Alle nieuws

Voorkom post-quantum verstrengeling: Met cryptografische wendbaarheid is je code zo gemigreerd!

3 months ago

Deze blog is geschreven door de deelnemers van het PCSI PQC Benchmarkingproject: TNO, Achmea, Belastingdienst, ABN AMRO en ING. De casus die in deze blog wordt beschreven, is uitgevoerd door ABN AMRO en TNO.

Introductie

In de komende jaren zullen veel kwetsbare cryptografische algoritmes die in software worden gebruikt, vervangen moeten worden door quantumveilige alternatieven zodat ze bestand zijn tegen aanvallen van quantumcomputers. Aangezien alle softwaremigraties in hoge mate afhankelijk zijn van ontwikkelaars, zijn zij de sleutel tot een toekomstbestendige cryptografische infrastructuur waarbij vertrouwelijkheid, integriteit en beschikbaarheid van gegevens gewaarborgd blijft. In deze blog delen we inzichten uit een van de eerste post-quantum

migraties van een commercieel cryptografisch softwaresysteem. Deze inzichten zijn waardevol voor ontwikkelaars van andere cryptografische toepassingen die ook voor de uitdaging van post-quantum migratie staan.

De software die we hebben onderzocht, maakt beveiligde communicatie mogelijk tussen twee punten via versleutelende kanalen en richt zich op identificatie, vertrouwelijkheid en ontraceerbaarheid via cryptografische mechanismen. Deze software bevat een groot aantal modules met duizenden regels code, wat representatief is voor productieklare toepassingen, en is intern ontwikkeld. Omdat de cryptografie niet bestand was tegen quantumaanvallers, stelden we het ambitieuze doel om het systeem quantumveilig te maken en gingen we aan de slag. Dus, ontwikkelaars: als je alles wilt leren over ons avontuur en concrete adviezen wilt krijgen voor je eigen post-quantum migratie, lees dan verder!

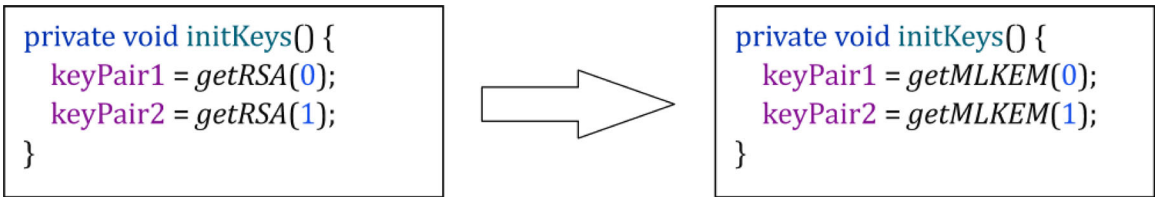
Ons migratieavontuur

Het systeem dat we hebben gemigreerd, heeft twee opties om beveiligde kanalen op te zetten: ofwel via een Diffie-Hellman-

sleuteluitwisseling om een symmetrische sleutel op te zetten, ofwel via een Key Encapsulation Mechanism (KEM). Diffie-Hellman is doorgaans de meest efficiënte aanpak, maar er is momenteel geen efficiënte post-quantum variant beschikbaar, dus konden we alleen KEM-gebaseerde oplossingen implementeren.

Aangezien er al post-quantum KEM's zijn gestandaardiseerd door NIST, was de meest voor de hand liggende aanpak om de klassieke KEM's in het systeem te vervangen door post-quantum varianten. Het systeem is ontworpen om ad-hoc een algoritme af te spreken tussen eindpunten bij het opzetten van een beveiligd kanaal. Omdat er geen KEM bestaat dat in elk gebruiksscenario als beste presteert en algoritmes tijdens het opzetten van een beveiligd kanaal kunnen worden vastgesteld, hebben we ons op drie post-quantum KEM's gericht, namelijk: **FrodoKEM**, **McEliece** en **ML-KEM**.

Onze aanpak was vrij eenvoudig: vind alle huidige verwijzingen naar RSA en Elliptic Curve Cryptography (ECC) en voeg opties toe voor onze drie alternatieven. Klinkt simpel, toch? Laten we kijken hoe het ging.



In dit codevoorbeeld vervangen we verwijzingen naar RSA door verwijzingen naar ML-KEM.

Stap 1: Cryptografische Logica Toevoegen

Als eerste stap zijn we naar de cryptografische kern genavigeerd waar de cryptografische logica van ons systeem

was gedefinieerd. We moesten daar nieuwe logica toevoegen voor de verschillende typen KEM's zodat onze API's ermee zouden kunnen werken. Omdat de codebase afhankelijk is van een externe bibliotheek voor cryptografische operaties, moesten we een versie vinden die de juiste algoritmes had geïmplementeerd en die vervolgens inladen. Gelukkig was er al zo'n versie van deze bibliotheek (Bouncy Castle) beschikbaar en na een paar wijzigingen en extra tests was de nieuwe logica voor de post-quantum KEM's klaar.

Stap 2: Statische Codeanalyse

De volgende stap was om te achterhalen welke delen van de code afhankelijk waren van de specifieke RSA- en ECC-logica, zodat we daarin wijzigingen konden aanbrengen. De meeste ontwikkelsoftware ondersteunt tegenwoordig statische codeanalyse, dus die hebben we ook ingezet om deze verwijzingen te vinden, maar in principe kun je ook externe tools gebruiken. Wat we toen aantroffen oversteeg al onze verwachtingen – en niet in positieve zin. Cryptografische afhankelijkheden bleken diep verweven te zijn in alle delen van de code. Om één extra algoritme te ondersteunen moesten honderden bestanden grotendeels handmatig worden aangepast.

Van:

```
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);
```

Naar:

```
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;
```

Dit is tijdrovend werk, vooral als documentatie ontbreekt of de oorspronkelijke ontwikkelaars niet meer beschikbaar zijn. Helaas was dat bij ons het geval. Dat heeft ons een belangrijke les geleerd.

> Cryptografische wendbaarheid is essentieel bij het migreren van cryptografische code

Het was duidelijk dat de structuur van de code moest worden aangepast zodat het flexibeler zou zijn en het daardoor eenvoudiger zou worden om nieuwe algoritmes toe te voegen. Interessant genoeg voldeed de architectuur van de code aan de richtlijnen voor kwalitatief hoogstaande code. Er werd bij het schrijven van de applicatie alleen niet gerekend op veel extra algoritmes, dus was het destijds efficiënter om bepaalde elementen op een rigide manier te implementeren, zoals

database-indelingen en tests. Dat brengt ons bij de tweede les.

> Kwalitatief hoogstaande code is niet altijd cryptografisch wendbaar

Cryptografische wendbaarheid vereist een andere benadering van codearchitectuur. Je hebt namelijk een structuur nodig waarin cryptografische algoritmes met minimale inspanning kunnen worden toegevoegd of verwijderd.

Stap 3: Herschrijven van de Structuur

We hoopten dat we klaar zouden zijn, maar het was duidelijk dat we nog veel aanpassingen moesten maken om de code cryptografisch wendbaar te maken. We moesten expliciete verwijzingen naar cryptografische algoritmes loskoppelen van de functionele logica van het systeem. Dit betekende concreet dat elke module herschreven moest worden zodat het zou passen binnen een abstracte beschrijving van encryptie en decryptie.

We hebben via dit proces uiteindelijk vijf codeprincipes kunnen identificeren die de cryptografische wendbaarheid van de code aanzienlijk heeft verbeterd. Nadat deze waren geïmplementeerd, was het een fluitje van een cent om nieuwe algoritmes toe te voegen. Dit is ideaal als je verschillende post-quantum algoritmes wil testen in je eigen omgeving en zorgt direct voor een toekomstbestendige codebase. Als een van de nieuwe gestandaardiseerde algoritmes kwetsbaar blijkt, moet je namelijk snel kunnen migreren.

Een cryptografisch wendbare codebase is dus beter bestand tegen onverwachte toekomstige migraties. Daarmee zijn we bij de laatste les aangekomen.

Stap 4: Resultaat Vieren

Onze codebase is nu beter voorbereid op toekomstige cryptografische migraties. De meeste codebases zijn niet van nature cryptografische wendbaar, dus er is een investering nodig. In onze migratie hebben we 293 Java-bestanden aangepast en 3918 regels code aangepast. Dit is een significant gedeelte van de codebase, want het totaal was 2227 Java-bestanden en 164.000 regels code.

Cryptografische wendbaarheid Verbeteren

Hieronder hebben we de vijf codeprincipes opgeschreven die essentieel waren voor het verbeteren van de cryptografische wendbaarheid van onze codebase:

1. Gebruik universele API's

Idealiter maakt de cryptografische kern de koppeling tussen algoritme-identificatie en de betreffende logica. Als bepaalde algoritmes als input extra parameters nodig hebben, zoals RSA en ECC, zorg dan dat de API’s deze extra informatie op een abstracte algortime-onafhankelijke manier accepteren. Het makkelijkste is om parameters in configuratiebestanden te zetten, zodat dat niet in de code zelf geschreven hoeft te worden.

```
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. Gebruik Dynamische Structuren voor geheugen

Als algoritme-specifieke waardes moeten worden opgeslagen, vermijd dan top-level variabelen. Gebruik in plaats daarvan dynamische structuren zoals maps en dictionaries, waarbij de sleutel (key) een identificatiewaarde is voor een algoritme, zodat algoritmes flexibel kunnen worden toegevoegd of verwijderd. Dit maakt het ook makkelijker voor andere modules om algoritmespecifieke waardes op te vragen.

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

Deze snippet toont een dynamische structuur in Java, specifiek een ‘Map’. Het vertaalt een identificatiewaarde van een asymmetrisch algoritme naar een object waarin encryptiesleutels worden opgeslagen.

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

Deze snippet laat zien hoe de 'Map' gebruikt kan worden in een test om te garanderen dat het opgevraagde object met encryptiesleutels overeenkomt met de verwachte waarde

3. Gebruik Geparametriseerde Tests

De beste unit tests testen of het systeem zich op de juiste manier gedraagt in verschillende scenario's. Het is beter om een geparametriseerde test te schrijven dan om per algoritme een losse test te schrijven, die allemaal hetzelfde gedrag testen. Als er een nieuw algoritme wordt toegevoegd, of een algoritme wordt verwijderd, hoef je de test niet meer aan te passen.

```
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 dit voorbeeld kun je zien dat de argumenten voor de testGenerateKeyPair test automatisch gegenereerd worden vanuit de lijst met bestaande asymmetrische algoritmes. Dit is alleen mogelijk als de tests via een identificatiewaarde van algoritmes bepaalde objecten kunnen opvragen. Dit laatste is in de vorige sectie beschreven.

4. Gebruik Een Geschikt Mechanisme Om Sleutels Op te Slaan

De meeste cryptografische applicaties slaan sleutels op in geheugen. Meestal is die opslag een aparte applicatie, zoals een database, en die verwacht dan een specifiek dataformat voor die sleutels. Als dit format alleen geschikt is voor RSA of ECC, dan is er een grote kans dat het systeem een foutmelding geeft als een ander algoritme wordt gebruikt. Het is daarom belangrijk dat de juiste instellingen worden gebruikt zodat meerdere algoritmetypes opgeslagen kunnen worden. Bijvoorbeeld, database indelingen kunnen worden aangepast met nieuwe tabellen die de vertaling tussen algoritmes en dataobjecten opslaan. Als er specifiek code-annotaties worden gebruikt om de database indeling in te stellen en cryptografische informatie in losse variabelen wordt opgeslagen, dan moeten er twee verandering plaatsvinden. Ten eerste moeten de variabelen worden vervangen door dynamische structuren, zoals eerder is uitgelegd. Daarna moeten de database-annotaties worden aangepast voor de nieuwe structuren.

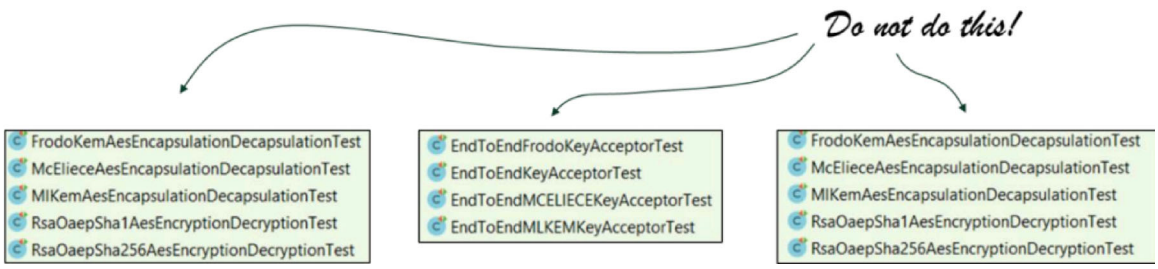
```
@ElementCollection
@CollectionTable(name = “example_table_name", joinColumns = @JoinColumn(name =
“example_id"))
@MapKeyColumn(name = "key_type")
private Map<AsymmetricKeyType, @NotNull @Valid EncryptionKeys> encryptionKeysMap;
```

In dit voorbeeld is dezelfde ‘Map’ te zien die in een eerder voorbeeld is gebruikt, waarbij Jakarta annotaties zijn toegevoegd om de database-indeling te specificeren.

Wat is goed is om te weten, is dat de sleutellengte van symmetrische algoritmes mogelijk ook aangepast moet worden. Dit hangt af van het cryptografische beleid van de organisatie. Mocht dit het geval zijn, zorg dan dat je sleutelopslag hiermee om kan gaan. Nog meer reden om een cryptografisch wendbaar systeem te hebben!

5. Gebruik een Modulaire Klassenstructuur

De bestandstructuur van een applicatie heeft een grote invloed op de cryptografische wendbaarheid. Als verschillende concepten een extra bestand vereisen per algoritme, dan moeten die over het algemeen handmatig worden toegevoegd en verwijderd. Het is daarom aan te raden om met klassen te werken die flexibel om kunnen gaan met verschillende algoritmes. Meestal gaat dit redelijk vanzelf als de eerdere suggesties al zijn doorgevoerd.



Dit overzicht toont verschillende Javabestanden. Deze structuur is niet cryptografisch wendbaar, omdat er voor elk concept en elk algoritme een apart bestand is.

Conclusie

Tijdens ons avontuur hebben we met succes post-quantum algoritmes in onze software kunnen verwerken. Maar belangrijker nog: we hebben op de harde manier geleerd dat het essentieel is om de code zodanig te herstructureren dat toekomstige wijzigingen eenvoudiger worden.

De migratie van cryptografische algoritmes kan aanzienlijk worden versneld door cryptografische wendbaarheid te integreren in je codebase. Helaas is hoogwaardige code niet automatisch cryptografisch wendbaar. Het kost wat tijd om het goed te doen, maar onze vijf principes kunnen daarin houvast bieden. Door deze principes aan te houden in je code,

minimaliseer je de impact van ontbrekende documentatie of expertise en zullen toekomstige migraties een stuk soepeler verlopen.

Houd de andere blogs in deze serie ook in de gaten! In deze reeks van vier delen, delen we zowel organisatorische als technische resultaten vanuit vier perspectieven: Management, Developers, IT Architecten en Vendoren.

Disclaimer: beeld is gecreerd met Copilot

Deel deze pagina

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