OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
Mercurial
GitHub

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

ORACLE

JEP 452: Key Encapsulation Mechanism API

Owner	Weijun Wang
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	21
Component	security-libs / javax.crypto
Discussion	security dash dev at openjdk dot org
Effort	M
Duration	M
Reviewed by	Alan Bateman, Sean Mullan
Endorsed by	Sean Mullan
Created	2023/01/25 03:48
Updated	2024/04/23 15:08
Issue	8301034

Summary

Introduce an API for key encapsulation mechanisms (KEMs), an encryption technique for securing symmetric keys using public key cryptography.

Goals

- Enable applications to use KEM algorithms such as the RSA Key Encapsulation Mechanism (RSA-KEM), the Elliptic Curve Integrated Encryption Scheme (ECIES), and candidate KEM algorithms for the National Institute of Standards and Technology (NIST) Post-Quantum Cryptography standardization process.
- Enable the use of KEMs in higher level protocols such as Transport Level Security (TLS) and in cryptographic schemes such as Hybrid Public Key Encryption (HPKE, RFC 9180).
- Allow security providers to implement KEM algorithms in either Java code or native code.
- Include an implementation of the Diffie-Hellman KEM (DHKEM) defined in §4.1 of RFC 9180.

Non-Goals

- It is not a goal to include key pair generation in the KEM API. The existing `KeyPairGenerator` API is sufficient.
- It is not a goal to support the ISO 18033-2 defined encryption option for the encapsulate function.
- It is not a goal to support authenticated encapsulation and decapsulation functions as defined by RFC 9180.

Motivation

Key encapsulation is a modern cryptographic technique that secures symmetric keys using asymmetric or public key cryptography. The traditional technique for doing so is to encrypt a randomly generated symmetric key with a public key, but that requires padding and can be difficult to prove secure. A key encapsulation mechanism (KEM) instead uses properties of the public key to derive a related symmetric key, which requires no padding.

The concept of a KEM was introduced by Crammer and Shoup in §7.1 of *Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack*. Shoup later proposed it as an ISO standard in §3.1 of *A Proposal for an ISO Standard for Public Key Encryption*. It was accepted as ISO 18033-2 and published in May 2006.

KEMs are a building block of Hybrid Public Key Encryption (HPKE). The NIST Post-Quantum Cryptography (PQC) standardization process explicitly calls for KEMs and digital signature algorithms to be evaluated as candidates for the next generation of standard public key cryptography algorithms. The Diffie-Hellman key exchange step in TLS 1.3 can also be modeled as a KEM.

KEMs will be an important tool for defending against quantum attacks. None of the existing cryptographic APIs in the Java Platform is capable of representing KEMs in a natural way (see below). Implementors of third-party security providers have already expressed a need for a standard KEM API. It is time to add one to the Java Platform.

Description

A KEM consists of three functions:

- A *key pair generation function* that returns a key pair containing a public key and a private key.
- A *key encapsulation function*, called by the sender, that takes the receiver's public key and an encryption option; it returns a secret key *K* and a *key encapsulation message* (called *ciphertext* in ISO 18033-2). The sender sends the key encapsulation message to the receiver.
- A *key decapsulation function*, called by the receiver, that takes the receiver's private key and the received key encapsulation message; it returns the secret key *K*.

The key pair generation function is covered by the existing `KeyPairGenerator` API. We define a new class, `KEM`, for the encapsulation and decapsulation functions:

```
package javax.crypto;

public class DecapsulateException extends GeneralSecurityException;

public final class KEM {

    public static KEM getInstance(String alg)
        throws NoSuchAlgorithmException;
    public static KEM getInstance(String alg, Provider p)
        throws NoSuchAlgorithmException;
    public static KEM getInstance(String alg, String p)
        throws NoSuchAlgorithmException, NoSuchProviderException;

    public static final class Encapsulated {
        public Encapsulated(SecretKey key, byte[] encapsulation, byte[] params);
        public SecretKey key();
        public byte[] encapsulation();
        public byte[] params();
    }

    public static final class Encapsulator {
        String providerName();
        int secretSize();           // Size of the shared secret
        int encapsulationSize();    // Size of the key encapsulation message
        Encapsulated encapsulate();
        Encapsulated encapsulate(int from, int to, String algorithm);
    }

    public Encapsulator newEncapsulator(PublicKey pk)
            throws InvalidKeyException;
    public Encapsulator newEncapsulator(PublicKey pk, SecureRandom sr)
            throws InvalidKeyException;
    public Encapsulator newEncapsulator(PublicKey pk, AlgorithmParameterSpec spec,
                                        SecureRandom sr)
            throws InvalidAlgorithmParameterException, InvalidKeyException;

    public static final class Decapsulator {
        String providerName();
        int secretSize();           // Size of the shared secret
        int encapsulationSize();    // Size of the key encapsulation message
        SecretKey decapsulate(byte[] encapsulation) throws DecapsulateException;
        SecretKey decapsulate(byte[] encapsulation, int from, int to,
                              String algorithm)
                throws DecapsulateException;
    }

    public Decapsulator newDecapsulator(PrivateKey sk)
            throws InvalidKeyException;
    public Decapsulator newDecapsulator(PrivateKey sk, AlgorithmParameterSpec spec)
            throws InvalidAlgorithmParameterException, InvalidKeyException;

}
```

The `getInstance` methods create a new `KEM` object that implements the specified algorithm.

The sender calls one of the `newEncapsulator` methods. These methods take the receiver's public key and return an `Encapsulator` object. The sender can then call one of that object's two `encapsulate` methods to get an `Encapsulated` object, which contains a `SecretKey` and a key encapsulation message. The `encapsulate()` method returns a key containing the full shared secret, with an algorithm name of `"Generic"`. This key is usually passed to a key derivation function. The `encapsulate(from, to, algorithm)` method returns a key whose key material is a sub-array of the shared secret, with the given algorithm name.

The receiver calls one of the `newDecapsulator` methods. These methods take the receiver's private key and return a `Decapsulator` object. The receiver can then call one of that object's two `decapsulate` methods, which take the received key encapsulation message and return the shared secret. The `decapsulate(encapsulation)` method returns the full shared secret with a `"Generic"` algorithm, while the `decapsulate(encapsulation, from, to, algorithm)` method returns a key with the user-specified key material and algorithm.

A KEM algorithm can define an `AlgorithmParameterSpec` subclass to provide additional information to the full `newEncapsulator` method. This is especially useful if the same key can be used to derive shared secrets in different ways. Instances of an `AlgorithmParameterSpec` subclass should be immutable. If any of the information inside an `AlgorithmParameterSpec` object needs to be transmitted along with the key encapsulation message so that the receiver is able to create a matching decapsulator then it will be included as a byte array in the `params` field inside the `Encapsulated` result. In that case, the security provider should provide an `AlgorithmParameters` implementation using the same algorithm name as the KEM. The receiver can initiate such an `AlgorithmParameters` instance with the received `params` byte array and recover an `AlgorithmParameterSpec` object to be used when it calls the `newDecapsulator` method.

Multiple concurrent invocations of the `encapsulate` or `decapsulate` methods of a particular `Encapsulator` or `Decapsulator` object, respectively, should be safe. Each invocation of an `encapsulate` method should generate a new shared secret and encapsulation.

Here is an example using a hypothetical `"ABC"` KEM. Before the key encapsulation and decapsulation, the receiver generates an `"ABC"` key pair and publishes the public key.

```
// Receiver side
KeyPairGenerator g = KeyPairGenerator.getInstance("ABC");
KeyPair kp = g.generateKeyPair();
publishKey(kp.getPublic());

// Sender side
KEM kemS = KEM.getInstance("ABC-KEM");
PublicKey pkR = retrieveKey();
ABCKEMParameterSpec specS = new ABCKEMParameterSpec(...);
KEM.Encapsulator e = kemS.newEncapsulator(pkR, specS, null);
KEM.Encapsulated enc = e.encapsulate();
SecretKey secS = enc.key();
sendBytes(enc.encapsulation());
sendBytes(enc.params());

// Receiver side
byte[] em = receiveBytes();
byte[] params = receiveBytes();
KEM kemR = KEM.getInstance("ABC-KEM");
AlgorithmParameters algParams = AlgorithmParameters.getInstance("ABC-KEM");
algParams.init(params);
ABCKEMParameterSpec specR = algParams.getParameterSpec(ABCKEMParameterSpec.class);
KEM.Decapsulator d = kemR.newDecapsulator(kp.getPrivate(), specR);
SecretKey secR = d.decapsulate(em);

// secS and secR will be identical
```

KEM configurations

A single KEM algorithm can have multiple configurations. Each configuration can accept different types of public or private keys, use different methods to derive the shared secrets, and emit different key encapsulation messages. Each configuration should map to a specific algorithm that creates a fixed size shared secret and a fixed size key encapsulation message. The configuration should be unambiguously determined by three pieces of information:

- The algorithm name passed to a `getInstance` method,
- The type of the key passed to a `newEncapsulator` or `newDecapsulator` method, and
- The optional `AlgorithmParameterSpec` object passed to a `newEncapsulator` or `newDecapsulator` method.

For example, the Kyber family of KEMs could have a single algorithm named `"Kyber"`, but the implementation could support different configurations based on key types, e.g., Kyber-512, Kyber-768, and Kyber-1024.

Another example is the RSA-KEM family of KEMs. The algorithm name could simply be `"RSA-KEM"`, but the implementation could support different configurations based on different RSA key sizes and different key derivation function (KDF) settings. The different KDF settings could be conveyed via an `RSAKEMParameterSpec` object.

In both cases, the configuration can only be determined after one of the `newEncapsulator` or `newDecapsulator` methods is called.

Delayed provider selection

The provider chosen for a given KEM algorithm can depend not only upon the name of the algorithm passed to a `getInstance` method but also upon the key passed to a `newEncapsulator` or `newDecapsulator` method. The selection of the provider is thus delayed until one of those methods is called, just as in other cryptographic APIs such as `Cipher` and `KeyAgreement`.

Each call of a `newEncapsulator` or `newDecapsulator` method can select a different provider. You can discover which provider is selected via the `providerName()` methods of the `Encapsulator` and `Decapsulator` classes.

The `encapsulationSize()` methods

Some higher-level protocols concatenate key encapsulation messages with other data directly, without providing any length information. For example, Hybrid TLS Key Exchange concatenates two key encapsulation messages into a single key_exchange field, and RSA-KEM concatenates the key encapsulation message with the wrapped keying data. These protocols assume that the length of the key encapsulation message is fixed and well-known once the KEM configuration is fixed. We provide the `encapsulationSize()` methods to retrieve the size of the key encapsulation message in case an application needs to extract the key encapsulation message from such concatenated data.

Shared secrets might not be extractable

All existing KEM implementations return shared secrets in a byte array. However, a Java security provider might be backed by a native-code implementation and the shared secret might not be extractable. Therefore it is not always possible to return the shared secret in a byte array. For that reason, the `encapsulate` and `decapsulate` methods always return the shared secret in a `SecretKey` object.

If the key is extractable, the format of the key must be `"RAW"` and its `getEncoded()` method must return either the full shared secret or the slice of the shared secret specified by the `from` and `to` parameters of an extended `encapsulate` or `decapsulate` method.

If the key is not extractable, the key's `getFormat()` and `getEncoded()` methods must return `null` even though internally the key material is either the full shared secret or a slice of the shared secret.

The KEM service provider interface (SPI)

A KEM implementation must implement the `KEMSpi` interface:

```
package javax.crypto;

public interface KEMSpi {

    interface EncapsulatorSpi {
        int engineSecretSize();
        int engineEncapsulationSize();
        KEM.Encapsulated engineEncapsulate(int from, int to, String algorithm);
    }

    interface DecapsulatorSpi {
        int engineSecretSize();
        int engineEncapsulationSize();
        SecretKey engineDecapsulate(byte[] encapsulation, int from, int to,
                                    String algorithm)
                throws DecapsulateException;
    }

    EncapsulatorSpi engineNewEncapsulator(PublicKey pk, AlgorithmParameterSpec spec,
                                          SecureRandom sr)
            throws InvalidAlgorithmParameterException, InvalidKeyException;
    DecapsulatorSpi engineNewDecapsulator(PrivateKey sk, AlgorithmParameterSpec spec)
            throws InvalidAlgorithmParameterException, InvalidKeyException;

}
```

An implementation must implement the `EncapsulatorSpi` and `DecapsulatorSpi` interfaces, and return objects of these types from the `engineNewEncapsulator` and `engineNewDecapsulator` methods of its `KEMSpi` implementation. Calls to the `secretSize`, `encapsulationSize`, `encapsulate`, and `decapsulate` methods of `Encapsulator` and `Decapsulator` objects are delegated to the `engineSecretSize`, `engineEncapsulationSize`, `engineEncapsulate`, and `engineDecapsulate` methods in the `EncapsulatorSpi` and `DecapsulatorSpi` implementations.

An implementation of the `engineEncapsulate` and `engineDecapsulate` methods must be able to encapsulate or decapsulate keys with a `"Generic"` algorithm, a `from` value of 0, and a `to` value of the shared secret's length. Otherwise, it can throw an `UnsupportedOperationException` if the combination of arguments is not supported because, e.g., the algorithm name cannot be mapped to an internal key type, the size of the key does not match the algorithm, or the implementation does not support slicing the shared secret freely.

Future Work

Encryption options

ISO 18033-2 defines an *encryption option* for the encapsulate function because some asymmetric ciphers allow scheme-specific options to be passed to the encryption algorithm. However, this option is not mentioned in either RFC 9180 or NIST's PQC KEM API Notes, so we do not include it here. If a compelling case for an algorithm that requires this option arises then a future enhancement could introduce another overload of the `encapsulate` method that allows the inclusion of algorithm-specific parameters.

`AuthEncap` and `AuthDecap` functions

RFC 9180 defines two optional KEM functions, `AuthEncap` and `AuthDecap`, which allow the sender to provide its own private key during the encapsulation process so that the receiver can be assured that the shared secret was generated by the holder of that private key. However, these two functions do not appear in any other KEM definitions, so we do not include them here. Support for these functions could be added in a future enhancement.

Alternatives

Use existing APIs

We considered using the existing `KeyGenerator`, `KeyAgreement`, and `Cipher` APIs to represent KEMs, but each of them has significant issues. Either they don't support the required feature set, or the API does not match the KEM functions.

- A `KeyGenerator` is able to generate a `SecretKey`, but not the key encapsulation message at the same time. As a workaround, we could potentially encode both the shared secret and the key encapsulation message as the encoded form of the `SecretKey`. However, this only works when the shared secret is extractable and this is not always true, as discussed above. For keys that can be extracted, it still requires the application to extract the secret and the key encapsulation message from the encoded form of the `SecretKey`, which is complex and error-prone. Alternatively, we could store the key encapsulation message inside the `SecretKey` as a separate field. However, that would require a new `SecretKey` subclass that has a public method to retrieve the key encapsulation message.
- A `KeyAgreement` can return a key encapsulation message as a phase key and the shared secret, via different methods. However, a `KeyAgreement` object is meant to be initialized with the caller's own private key, but for a KEM there is no need to create a private key on the sender side. Also, the key encapsulation message of a KEM is defined as an opaque byte array but `KeyAgreement` returns the phase key as a `Key` object. New `KeyFactory` and `EncodedKeySpec` subclasses would be required to translate between key encapsulation messages and keys.
- A `Cipher` is able to wrap an existing key and then unwrap it. However, in a KEM the shared secret is generated by the encapsulation process. We could pass in a dummy or `null` key and store the actual shared secret in the output, but this has the same problem as `KeyGenerator`: It only works when the shared secret is extractable, and the application must extract the key and the key encapsulation message from the wrapped result. Moreover, wrapping a key and then unwrapping it should return the same key, but passing a dummy input to the wrap method does not conform to this convention.

In short, each of these alternatives would be a hack to work around an API that was not designed to represent a KEM. Extra classes and methods would be required, and the implementations would be complex and fragile. Without a standard KEM API, security providers are likely to implement KEMs in inconsistent and awkward ways which will be difficult for developers to use.

Include a key pair generation function

All KEM definitions contain a key pair generation function. We could have included such a function in the KEM API, but we chose not to do so since the existing `KeyPairGenerator` API was specifically designed for this purpose. Including an identical function in the KEM API could lead to confusion for provider implementors and for developers.

Testing

We will add conformance tests on input, output, and exceptions, and the DHKEM known-answer tests from RFC 9180.