SYNGENIO AG

Java Upgrade und Security

Von Erich Becker, SYNGENIO AG

Es ist 2016, Java 7 ist ein alter Hut, Java 8 läuft auf produktiven Systemen und wir warten darauf, was das verschobene Java 9 denn so bringt. Ist das überall so? Nein, denn es gibt durchaus große Firmen, die aus gutem Grund eine eher konservative Upgrade-Politik fahren und auf deren Servern vielleicht nicht mehr Java 5, aber zumindest Version 6 läuft. Angesichts der Tatsache, dass Oracle bereits (seit fast einem Jahr) den Support für Java 7 eingestellt hat, müssen aber auch diese Nutzer über ein Upgrade auf mindestens JDK 7, besser 8, nachdenken. Das ist ja auch nicht so schwierig, oder? Man kann ja sogar dem Compiler mitteilen, dass man Code Kompatibilität mit älteren Versionen wünscht. Was kann schiefgehen? Nun, eine ganze Menge, z. B. fordert Java 7 einen größeren Stack als Java 6, und hat als Standard einen anderen Garbage Collector eingestellt. Hier ist also Feintuning angesagt. Dazu findet man im Netz ein paar Anleitungen. Hier geht es um das Risiko, sich durch die verbesserte technische Security eine schlechtere Interoperabilität einzuhandeln, was, wenn man nach den Artikeln in Netz geht, offenbar nicht so häufig auftritt, aber dafür beim Eintreten erhebliche Probleme hervor rufen kann.

Sichere Kommunikation

Wir leben in einem Zeitalter der Hacker, Netzspionage, und Cyber Angriffe. Daher ist es nur natürlich und gut, wenn die Kommunikation von IT-Systemen möglichst über gesicherte Kanäle abläuft. Für Kommunikation mittels SOAP oder REST bietet sich HTTPS an. Aber HTTPS ist nur eine Hülle für ein Handshake-Protokoll wie SSL und TLS und der Handshake vereinbart dann einen Satz von kryptografischen Algorithmen für den Schlüsselaustausch, die Verschlüsselung selbst und die Prüfsummenberechnung, eine sogenannte Cipher Suite.

Normalerweise braucht man sich nicht selbst darum zu kümmern, das erledigt entweder der Application Server (bei eingehenden HTTPS-Verbindungen) oder die HTTP Client Library für einen.

Nun ist es aber so, dass es ein Wettrennen zwischen den Erfindern und Implementatoren kryptografischer Algorithmen auf der einen Seite und Hackern, die die Algorithmen selbst oder deren Implementierungen brechen wollen, auf der anderen Seite gibt. Was gestern als sicher galt, ist heute schon bedenklich. Dazu kommt, dass das Moorsche Gesetz auch dazu führt, dass es immer leichter wird, eine verschlüsselte Kommunikation mit einem Brute-Force-Angriff zu knacken. Das führt dazu, dass neue Algorithmen hinzukommen und bei anderen größere Schlüssellängen unterstützt werden. Dankenswerterweise unterstützt uns da das JDK, in dem es diese neuen Algorithmen und Schlüssellängen zur Verfügung stellt und alte, als unsicher geltende, deaktiviert. Damit sind wir also auf der sicheren Seite.

Was aber, wenn wir eine Komponente haben, die in einer heterogenen Umgebung mit sehr vielen Partnern über abgesicherte Kanäle kommuniziert? Was ist, wenn nicht alle dieser Partner auf neuere Algorithmen wechseln können oder (z. B. aus Kostengründen) wollen?

Dann kann es sein, dass einer der Kommunikationspartner sich nach einem Java-Ugrade nicht mehr mit einem Anderen auf eine Cipher Suite – einen Satz zu verwendender kryptografischer Algorithmen – einigen kann. Oder sie kommen sogar nicht einmal bei der Methode, wie sie sich einigen sollen, auf einen Nenner. Letzteres kann passieren, wenn einer der Kommunikationspartner nur SSL spricht, und ein anderer auf JDK 7 aktualisiert wird, in dem SSL(v3)[1] normalerweise deaktiviert ist. (SSLv2 gilt schon seit Jahren als unsicher und sollte eigentlich schon lange nicht mehr verwendet werden.)

Einfache Lösung

Die Lösung in diesem Fall ist im Prinzip einfach: Java unterstützt die Reaktivierung ausgeschalteter Algorithmen. Alle Cipher Suites, die es in JDK 6 gab, gibt es auch in JDK 7 und 8.

Das wirft die folgenden Fragen auf:

1. Wie reaktiviere ich Algorithmen?
2. Wenn ich alte Algorithmen erlaube, wird mein System dadurch dann nicht unsicherer?

Für die erste Frage finden sich Lösungen in der Oracle-Dokumentation und auch in Foren im Netz. Die Antwort auf die zweite Frage ist ein klares Jein. Nein, denn das System hat ja vor dem JDK Update schon die gleichen Algorithmen unterstützt. Und ja, was gestern noch sicher war, ist heute vielleicht unsicher, stärkere Algorithmen und größere Schlüssellängen sollten auf jeden Fall vorgezogen werden. Zudem raubt man den Kommunikationspartnern den Anreiz, auf neue Algorithmen zu wechseln, wenn man die alten weiterhin unterstützt.

Andererseits ist der Hauptzweck einer Firmen-IT, Geld entweder direkt zu verdienen oder die Fachbereiche beim Geldverdienen zu unterstützen. Eine sichere Kommunikation, die nicht stattfindet, weil sich die Systeme nicht einigen können, verhindert dies, und nützt damit niemandem. Ziel muss es also sein, alte Cipher Suites zunächst weiter zu unterstützen und nach und nach durch neue zu ersetzen.

(De)Aktivieren von Algorithmen in Java

Es gibt in Java zwei Möglichkeiten, bestimmte Algorithmen zu aktivieren oder deaktivieren:

1. Programmatisch. Dies soll hier nur kurz erwähnt werden, aber nicht weiter auf die Details eingegangen werden. Wenn man in Java eine HTTPS-Verbindung öffnen will, muss man die `SslSocketFactory` überschreiben, dort im `SSLContext` vor dem Handshake ein `SSLParameter` Objekt setzen, in dem die erlaubten Cipher Suites und optional die erlaubten Protokolle (SSL,TLS) gesetzt sind. Dies ist also ein Whitelist-Verfahren. Es hat zwei Nachteile:

1 SSL (Secure Socket Layer) wurde 1994 durch Netscape in der Version v1 eingeführt. Diese hatte jedoch erhebliche Sicherheitslücken, sodass erst Version v2 wirklich zum Einsatz kam. In die Version v3 flossen Verbesserungen aus Microsofts Alternativprotokoll PCT ein. TLS (Transport Layer Security) ist der Nachfolger von SSLv3 und liegt zurzeit in Version 1.3 vor.

- Es erfordert einen Eingriff in das bestehende Programm. Je nachdem, wie die `SslSocketFactory` überschrieben wird, hat das Nebenwirkungen auf ganz andere Systemteile.
- Es funktioniert nur bei ausgehenden Verbindungen, oder wenn man den HTTP(S) Server selbst implementiert. Warum sollte man Letzteres tun, wenn die Funktionalität im Application Server oder Servlet Container schon enthalten ist?

2. Es gibt eine Möglichkeit, das Problem konfigurativ zu lösen. Im JDK gibt die Datei `$JAVA_HOME/jre/lib/security/java.security`. Dort ist unter anderem festgelegt, welche Algorithmen benutzt werden dürfen und sollen. Insbesondere interessieren uns die Properties `jdk.tls.disabledAlgorithms` und `jdk.tls.legacyAlgorithms` (Letzteres erst ab JDK 8). Das Erste ist eine Blacklist, das Zweite eine Art von Greylist, d. h. die dort gelisteten Algorithmen werden zwar unterstützt, aber nur mit niedriger Priorität, d. h. sie werden nur ausgewählt, wenn die Gegenseite keine der moderneren Alternativen unterstützt. Diese Properties unterstützen auch Wildcards. Es empfiehlt sich, dazu die Dokumentation in der Datei `java.security` genau durchzulesen. Sollte es Probleme mit Zertifikaten geben, gibt es übrigens auch hierfür passende Konfigurations-Properties.

Ich rate allerdings davon ab, die Datei im JDK selbst zu verändern. Die Gründe dafür sind:

- Im Zweifel werden zusätzliche Rechte für die Veränderung benötigt, weil jemand anderes für die Installation des JDK zuständig ist.
- Es kann sein, dass mehrere Anwendungen sich eine JDK-Installation teilen. Durch die Änderung sind dann alle Anwendungen betroffen.
- Die Änderungen müssen bei jedem minor Update des JDK nachgezogen werden.

Glücklicherweise bietet Java die Möglichkeit, seine Sicherheitseinstellungen zusätzlich aus einer weiteren Datei zu ziehen: Mit dem Kommandozeilen-Parameter `-Djava.security.properties=datei-im-classpath` kann diese Datei angegeben werden. Diese zusätzliche Datei muss nur die Properties enthalten, die überschrieben werden sollen, in unserem Fall also `jdk.tls.disabledAlgorithms` und ggf. `jdk.tls.legacyAlgorithms` . Um die mit JDK deaktivierten SSL3 Algorithmen wieder zu erlauben, muss die entsprechende Property leer sein, also `jdk.tls.disabledAlgorithms=`

3. Neben der bisher beschriebenen Möglichkeit der Konfguration auf Java-Ebene gibt es noch die Möglichkeit, im Application Server die zugelassenen Cipher Suites zu konfigurieren. Dies geschieht unter Tomcat in der `server.xml`, unter Weblogic in der `weblogic.xml`. Damit ist aber nur eine weitere Einschränkung gegenüber den in Java verfügbaren Ciphers möglich, d. h. eine Erweiterung funktioniert nicht. Trotzdem kann es gewünscht sein, diese Einschränkung vorzunehmen, etwa wenn für ausgehende Verbindungen noch veraltete Algorithmen benötigt werden, für eingehende aber nicht und deren Verwendung ausgeschlossen werden soll.

Analyse der Kommunikation

Bevor wir das JDK-Upgrade durchführen, sollten wir zunächst feststellen, welche Algorithmen denn überhaupt zum Einsatz kommen. Auch hier gibt es wieder zwei verschiedene Ansätze:

1. Mit dem Kommandozeilen-Parameter `-Djavax.net.debug=ssl:handshake` teilen wir Java mit, dass alle SSL Handshakes im Detail nach System.out geloggt werden sollen. Das funktioniert bei eingehenden und ausgehenden Verbindungen und zeigt auch an, wenn der Handshake nicht funktioniert hat, etwa, weil sich die Kommunikationspartner nicht auf eine gemeinsame Cipher Suite einigen konnten.

Ein Ausschnitt des Loggings könnte etwa so aussehen:

```
catalina-exec-166, received EOFException: error
Ignoring unavailable cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA
Ignoring unavailable cipher suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
Ignoring unavailable cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA256
Ignoring unavailable cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
Ignoring unavailable cipher suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA
Ignoring unavailable cipher suite: TLS_RSA_WITH_AES_256_CBC_SHA
catalina-exec-166, handling exception: javax.net.ssl.SSLHandshakeException: Re-
mote host closed connection during handshake
catalina-exec-161, READ: TLSv1.2 Handshake, length = 229
catalina-exec-166, SEND TLSv1 ALERT:  *** ClientHello, TLSv1.2
RandomCookie:  GMT: 1440833454 bytes = { fatal, *** ClientKeyExchange, RSA Pre-
MasterSecret, TLSv1
219, 207, 29, 194catalina-exec-173, WRITE: TLSv1 Handshake, length = 1733
```

Der Nachteil ist, dass hier sehr große Datenmengen geschrieben werden, sodass der Parameter nur für kurze Zeit und idealerweise nur für bilaterale Tests zwischen zwei Systemen gesetzt sein sollte. Außerdem müssen die erzeugten Logfiles interpretiert werden.

2. Es ist auch möglich, gezielt Informationen zu den ausgehandelten Protokollen und Algorithmen in Java programmatisch zu sammeln. Das hat natürlich auch Nachteile:

    - Es ist ein Eingriff in bestehenden Code erforderlich.
    - Der zu verwendende Methode unterscheidet sich für eingehende und ausgehende Verbindungen.
    - Die verfügbaren Informationen unterscheiden sich für ausgehende und eingehende Verbindungen.
    - Für eingehende Verbindungen erfährt man nichts über Abbrüche des Handshakes, weil der eigene Code ja erst nach erfolgreichem Handshake aufgerufen wird.

Eingehende Verbindungen

Bei eingehenden Verbindungen übernimmt der Application Server den Handshake. D. h., unser Code wird nur aufgerufen, wenn der Handshake erfolgreich war. Wir können aber über Request-Attribute auf die ausgehandelte Cipher Suite und die Schlüssellänge sowie das Client-Zertifikat zugreifen.

```
String cypherSuite = (String)request.getAttribute(„javax.servlet.request.
cipher_suite“);
int keyLength = (Integer)request.getAttribute(„javax.servlet.request.key_
size“);
Object certificate = request.getAttribute(„javax.servlet.request.X509Certifica-
te“);
```

Vorsicht, das Zertifikat kann entweder als `X509Certificate` oder in der serialisierten Form vorliegen. Laut Dokumentation sollte auch die `SSLSession` als Attribut `javax.net.ssl.session` vorliegen, das wird jedoch nicht von jedem Application Server interpretiert. Damit kann man z. B. nicht das Protokoll abfragen.

Ausgehende Verbindungen

Wenn man selbst eine HTTPS-Verbindung aufbaut, z. B. mit der Apache HTTP Library, kann man seine eigene SocketFactory registrieren:

```
HttpCoreClient httpCoreClient = HttpCoreClientFactory.getHttpCoreClient();
[...]
httpCoreClient.registerHttpCoreSSLSocketFactory(sslSocketFactory);
```

In der Factory selbst kann man in der Methode `connectSocket(...)` einen HandshakeCompletedListener registrieren:

```
sslSsocket.addHandshakeCompletedListener(this);
```

Ein `HandshakeCompletedListener` implementiert die folgende Methode, die aufgerufen wird, wenn ein SSL-Handshake abgeschlossen wurde:

```
@Override
public void handshakeCompleted(HandshakeCompletedEvent handShakeEvent) {
      final SSLSession session = handShakeEvent.getSession();
      String cipherSuite = session.getCipherSuite();
    [...]
}
```

Zu beachten ist, dass eine SSLSession keinen direkten Zugriff auf die Schlüssellänge erlaubt, dafür kann man das Protokoll und die Zertifikat Kette des Servers abfragen.

Weiteres Vorgehen

Wenn wir festgestellt haben, welche Algorithmen im Augenblick genutzt werden, können wir gezielt im neuen JDK aktivieren, bzw. von der Blacklist nehmen. Wenn das neue JDK die Version 1.8.x hat, können wir sie zusätzlich als Legacy markieren (s. o.), um Partnersysteme zu entmutigen, sie weiter zu nutzen. Um zu verhindern, dass neue Kommunikationspartner veraltete, bisher nicht genutzte Algorithmen verwenden, sollten diese auf jeden Fall deaktiviert bleiben.

Eine weitere Möglichkeit, die Reihenfolge der Bevorzugung zu beeinflussen, wurde bisher noch nicht erwähnt: Der Komandozeilen-Parameter -Dhttps.protocols=TLS,SSL legt fest, dass bevorzugt TLS, dann erst SSL verwendet werden soll. Hier sind beliebige Kombinationen denkbar.

Ein Hinweis noch zum Schluss: Nicht immer sind neuere Versionen des JDK auch besser. Das gilt auch für das Gebiet der Sicherheitsprotokolle und Algorithmen. So wurde etwa im JDK 7 Update 75 eine Sicherheitslücke gegen den POODLE Angriff geschlossen, aber der Fix war fehlerhaft, er schloss zwar die Sicherheitslücke, unterband aber unter bestimmten Bedingungen auch legitime Verbindungen. Das wurde erst in JDK 7u85 behoben. Es muss also im Einzelfall geprüft werden, welche JDK Version für das System und die Kommunikationspartner die geeignete ist. Hierzu sollten vorab Tests in einer nichtproduktiven Umgebung, aber mit den gleichen Kommunikations-Partnern und -Parametern wie in der Produktion durchgeführt werden.