Don't Get Entangled in PQC: Embrace Crypto Agility for Smooth Code Migration!

one month ago

This blog is written by the participants of the PCSI PQC Benchmarking project, from TNO, Achmea, Belastingdienst, ABN Amro and ING. The test case described in this blog was done by ABN AMRO and TNO.

Introduction

In the coming years, many vulnerable cryptographic algorithms used in software will need to be replaced by quantum-safe alternatives to be safe against attacks enabled by quantum computers. Since all software migrations crucially depend on developers, they are the enablers of a future-proof cryptographic infrastructure that will ensure the confidentiality, integrity and availability of data. In this blog, we provide insights from one of the first post-quantum migrations of a commercial cryptographic software system that will be valuable to developers of other cryptographic applications when facing the upcoming challenge of post-quantum migration.

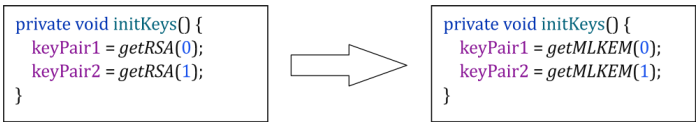
The software we looked at builds encrypted channels as an architecture for secured communication between endpoints, aiming at identification, confidentiality and untraceability through cryptographic mechanisms. This software has many modules with many lines of code – typical for production-grade applications - and was built in-house. Since the cryptography was not resistant to quantum attackers, we set the ambitious goal to migrate the system to a quantum-secure state and got to work. So, developers, if you want to learn all about our journey and obtain concrete advise for your own post-quantum migration, read on!

The Migration Journey

The system we were migrating has two options to establish secure channels. Either a Diffie-Hellman key exchange is used to establish a symmetric key, or a Key Encapsulation Mechanism (KEM)-based approach is taken. Diffie-Hellman key exchange is generally the most efficient approach, but there is currently no efficient post-quantum alternative, which means we could only implement KEM based solutions.

Since post-quantum KEMs have been standardized by NIST, the most straightforward approach was to replace the classical KEMs used in the KEM-based approach by post-quantum KEMs. The system in question is designed to ad-hoc negotiate an algorithm to be used between endpoints when establishing secure channels. Since there is no single best KEM for every use-case, and algorithms can be negotiated at run-time, we focused on three post-quantum KEMs: **FrodoKEM**, **McEliece**, and **ML-KEM**.

Our approach was simple – find all current references to RSA and Elliptic Curve Cryptography (ECC) and add options for our three alternatives. Sounds easy enough, how hard can it be? Let's see how it went.



In this code example, we replace calls to RSA by calls to ML-KEM.

Step 1: Add Cryptographic Logic

The first step was to navigate to the crypto core where the cryptographic logic of our system is defined. We needed to add some logic there for the different types of KEMs so that our APIs could work with them. Since this code base itself depends on an external library for the cryptographic operations, we needed to search for a version of that library that had implemented the correct types and use that as a security provider. Luckily this library (called Bouncy Castle) was already available and a few calls to that library and some extra tests later, we were done.

Step 2: Static Code Analysis

The next step was to figure out what parts of the code base depend on the specifics of the RSA logic, so we can make changes there. Nowadays, most IDEs support static code analysis, so that is what we employed to find these references, but third-party tooling can be used as well. It is no overstatement when we say that our static analysis results opened the box of pandora. It turned out that cryptographic dependencies were deeply embedded in the entire code base. Updating the code base to support one extra algorithm, required alterations in hundreds of files, many of which required manual labor.

From:

```
public final class AsymmetricDecryption extends AbstractDecryption {
   private static final String ENCRYPTION_ALGORITHM =
"RSA/NONE/OAEPWithSHA256AndMGF1Padding";
   private static final String PROVIDER_NAME = AsymmetricEncryption.PROVIDER_NAME;
   private static final AlgorithmParameterSpec PARAMETER_SPEC = new
OAEPParameterSpec("SHA-256", "MGF1", MGF1ParameterSpec.SHA256,
PSource.PSpecified.DEFAULT);
```

To:

```
public final class AsymmetricDecryption extends AbstractDecryption {
   private static final String ENCRYPTION_ALGORITHM =
AsymmetricEncryption.ENCRYPTION_ALGORITHM;
   private static final String PROVIDER_NAME = AsymmetricEncryption.PROVIDER_NAME;
   private static final AlgorithmParameterSpec PARAMETER_SPEC =
AsymmetricEncryption.PARAMETER_SPEC;
```

This is time-consuming work, especially if documentation is lacking or the original developers could not be involved anymore. Unfortunately, both were the case for our system. That brings us to a key insight of this process.

> ### Crypto agility is key in cryptographic code migration

It was clear that the code structure needed to be changed to make it more flexible, so it would be easier to add new algorithms. An interesting observation was that the code adhered to **best practices** from a software architecture perspective. Limited extra algorithms were (and still are) envisioned when the application was written, so the most efficient way to build the application at that time was to hardcode certain elements, like database layouts and test setups. That brings us to the second insight.

> ### Good quality code is not always crypto agile

Crypto agility requires us to approach the code architecture from a different point of view, namely finding the most suitable code structure such that cryptographic algorithms can be added and removed in the least labor-intensive way.

Step 3: Rewrite the Code Structure

We hoped to be done by now, but it was clear from the many hardcoded references that a thorough overhaul was necessary. Explicit references to crypto algorithms needed to be abstracted away from the business functionality of the system and every module of the system would have to adhere to this abstract encrypting and decrypting flow.

We followed five code concepts that helped boost the crypto agility of the code base, which we dive into further in the last section of this blog.

After we implemented these, adding new algorithms was a piece of cake. This is not only ideal when you want to try out various different post-quantum algorithms for your environment but immediately results in a future-proof code base as well. If one of the newly standardized post-quantum

schemes is found to have a vulnerability, then a new migration will be required as soon as possible.

A crypto agile code base is therefore more capable of coping with unexpected future migrations, which brings us to the final key insight.

> Crypto agility can speed up your migration

Step 4: Celebrate

Our code base is now better equipped to handle future cryptographic migrations. Most code bases are not inherently cryptographically agile, so some investment needs to go into this process. In our migration, we changed 293 Java files and adjusted 3918 lines of code. This is quite significant compared to the total 2227 Java files and 164k lines of code in the project.

Boosting Crypto Agility

We present five coding concepts that were essential in improving the cryptographic agility of the code base we migrated. Each concepts shows a snippet of code from our code base that relates to the concept.

1. Use Universal APIs

Ideally, the cryptographic core makes the connection between algorithm identifiers and cryptographic logic. APIs that require parameters related to specific algorithms, such as RSA or ECC, should be rewritten to be applicable for any cryptographic algorithm. Config files can be used to set cryptographic parameters, such that the code base can remain agnostic.

```
public AsymmetricKeysGenerator get(AsymmetricKeyType keyType) {

  return switch (keyType) {
    case EC -> new AsymmetricGenericKeysGenerator("EC", "BC");
    case RSA -> new AsymmetricGenericKeysGenerator("RSA", "BC");
    case MLKEM -> new AsymmetricGenericKeysGenerator("KYBER512", "BCPQC");
    case FRODO -> new AsymmetricGenericKeysGenerator("Frodo", "BCPQC");
    case MCELIECE -> new AsymmetricGenericKeysGenerator("CMCE", "BCPQC");
  };
}
```

This code example shows a universal API to retrieve a key generation object in Java. The body of the function handles all the specific calls for each algorithm.

2. Use Dynamic Memory Structures

Another example of crypto agile code is the following. If variables need to be tracked with references to specific algorithms, make sure that these are not stored in (top-level) variables, since that would require manual addition and removal of variables with every algorithm change. Instead, dynamic memory structures, such as dictionaries/maps, can be used to store variables per algorithm, using an algorithm identifier as the key. This also allows other modules to reference these objects in an algorithm-agnostic way.

```
private Map<AsymmetricKeyType, EncryptionKeys> encryptionKeysMap;
```

This code example shows a dynamic memory structure in Java, called a Map. It maps a key type to an object that records keys of that type.

```
assertThat(returnedObject.encryptionKeysMap.get(keyType),
is(equalTo(expectedObject.encryptionKeysMap.get(keyType))));
```

Code snippet showing the usage of the map in a test to assert equivalence. This test is now algorithm agnostic and can be parametrized, such that the test is run for each algorithm without having to duplicate the testing code.

3. Use Parameterized Tests

The best type of unit test tests behavior of the system under certain conditions. Instead of writing specific tests for each different algorithm, it is best to write one parametrized test that tests the behavior of the system. The addition of a new algorithm would then automatically add an extra test for that algorithm.

```
public static Stream<Arguments> testGenerateKeyPair() {
 return Arrays.stream(AsymmetricKeyType.values()).map((x) -> arguments(x));
}

@ParameterizedTest
@MethodSource
public void testGenerateKeyPair(AsymmetricKeyType keyType) {
 // test code body

}
```

In this code example, the arguments to the testing function testGenerateKeyPair are automatically extracted from the set of asymmetric key types. This only works if the test code body can reference dynamic memory structures to retrieve relevant objects for the respective algorithm, which is explained in the previous section.

4. Use An Appropriate Key Store

Most cryptographic applications store cryptographic keys in memory. These are usually separate applications, such as database applications, that expect a certain data layout. If this data layout is hardcoded to work for RSA or ECC, then it will most likely break if a key for a different algorithm is provided. It is therefore essential that the right key storage configurations are made to support the storage of various different types of keys. This could mean that database layouts need to be changed such that an arbitrary number of different types of keys can be stored. For example, if a database were used for RSA and ECC keys, then two changes need to happen. The hardcoded variables need to be replaced by dynamic memory storage, like a map in Java, and the annotations need to be updated to properly store the map.

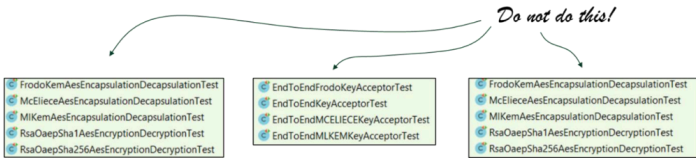
```
@ElementCollection
@CollectionTable(name = “example_table_name”, joinColumns = @JoinColumn(name =
“example_id”))
@MapKeyColumn(name = "key_type")
private Map<AsymmetricKeyType, @NotNull @Valid EncryptionKeys> encryptionKeysMap;
```

In this code example, Jakarta annotations are shown for a Map in Java, which handle the database layout to store the keys.

As a final note, depending on the cryptographic policy of a company, the size of symmetric keys might also need to be updated, which in turn needs to be supported by the key store if session keys are saved in long-term memory. All the more reason to have a crypto agile key store!

5. Use A Modular Class Structure

The project structure has a big influence on the crypto agility of the code base. If many different concepts require separate classes or files for each algorithm, then these classes need to be manually added upon each algorithm change. It is therefore best to write classes that flexibly reference different algorithms. Usually, the adoption of the first four concepts also leads to a better project structure.



This overview shows different Java files. Since there are files for each algorithm, the structure is not crypto agile.

Conclusion

During our journey, we successfully implemented post-quantum algorithms in our software application. However, the conclusion is not only that this was feasible, but more importantly, that we should also refactor the code in such a way to make future changes easier.

The migration of cryptographic algorithms in code can be severely sped up by integrating cryptographic agility into your code base. Unfortunately, most high-quality code is not necessarily crypto agile. It will take some time investment to get it done, but our five coding principles should give some guidance in how to effectively make this adjustment. Once your code is crypto agile, the effect of missing documentation or expertise is minimized and should enable a smooth transition for any algorithm migration to come.

Keep a look out for the other blogs in this series! In this series of four we share both organizational and technical results, from four different perspectives: Management, Vendor Management, Architects and Developers. Read the first blog here.

Disclaimer: Image created using Copilot

Alleen door samenwerking kunnen we de beste resultaten behalen in de strijd tegen cybercriminaliteit

PCSI is een samenwerking van

ABN·AMRO achmea Belastingdienst ING TNO