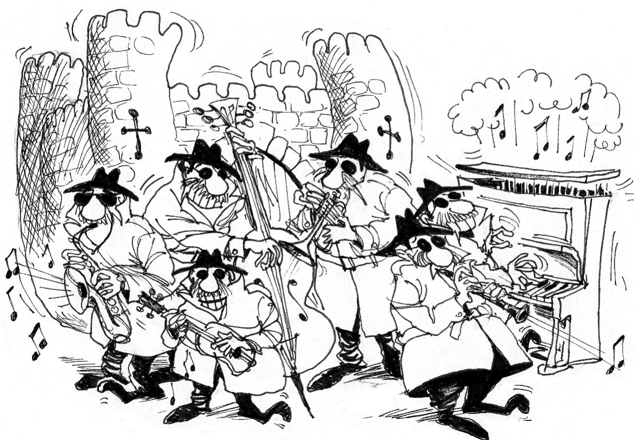
Legion of the Bouncy Castle Inc.
PQC-Addon

User Guide

Version: 1.0.1 Date: 05/04/22



Legion of the Bouncy Castle Inc.
(ABN 84 166 338 567)
https://www.bouncycastle.org

Copyright and Trademark Notice

This document is licensed under a Creative Commons Attribution 3.0 Unported License (http://creativecommons.org/licenses/by/3.0/)

Acknowledgements

Work on this project has been funded through support contract consulting hours made available from holders of Bouncy Castle support contracts with https://www.cryptoworkshop.com.

For further information about this distribution, or to help support this work further, please contact us at office@bouncycastle.org.

Table of Contents

1 Introduction....5
2 Installation....6
3 Using the PQC-Addon Package....7
3.1 An Example of NHOtherInfoGenerator....7
3.2 An Example of PQCOtherInfoGenerator....8
3.3 Using NHSecretKeyProcessor....9
3.4 Using PQCSecretKeyProcessor....10

1 Introduction

This document discusses an experimental API put together to allow a PQC key exchange and KEM to be piggy backed onto a regular key exchange using either KAS, or key transport using algorithms like RSA.

The PQC-addon is currently built on the New Hope key exchange algorithm and the round 3 NIST PQC candidate algorithms FrodoKEM and Classic McEliece. This is hidden in the API and public wise it allows for transforming an existing secret key using XOR, or it can be used to generate a KAS OtherInfo object where the supplementary private information is filled in using the result of the New Hope key exchange.

The addon is designed for “least harm” in the sense that assuming PQC algorithms really do stand the test of time, it will protect exchanged keys for well past the time the regular algorithms become obsolete. On the other hand, if it turns out to be flawed, the worst case security should be that associated with the original pre-quantum algorithms used.

The addon is the result of thought processes which began with a submission to NIST which can be found at: https://csrc.nist.gov/CSRC/media/Publications/nistir/8105/final/documents/nistir-8105-public-comments-mar2016.pdf when we at BC were looking for ways of “filling in the gap” that we seem to be currently in.

2 Installation

The bcpqc-addon jar can be installed in either jre/lib/ext for your JRE/JDK, or in many cases, on the general class path defined for the application you are running. While the jar does make use of some classes in the published BC-FJA API, it does not need to be loaded by any system class loaders and does its work in application space.

3 Using the PQC-Addon Package

The PQC add on has 5 public classes and two interfaces. The five public classes are NHOtherInfoGenerator, NHSecretKeyProcessor, PQCParameters, PQCOtherInfoGenerator, and PQCSecretKeyProcessor. The first two of these classes make use of the New Hope key exchange algorithm internally so there are a couple of steps that must be performed by both parties to the original secret key exchange to allow the process to work. The second two classes currently make use of the Round 3 candidate algorithms FrodoKEM and Classic McEliece.

3.1 An Example of NHOtherInfoGenerator

For the most part, the NHOtherInfoGenerator looks like a regular OtherInfoGenerator in line with what is described in NIST SP 800-56A. The initial parameters provide distinguisher input for the IV to be used with the eventual KDF invocation that specify the base algorithm for the KDF and the identity of party U and party V – the two parties taking part in the agreement process.

Where the parameters diverge is on the last one, the `SecureRandom` which is needed to provide the entropy for the New Hope key exchange. This makes up for the lack of a pre-shared secret at the start of the KAS agreement which the New Hope algorithm is used to generate.

The process of arriving at an OtherInfo object for use as user-keying material with the KAS algorithm being used takes 4 steps and involves the exchange of information that can be public between the two parties.

The first step is for party U to generate partA – which is effectively a New Hope public key. This is done as follows:

```
SecureRandom random = new SecureRandom();
NHOtherInfoGenerator.PartyU partyU = new NHOtherInfoGenerator.PartyU(
                  new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
                              Hex.decode("beef"), Hex.decode("cafe"), random);

byte[] partA = partyU.getSuppPrivInfoPartA();
```

The second step is for party V to create their own generator and take the initial partA data to generate a response (partB) to send back to party U.

```
SecureRandom random = new SecureRandom();
NHOtherInfoGenerator.PartyV partyV = new NHOtherInfoGenerator.PartyV(
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
                        Hex.decode("beef"), Hex.decode("cafe"), random);

byte[] partB = partyV.getSuppPrivInfoPartB(partA);
```

The third step is for party U to generate the final OtherInfo object by using partB as follows.

```
DEROtherInfo otherInfoU = partyU.generate(partB);
```

At this point all that is required is for party V to use their generator to complete the process.

```
DEROtherInfo otherInfoV = partyV.generate();
```

The encoding of the OtherInfo objects can then be passed as user keying material. Assuming keypairs are already available for ECCDH for party U (kpU) and party V (kpV) for a simple case this can be done using the following for party U:

```
KeyAgreement agreementU = KeyAgreement.getInstance(
                                             "ECCDHwithSHA256KDF", "BCFIPS");
agreementU.init(
       kpU.getPrivate(), new UserKeyingMaterialSpec(otherInfoU.getEncoded()));

agreementU.doPhase(kpV.getPublic(), true);

byte[] secretU = agreementU.generateSecret();
```

and the following for party V:

```
KeyAgreement agreementV = KeyAgreement.getInstance(
                                            "ECCDHwithSHA256KDF", "BCFIPS");
agreementV.init(
      kpV.getPrivate(), new UserKeyingMaterialSpec(otherInfoV.getEncoded()));

agreementV.doPhase(kpU.getPublic(), true);

byte[] secretV = agreementV.generateSecret();
```

`secretU` and `secretV`, or whatever variant of the `generateSecret()` method you use, will be the same value, but in addition to the regular KAS shared secret will also have the post-quantum New Hope shared secret mixed into them.

Note that the classes from the `NHOtherInfoGenerator` are single use.

3.2 An Example of PQCOtherInfoGenerator

The PQCOtherInforGenerator is similar to the NHOtherInfoGenerator except it requires the passing in of the PQC KEM algorithm to use. Also, as the KEM algorithms dynamically generate secrets internally and the public keys involved are not single use the class can be reused.

The process of arriving at an OtherInfo object for use as user-keying material with the KAS algorithm being used takes 4 steps and involves the exchange of information that can be public between the two parties.

The first step is for party U to generate partA – which is the KEM public key. For FrodoKEM, this is done as follows:

```
SecureRandom random = new SecureRandom();
PQCOtherInfoGenerator.PartyU partyU = new PQCOtherInfoGenerator.PartyU(
                  PQCParameters.mceliece348864fr,
                  new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
                              Hex.decode("beef"), Hex.decode("cafe"), random);

byte[] partA = partyU.getSuppPrivInfoPartA();
```

The second step is for party V to create their own generator and take the initial partA data to generate a response (partB) to send back to party U.

```
SecureRandom random = new SecureRandom();
PQCOtherInfoGenerator.PartyV partyV = new PQCOtherInfoGenerator.PartyV(
            PQCParameters.mceliece348864fr,
```

```
            new AlgorithmIdentifier(NISTObjectIdentifiers.id_sha256),
                        Hex.decode("beef"), Hex.decode("cafe"), random);

byte[] partB = partyV.getSuppPrivInfoPartB(partA);
```

The third step is for party U to generate the final OtherInfo object by using partB as follows.

```
DEROtherInfo otherInfoU = partyU.generate(partB);
```

At this point all that is required is for party V to use their generator to complete the process.

```
DEROtherInfo otherInfoV = partyV.generate();
```

The encoding of the OtherInfo objects can then be passed as user keying material. Assuming keypairs are already available for ECCDH for party U (kpU) and party V (kpV) for a simple case this can be done using the following for party U:

```
KeyAgreement agreementU = KeyAgreement.getInstance(
                                             "ECCDHwithSHA256KDF", "BCFIPS");
agreementU.init(
       kpU.getPrivate(), new UserKeyingMaterialSpec(otherInfoU.getEncoded()));

agreementU.doPhase(kpV.getPublic(), true);

byte[] secretU = agreementU.generateSecret();
```

and the following for party V:

```
KeyAgreement agreementV = KeyAgreement.getInstance(
                                            "ECCDHwithSHA256KDF", "BCFIPS");
agreementV.init(
      kpV.getPrivate(), new UserKeyingMaterialSpec(otherInfoV.getEncoded()));

agreementV.doPhase(kpU.getPublic(), true);

byte[] secretV = agreementV.generateSecret();
```

`secretU` and `secretV`, or whatever variant of the `generateSecret()` method you use, will be the same value, but in addition to the regular KAS shared secret will also have the post-quantum KEM shared secret mixed into them.

3.3 Using NHSecretKeyProcessor

There are not many situations that you cannot deal with by just making use of a KDF and using whatever secret key you have as the seed for it. That said, there is another way to make use of the key exchange, which can be simpler as well.

The concepts behind the NHSecretKeyProcessor: making use of the U ⊕ V operation in SP 80-133, which is also discussed in FIPS IG 7.8 concerning post processing of secret keys. The long and the short of it is that it is safe to XOR a value with an existing secret key. You might do this in real life between two parties where you are also treating the value as a shared secret. In any case, the NHSecretKeyProcessor takes the New Hope key exchange value and applies it to a SHAKE-256 operator which is then used to generate bytes to XOR with a passed in secret key, producing a new secret key.

As with the OtherInfo generator, setting up a secret key processor is a four step process.

In the first step, party U needs to generate the partA component to send to party V.

```
SecureRandom random = new SecureRandom();
NHSecretKeyProcessor.PartyUBuilder partyU = new NHSecretKeyProcessor
                                                        .PartyUBuilder(random);
byte[] partA = partyU.getPartA();
```

In the second step, party V takes the data sent by party U and applies the data to its own builder, producing partB.

```
SecureRandom random = new SecureRandom();
NHSecretKeyProcessor.PartyVBuilder partyV = new NHSecretKeyProcessor
                                                        .PartyVBuilder(random);
byte[] partB = partyV.getPartB(partA);
```

PartB is then sent back to party U, who uses the data to construct their own version of the `SecretKeyProcessor`.

```
SecretKeyProcessor uProcessor = partyU.build(partB);
```

Finally party V calls its own builder and both parties are now in sync.

```
SecretKeyProcessor vProcessor = partyV.build();
```

Assuming a common `SecretKey`, `agreedKey`, has been arrived at by other means, the `SecretKeyProcessor` can be applied to produce a new key by passing the `agreedKey` to the `processKey()` method. For both party U, as in:

```
SecretKey uKey = uProcessor.processKey(agreedKey);
```

and party V:

```
SecretKey vKey = vProcessor.processKey(agreedKey);
```

Note that the classes from the `NHSecretKeyProcessor` are single use.

3.4 Using PQCSecretKeyProcessor

The PQCSecretKeyProcessor is a drop in replacement for the NHSecretKeyProcessor – the only extra requirement being the need to provide a KEMParameters object from the PQCParameters class to describe which KEM is in use.

As the KEM algorithms supported make use of re-usable public keys the PartyU class can be reused and does not need to be recreated for each use.