependent on a good initial provisioning of the gateway with a trusted root certificate. The initial provisioning scenario, and how the resulting trusted root certificates and the gateway’s private key(s) are made accessible to services (including the PKI Access service) will be discussed in a separate document.

4.2 Recommendations for certificate validation algorithms

The certificate validation algorithm is a security critical part of a gateway. Moreover, it is well known that designing and implementing a good certificate validation algorithm is a subtle and error-prone process. The validation algorithm depends on at least the following aspects:

1. the concrete certificate format, e.g. whether the certificate format supports constraints
2. the trust model
3. the risk associated with a validation error

In this section, some recommendations for implementers of a validation algorithm are given.

Use of standardized validation algorithms

For some standardized certificate formats, also standardized validation algorithms are published. For example for the PKIX profile X.509 certificates ([5]), a certificate path validation algorithm is specified. PKI Access objects that support such standardized certificate formats are strongly encouraged to also support the corresponding validation algorithm.

Risk dependent revocation checking

All Page Within This Box

Certificate revocation checking can be implemented in different ways, trading off communication overhead to risk. For low-risk to medium-risk applications, Certificate Revocation Lists are recommended. A CRL lists all revoked certificates, and must be published automatically and on a regular basis, i.e., when a certificate is revoked, the CRL can be automatically published to the Certificate Repository providing near-immediate availability. Experience has shown that correct handling of CRLs is a very complex and non-trivial matter.

For medium-risk to high-risk applications, it may be necessary to allow instant revocation of certificates. In this case, an online lookup from a trusted party can be performed to verify that a certificate is valid. Approaches for this online lookup include the Online Certificate Status Protocol (OCSP) and a trusted directory in which all non-revoked certificates are stored. When using the OCSP to obtain up to date Certificate Status Information, the entity that needs to validate a certificate relies on an OCSP-responder to produce up-to-date certificate status information. With a trusted directory, a certificate is invalid if it cannot be retrieved from the directory. These methods cause problems for systems that do not have constant access to the network, and may therefore be backed up with, e.g., the CRL solution.

For medium-risk to high-risk applications, a third method may be considered: if an entity that needs to validate another entity's certificate has no online connection, it may receive an OCSP-like response proving the certificate's validity, where the second entity forwards the OCSP-like response to the first entity having no online network connection. Using certificates with a short validity period for the second entity provides similar properties.

5 Document Support

5.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Knudsen, J., "Java Cryptography", O'Reilly.

[3]. http://java.sun.com/j2se/1.3/docs/guide/security/CryptoSpec.html

[4]. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_055_certp.html

[5]. Internet X.509 Public Key Infrastructure Certificate and CRL Profile, http://www.imc.org/draft-ietf-pkix-new-part1

5.2 Author's Address

Name	Frank Piessens
Company	Acunia

All Page Within This Box

Address	
Voice	
e-mail	Frank.Piessens@acunia.com

5.3 Acronyms and Abbreviations

- CA: Certification Authority
- CRL: Certificate Revocation List
- JCA: Java Cryptography Architecture
- JCE: Java Cryptography Extensions
- OCSP: Online Certificate Status Protocol
- PKI: Public Key Infrastructure

5.4 End of Document