Streaming-basierte Ver-/Entschlüsselung und WS-Policy Validierung von SOAP-Nachrichten

Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Netz- und Datensicherheit
Horst Görtz Institut für IT-Sicherheit
Ruhr-Universität Bochum

Marc Giger

07.01.2011

1. Prüfer: Prof. Dr. rer. nat. Jörg Schwenk
2. Prüfer: Jun. Prof. Dr. Thorsten Holz
Betreuer: M. Sc. Juraj Somorovský

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Masterarbeit mit dem Thema

Streaming-basierte Ver-/Entschlüsselung und WS-Policy Validierung von SOAP-Nachrichten

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bern, den 7. Januar 2011

Zusammenfassung

Web Services ist ein Standard, der die interoperable Kommunikation von Computer zu Computer über ein Netzwerk ermöglicht. Für die Web Service Kommunikation wird heute meistens das SOAP Protokoll verwendet. Zur Sicherung von Web Services wurde der WS-Security Standard definiert. WS-Security stellt die Integrität, Authentizität und Vertraulichkeit der Daten auf der Nachrichten-Ebene sicher. Die Verarbeitung der gesicherten Nachrichten geschieht im Allgemeinen über die Standard XML DOM-API. Das Einlesen von XML mit einem DOM-Parser, zusätzlich mit aufwendigen kryptographischen Berechnungen, ermöglicht auf einfachste Art und Weise Denial-of-Service Angriffe. Weitere Angriffsflächen ergeben sich durch den Einsatz von WS-Security selbst, oder durch falsche XML-Parser Einstellungen.
WS-SecurityPolicy ist ein Standard, mit dem sich die Anforderungen und Bedingungen eines Web Services bezüglich Sicherheit beschreiben lassen. Mit der XML-Schema Validierung und sorgfältig ausgearbeiteten Security-Policies können Angriffe wie "XML Signature Element Wrapping Attacks" verhindert werden.
Diese Arbeit präsentiert eine Lösung zu den Problemen, die das Verarbeiten von grossen SOAP-Nachrichten auf effiziente Art und Weise ermöglicht. Das Streaming-WebServices-Security-Framework ist entstanden. Wie der Name bereits andeutet, werden die Nachrichten streaming-basiert verarbeitet. Die Integration von WS-SecurityPolicy ermöglicht den sofortigen Abbruch, falls eine Policy Verletzung stattgefunden hat. Die Implementierung unterstützt das Verschlüsseln ausgehender, sowie die Entschlüsselung eingehender SOAP-Nachrichten. Die Policy Validierung geschieht unmittelbar beim Auftreten eines sicherheitsrelevanten Ereignisses. Die Evaluierung zeigt, dass der streaming-basierte Ansatz effizienter und ressourcenschonender ist im Vergleich zur Standard DOM Implementierung. Ebenfalls kann gezeigt werden, dass das Streaming-WebServices-Security-Framework resistent gegen die bekannten Angriffe ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Der Begriff Web Services wird oft im Zusammenhang mit SOA (Service Oriented Architecture, Dienstorientierte Architektur) genannt. Gemeint ist aber keineswegs das Gleiche. Die dienstorientierte Architektur ist ein Konzept, das einem Client unabhängige, lose und wiederverwendbare Dienste anbietet. Dabei werden die Dienste nicht an einem zentralen Ort angeboten, sondern sind über die verschiedenen Prozesse in einem Unternehmen, die die Daten verwalten, über wohldefinierte Schnittstellen erreichbar. SOA ist ein übergreifendes Konzept und schreibt nicht die Benutzung eines bestimmten Protokolls oder Standards vor. Web Services ist ein konkreter Standard [Web] auf Basis einer dienstorientierten Architektur und ist heute der meist eingesetzte im SOA Bereich. Entwickelt und standardisiert wird Web Services durch OASIS [OAS] und durch W3C [W3Ca]. Die zugrunde liegende Idee von Web Services ist nicht neu, wie man meinen könnte. Als Vorreiter können zum Beispiel CORBA und DCOM genannt werden, die alle mit dem Gedanken entwickelt wurden, eine verteilte Softwarearchitektur zu schaffen.
Web Services selber ist auch nur ein Überbegriff für eine Sammlung von Technologien. Eine zentrale Rolle nimmt SOAP [BEK+00] ein, welches meist als Kommunikationsprotokoll im Web Services Umfeld verwendet wird. SOAP Nachrichten basieren auf dem XML Standard und werden im Internet meistens über das HTTP Protokoll übertragen. Oftmals beinhalten die SOAP Nachrichten sensitive und vertrauenswürdige Daten, die geschützt werden müssen. Ein gängiges Verfahren um die Informationen zu schützen ist TLS (Transport Layer Security). Bei TLS wird ein Kommunikationskanal zwischen den Partnern aufgebaut der die Daten auf der Transport-Ebene vollständig sichert. Die Nachrichten liegen dann beim Empfänger wieder in ungesicherter Form vor, da diese nur auf dem Transportweg gesichert wurden. Dieser Empfänger muss aber nicht unbedingt der Ziel-Service sein, sondern kann eine intermediäre Stelle, der die Daten zum Beispiel filtert und dann weiterleitet sein. In anderen Szenarien ist es vorstellbar, dass die Nachrichten nicht sofort verarbeitet werden können und verschlüsselt zwischengespeichert werden müssen.
Aus diesen Gründen ist es sinnvoll, die Nachrichten nicht nur auf Transport-Ebene sondern auch auf Nachrichten-Ebene sichern zu können. Zu diesem Zweck ist von OASIS der WS-Security Standard [LK06] entwickelt worden, der auf weiteren Standards wie XML Signature [RSH+08] und XML Encryption [IDS02] aufbaut. WS-Security ermöglicht das Verschlüsseln beliebiger Teile der SOAP-Nachricht mit Schlüsseln verschiedener Empfänger. Der jeweilige Adressat kann somit nur die Informationen lesen, die für ihn bestimmt sind.
Der Einsatz von Sicherheitsmechanismen auf Nachrichten-Ebene bietet auch neue Angriffsflächen. So lässt sich ein Web Service durch aufwendige kryptographische Berechnungen in die Knie zwingen. Andererseits birgt selbst auch der WS-Security Standard durch seine Flexibilität neue Gefahrenpotentiale.
Um WS-Security anwenden zu können wird typischerweise das DOM Verarbeitungsmodell verwendet. Dies stellt ein zusätzlicher Schwachpunkt dar. Für die weitere Verarbeitung muss das XML-Dokument vom DOM-Parser zuerst vollständig in ein Objekt-Baum eingelesen

werden. Der ganze Objekt-Baum liegt während dieser Zeit im Speicher des Computers. Dieser Vorgang benötigt sehr viel Prozessor- und Speicher- Ressourcen. Sendet ein Angreifer übergrosse SOAP-Dokumente, kann dies zu einer Denial-of-Service (DoS) Attacke führen. Bei verschlüsselten Dokumenten ist der Speicherverbrauch noch höher. Zuerst muss die ganze SOAP- Nachricht in den Speicher gelesen werden, danach findet die Entschlüsselung statt. Als letztes wird der entschlüsselte XML-Teil wiederum als Objekt-Baum aufgebaut. Zu diesem Zeitpunkt sind die verschlüsselten Daten und das entschlüsselte Teil-Dokument im Speicher vorhanden. Erst dann, kann der verschlüsselte XML Teil mit dem entschlüsselten ersetzt werden.
WS-Security ermöglicht die Integrität, Authentizität und Vertraulichkeit auf Nachrichten-Ebene. Welche Teile der SOAP-Meldung wie gesichert werden müssen, ist im WS-Security Standard aber nicht beschrieben. Was sind die Anforderungen an einen SOAP Client für einen erfolgreichen Zugriff auf einen Web Service? Muss der komplette SOAP-Body verschlüsselt sein? Wird ein Zeitstempel erwartet? Muss die Nachricht signiert sein? Welche Schlüssel und in welchem Format werden sie benötigt? Um diese und weitere Anforderungen und Bedingungen ausdrücken zu können, wurde von OASIS [OAS] der WS-SecurityPolicy Standard [NGG+09] eingeführt.
Wird WS-SecurityPolicy in einem DOM Umfeld angewendet und der Client sendet eine Nachricht, die nicht der Policy entspricht, werden wieder viele Computer-Ressourcen unnötig verschwendet. Der DOM-Parser liest die Nachricht vollständig ein, das WS-Security Framework verarbeitet das Dokument anhand des Security-Headers, und zuletzt wird bemerkt, dass das Dokument nicht so gesichert wurde wie es die Policy verlangt.

Diese Arbeit stellt ein streaming-basiertes WebServices-Security-Framework vor, mit der Fähigkeit, grosse SOAP-Dokumente effizient zu verarbeiten. Es unterstützt das Verschlüsseln ausgehender sowie die Entschlüsselung eingehender SOAP-Nachrichten. Die Verarbeitung der Nachrichten geschieht streaming-basierend über die StAX API [StA]. Durch den streaming-orientierten Ansatz ist es möglich, ein eingehendes Dokument nach und nach einzulesen und zu verarbeiten, ohne dass das ganze Dokument im Speicher gehalten werden muss. Ist es nicht möglich die Nachricht zu verarbeiten, weil zum Beispiel der Schlüssel zum Entschlüsseln unbekannt ist, kann die Verarbeitung sofort abgebrochen werden.
Die Integration von WS-SecurityPolicy ins Framework ermöglicht den schnellen Abbruch ("fail-fast") bei Policy Verletzungen. Dieses Verhalten wird erreicht, indem die Policy immer unmittelbar beim Auftreten eines Policy relevanten Ereignisses verifiziert wird.

2 Standards und Technologien

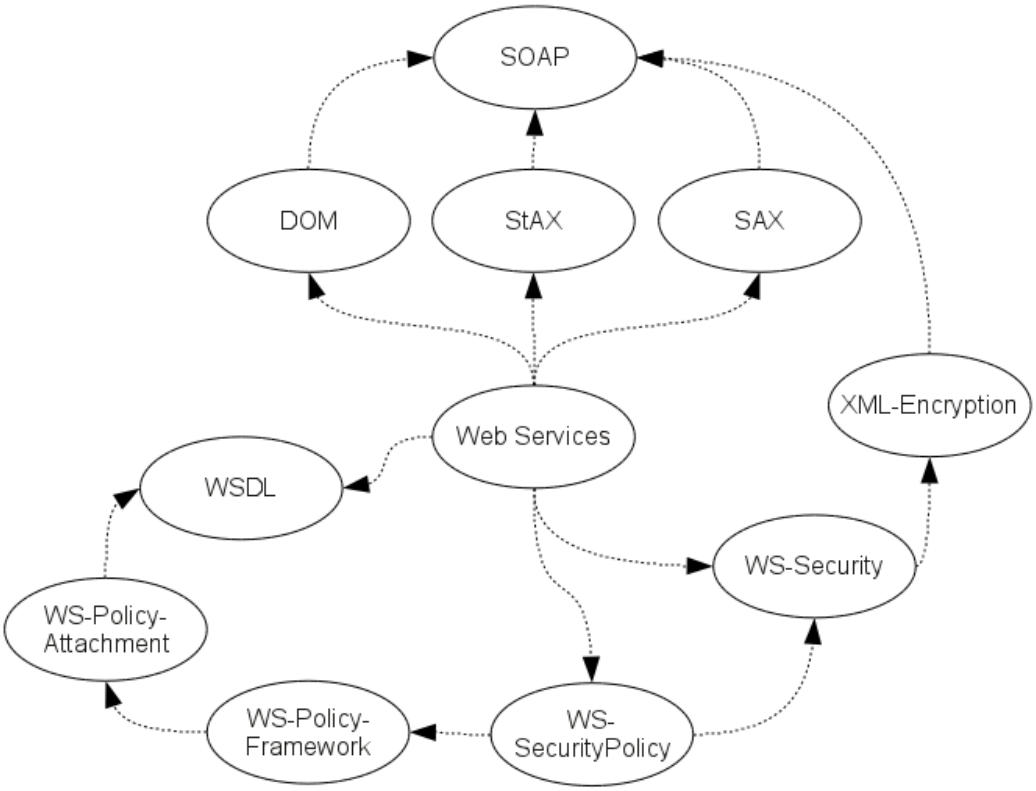


Abbildung 2.1: Übersicht über die verwendeten Standards

2.1 Verarbeitung von XML Dokumenten

Zurzeit existieren drei verschiedene Ansätze um XML Dokumente zu verarbeiten:

- DOM-API: Die ganze XML Baumstruktur wird im Speicher abgebildet.
- Push-API (SAX): Der Parser liest das XML ein und meldet XML-Ereignisse über Callback an die Applikation.
- Pull-API (StAX): Ähnlich wie mit einem Iterator kann über die XML-Ereignisse iteriert werden.

2.1.1 Document Object Model (DOM)

DOM ist ein vom W3C [W3Ca] entwickelter Standard [W3Cb] um eine einheitliche Schnittstelle unabhängig von der Programmiersprache zu schaffen. Der Zugriff erfolgt über einen Objektbaum, womit es möglich ist ein XML nach belieben einzulesen, zu modifizieren und auch wieder zu schreiben. Da das ganze XML im Speicher gehalten wird, ist es nicht geeignet für grosse Dokumente. Die maximale Grösse der XML Dokumente die verarbeitet werden können, wird limitiert durch den verfügbaren Arbeitsspeicher.

2.1.2 Simple API for XML (SAX)

Anders als bei DOM wird bei SAX [Meg] nicht das ganze Dokument eingelesen. Der SAX-Parser durchläuft ein XML Dokument sequentiell und ruft bei jedem XML Ereignis (Start-Element, End-Element, Text, et cetera) eine von der Applikation registrierte Callback-Methode auf (Push-Parsing). Daraus ergibt sich aber der Nachteil, dass ohne Zwischenspeicherung kein wahlfreier Zugriff stattfinden kann. Ein weiterer Nachteil von SAX ist, dass keine Dokumente erzeugt werden können.

2.1.3 Streaming API for XML (StAX)

Im Gegensatz zum SAX Parser wird beim StAX [StA] Parser aktiv der nächste Teil des XML Dokuments angefordert (Pull-Parsing). StAX unterscheidet weiterhin zwischen den grundsätzlichen Verarbeitungsmodellen “Iterator” und “Cursor”. Beim Cursor-Modell [Listing 2.1] arbeitet man direkt mit dem Parser Objekt. Beim Iterator-Modell [Listing 2.2] hingegen, liefert der Parser ein “XMLEvent” Objekt, das an andere Methoden übergeben und zwischengespeichert werden kann. Die Verarbeitung mit dem Iterator Modell ist flexibler aber ein wenig aufwändiger. Das Cursor Modell ist einfacher und ein bisschen schneller aber nicht ganz so flexibel. Ein Vorteil gegenüber von SAX ist, dass auch das Schreiben von Dokumenten möglich ist.

```
XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader
    (inputStream);
while (xmlStreamReader.hasNext()) {
    int event = xmlStreamReader.next();
    switch (event) {
        case XMLStreamConstants.START_ELEMENT:
            QName startElement = xmlStreamReader.getName();
            break;
        case XMLStreamConstants.CHARACTERS:
            String characters = xmlStreamReader.getText();
            break;
        case ...
    }
}
```

Listing 2.1: StAX Cursor-Modell

```
XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
XMLEventReader xmlEventReader = xmlInputFactory.createXMLEventReader(
    inputStream);
while (xmlEventReader.hasNext()) {
    XMLEvent xmlEvent = xmlEventReader.nextEvent();
    int eventType = xmlEvent.getEventType();
    switch (eventType) {
        case XMLStreamConstants.START_ELEMENT:
            StartElement startElement = xmlEvent.asStartElement();
            break;
        case XMLStreamConstants.CHARACTERS:
            Characters characters = xmlEvent.asCharacters();
            break;
        case ...
    }
}
```

Listing 2.2: StAX Iterator-Modell

2.2 XML Security Standards

Die von W3C [W3Ca] und OASIS [OAS] entwickelten XML Security Standards verfolgen das Ziel der Sicherung von XML Dokumenten. Dabei führen diese Standards keine neuen Sicherheitsalgorithmen ein, sondern definieren Metadaten zur Sicherung. Zu diesen Standards gehören unter anderem XML-Signature (XDSIG) [RSH+08], XML-Encryption (XENC) [IDS02], XML Key Management Specification (XKMS), Security Assertion Markup Language (SAML). XML-Encryption bildet die Grundlage dieser Arbeit, weshalb im nächsten Abschnitt genauer darauf eingegangen wird.

2.2.1 XML Encryption (XENC)

XML Encryption [IDS02] gewährleistet die Vertraulichkeit von XML-Daten durch Verschlüsselung. Dabei können ganze XML Dokumente oder auch nur Teile davon verschlüsselt werden. Ebenfalls ist es möglich, dass verschiedene Teile des XML Dokuments mit unterschiedlichen Schlüsseln verschlüsselt werden. Dies ermöglicht zum Beispiel, dass die gesicherten Teile des XML nur für bestimmte Empfänger lesbar sind.

Struktur Die Struktur einer XML Verschlüsselung nach XENC zeigt Listing 2.3 auf (wobei ein "?" das 0..1, ein "+" das 1..n, ein "*" das 0..n vorkommen bedeutet. Ein leeres Element "/>" bedeutet, dass es keine Kind-Knoten besitzt).

Einen verschlüsselten XML Block wird immer mit dem Element `<EncryptedData>` eingeleitet. Mittels dem Kind-Element `<EncryptionMethod>` wird der Verschlüsselungsalgorithmus beschrieben, der Angewendet wurde. Falls das Element nicht vorhanden ist, muss der Algorithmus beim Empfänger bereits bekannt sein, da sonst die Entschlüsselung nicht möglich ist.

```
<EncryptedData Id? Type? MimeType? Encoding?>
  <EncryptionMethod/>?
  <ds:KeyInfo>
    <EncryptedKey>?
    <AgreementMethod>?
    <ds:KeyName>?
    <ds:RetrievalMethod>?
    <ds:*>?
  </ds:KeyInfo>?
  <CipherData>
    <CipherValue>?
    <CipherReference URI?>?
  </CipherData>
  <EncryptionProperties>?
</EncryptedData>
```

Listing 2.3: XML Verschlüsselung Struktur

Das <KeyInfo> Element referenziert den Schlüssel mit welchem die Daten verschlüsselt wurden. Für die Schlüsselreferenzierung wurde von XENC auf den XDSIG Standards zurückgegriffen. Der XDSIG Standard spezifiziert diverse Möglichkeiten wie der Schlüssel zur Verfügung gestellt werden kann. Arbeitet man zum Beispiel mit dem PKI Verfahren, wird normalerweise nicht direkt der öffentliche Schlüssel zum verschlüsseln der Daten verwendet, da asymmetrische Verschlüsselungsverfahren im Gegensatz zu symmetrischen Verfahren bekannterweise relativ langsam sind. Deshalb wird in diesem Fall ein zusätzlicher symmetrischer Schlüssel zum Verschlüsseln der Daten generiert. Dieser frisch erzeugte symmetrische Schlüssel, welcher wiederum mit dem öffentlichen Schlüssel des Empfängers verschlüsselt wird, kann in diesem Fall im <EncryptedKey> Element transportiert werden. Ebenfalls besteht die Möglichkeit, falls von vornherein ausgetauschte Schlüssel (pre-shared keys) zum Einsatz kommen, den verwendeten Schlüssel mit dem <KeyInfo> Element zu benennen. In diesem Fall wird der Schlüssel nicht mit Übertragen, sondern nur eindeutig benannt, damit der Empfänger weiss, welcher Schlüssel benötigt wird um die Daten entschlüsseln zu können. Als letztes folgt das Element <CipherData> mit den verschlüsselten Daten. Die verschlüsselten Daten werden hier entweder direkt, als Base64 kodierten Text, im <CipherValue> Element abgelegt, oder mittels <CipherReference> referenziert.

Granularität Die Verschlüsselung kann auf verschiedenen Ebenen im XML Dokument angewendet werden. Jede Art der Verschlüsselung erzeugt wieder ein wohlgeformtes XML. Die originalen Daten werden an der selben Stelle durch ihre verschlüsselten ersetzt.

Im Standard sind die folgenden fünf Arten beschrieben wie XML verschlüsselt werden kann:

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <Kreditkarte>
    <Nummer>1234</Nummer>
    <Aussteller>Institut $$$</Aussteller>
    <Ablaufdatum>05.05.2012</Ablaufdatum>
  </Kreditkarte>
</Zahlungsinformationen>
```

1. XML Element

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>Z63Y21XW</CipherValue>
    </CipherData>
  </EncryptedData>
</Zahlungsinformationen>
```

   In diesem Beispiel wurde das ganze `<Kreditkarte>` Element verschlüsselt. Ein möglicher Mitleser weiss somit nicht, dass eine Kreditkarte verwendet wurde. Da eine Element-Verschlüsselung stattgefunden hat, wird der Wert des Attributs `Type` mit `http://www.w3.org/2001/04/xmlenc#Element` angegeben.

2. Inhalt eines XML Elements (Kind-Elemente und Text)

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <Kreditkarte>
    <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
      Type='http://www.w3.org/2001/04/xmlenc#Content'>
      <CipherData>
        <CipherValue>Z63Y21XW</CipherValue>
      </CipherData>
    </EncryptedData>
  </Kreditkarte>
</Zahlungsinformationen>
```

   Hier wurde der komplette Inhalt des `<Kreditkarte>` Elements verschlüsselt. Der Inhalt kann dabei wiederum aus Element sowie aus Text-Knoten bestehen. Das Attribut `Type` erhält den Wert `http://www.w3.org/2001/04/xmlenc#Content` da eine “Content Encryption” stattgefunden hat.

3. Inhalt eines XML Elements (Text)

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <Kreditkarte>
    <Nummer>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>Z63Y21XW</CipherValue>
        </CipherData>
      </EncryptedData>
    </Nummer>
    <Aussteller>Institut $$$</Aussteller>
    <Ablaufdatum>05.05.2012</Ablaufdatum>
  </Kreditkarte>
</Zahlungsinformationen>
```

In diesem Beispiel wurde nur der Inhalt vom <Nummer> Element verschlüsselt, welcher nur aus Text Knoten bestand und keine Elemente besass. Hier wurde ebenfalls wieder eine Inhalts-Verschlüsselung vorgenommen. Deshalb wird auch hier wieder der Wert des Attributs Type auf http://www.w3.org/2001/04/xmlenc#Content gesetzt.

4. Beliebige Daten

```
<?xml version='1.0'?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
  MimeType='text/xml'>
  <CipherData>
    <CipherValue>Z63Y21XW</CipherValue>
  </CipherData>
</EncryptedData>
```

Ebenfalls ist es möglich ganze Dokumente zu Verschlüsseln. Diese Art der Verschlüsselung beschränkt sich dabei nicht nur auf XML Dokumente, sondern auf beliebige Daten.

5. Super Encryption

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <EncryptedData Id='ED1' xmlns='http://www.w3.org/2001/04/xmlenc#
      '
    Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>urspr ngliche verschl sselte Daten</
          CipherValue>
    </CipherData>
  </EncryptedData>
</Zahlungsinformationen>

<?xml version='1.0'?>
<Zahlungsinformationen>
  <EncryptedData Id='ED2' xmlns='http://www.w3.org/2001/04/xmlenc#
      '
    Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>erneut verschl sselte Daten</CipherValue>
    </CipherData>
  </EncryptedData>
</Zahlungsinformationen>
```

Wurde bereits ein Teil des Dokuments verschlüsselt, kann dieser Teil gegebenenfalls mit einem anderen Schlüssel erneut verschlüsselt werden. Diese Art der Verschlüsselung wird “Super-Encryption” genannt. Super-Encryption kann zum Beispiel dann sinnvoll sein, wenn ein Dokument an verschiedene Empfänger gelangen muss, aber jeder Empfänger nur einen bestimmten Ausschnitt lesen darf. In diesem Beispiel wurde das <EncryptedData> Element mit der Id='ED1' erneut verschlüsselt und als Inhalt von <CipherValue> bei “erneut verschlüsselte Daten” abgelegt.

2.3 Web Services

Web Services ist ein vom W3C [W3Ca] entwickelter Standard [HBN+04] um die Interoperabilität zwischen verschiedenen Software Applikationen, Plattformen und Frameworks sicher zu stellen. Ein Web Service ist ein Software System für die interoperable Maschine zu Maschine Kommunikation über ein Netzwerk. Die im Folgenden beschriebenen Technologien bilden das Grundgerüst und/oder erweitern Web Services um zusätzliche Funktionalitäten.

2.3.1 SOAP

SOAP [BEK+00] ist das meistgenutzte Protokoll um Nachrichten zwischen Web Services auszutauschen. Eine SOAP-Nachricht ist selbst ein XML Dokument welches die folgende Struktur besitzt:

```
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
  </env:Header>
  <env:Body>
  </env:Body>
</env:Envelope>
```

Listing 2.4: Struktur einer SOAP-Nachricht

Das `<env:Header>` Element ist optional und kann zur Unterbringung von Meta-Informationen wie zum Beispiel Routing, Verschlüsselung oder Transaktionsidentifizierung verwendet werden. `<env:Body>` enthält die Nutzdaten, welche im XML Format vorliegen müssen.

2.3.2 Web Services Description Language (WSDL)

In einem WSDL [CCMW01] Dokument werden die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle eines Web Services beschrieben. Durch ein WSDL wird bestimmt, auf welche Art und Weise ein Client auf den Web Service zugreifen kann. Es besteht aus den folgenden Haupt-Elementen, durch die ein Service definiert wird:

- types - Container für die Datentypen, welche die Nachrichten beschreiben die ausgetauscht werden.
- message - Abstrakte Definitionen der Daten die ausgetauscht werden. Eine Nachricht besteht aus einem oder mehreren logischen Teilen, die jeweils mit einer Datentyp-Definition verknüpft sind.
- portType - Abstrakte Menge von Operationen, die der Service anbietet. Jede Operation verweist auf eine Input-Message und Output-Message.
- binding - Spezifiziert das konkrete Protokoll und das Datenformat für die Operationen und Nachrichten eines portType.

- port - Spezifiziert die Adresse des Services für ein binding.
- service - Zusammenfassung von zusammengehörenden ports.

Listing 2.5 zeigt ein WSDL Beispiel (aus [CCMW01]) mit allen genannten Elementen.

```
<?xml version="1.0"?>
<definitions name="StockQuote" targetNamespace="http://example.com/
   stockquote.wsdl"
            xmlns:tns="http://example.com/stockquote.wsdl" xmlns:xsd1=
               "http://example.com/stockquote.xsd" ...>
   <types>
      <schema targetNamespace="http://example.com/stockquote.xsd">
         <element name="TradePriceRequest"> ... </element>
         <element name="TradePrice"> ... </element>
      </schema>
   </types>
   <message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:TradePriceRequest"/>
   </message>
   <message name="GetLastTradePriceOutput">
      <part name="body" element="xsd1:TradePrice"/>
   </message>
   <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
         <input message="tns:GetLastTradePriceInput"/>
         <output message="tns:GetLastTradePriceOutput"/>
      </operation>
    </portType>
    <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType"
       >
      <soap:binding style="document" transport="http://schemas.xmlsoap.
         org/soap/http"/>
      <operation name="GetLastTradePrice">
         <soap:operation soapAction="http://example.com/
            GetLastTradePrice"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
   </binding>
   <service name="StockQuoteService">
      <port name="StockQuotePort" binding="tns:StockQuoteBinding">
         <soap:address location="http://example.com/stockquote"/>
      </port>
   </service>
</definitions>
```

Listing 2.5: Beispiel WSDL

2.3.3 Web Services Security

Web Services Security [LK06] ist ein weiterer Standard, der OASIS [OAS] herausgegeben hat. Der Standard bietet folgende drei Hauptmechanismen:

- Übertragung der Security-Token als Teil der Nachricht
- Integrität der Nachricht
- Vertraulichkeit der Nachricht

Wie es sich vermuten lässt, wird die Sicherung nicht auf Transport, sondern auf Nachrichten-Ebene vorgenommen. Dabei ist der Standard so ausgelegt, dass eine Vielzahl von Security-Token Formate, Signatur Formate und Verschlüsselungstechnologien verwendet werden können. Web Services Security baut auf den bestehenden Standards von XML Encryption und XML Signature auf um Vertraulichkeit und Integrität zu erreichen. Eine typische, mit WS-Security gesicherte, SOAP-Nachricht zeigt Listing 2.6. Im Beispiel sind die erwähnten Mechanismen wiederzufinden, die Vertraulichkeit (`<xenc:EncryptedData>`), die Integrität (`<ds:Signature>`) und ein Security-Token (`<wsse:BinarySecurityToken>`).

```
<env:Envelope xmlns:env="..." xmlns:xenc="...">
   <env:Header>
      <wsse:Security xmlns:wsse="..." env:mustUnderstand="1">
         <xenc:EncryptedKey Id="EncKeyId-1">
            ...
         </xenc:EncryptedKey>
         <wsse:BinarySecurityToken wsu:Id="CertId-1" ...>
            ...
         </wsse:BinarySecurityToken>
         <ds:Signature xmlns:ds="...">
            ...
         </ds:Signature>
      </wsse:Security>
   </env:Header>
   <env:Body xmlns:wsu="..." wsu:Id="id-1">
      <xenc:EncryptedData Id="EncDataId-1" Type="http://www.w3.org
         /2001/04/xmlenc#Content">
         ...
      </xenc:EncryptedData>
   </env:Body>
</env:Envelope>
```

Listing 2.6: Struktur einer mit WS-Security gesicherten SOAP-Nachricht

XML Encryption in WS-Security

XML Encryption in WS-Security ist der erste grosse Teil um den es sich bei dieser Arbeit handelt. Deshalb wird hier detaillierter auf die Einzelheiten von verschlüsselten SOAP-Nachrichten eingegangen.

Wie bereits im Abschnitt über Web Service Security beschrieben, baut WS-Security auf dem XML Encryption Standard [IDS02] auf und erweitert diesen, um die benötigte Flexibilität

im SOAP Umfeld zu erreichen.

```
1  <env:Envelope xmlns:env="..." xmlns:xenc="...">
2     <env:Header>
3        <wsse:Security xmlns:wsse="..." env:mustUnderstand="1">
4           <xenc:EncryptedKey Id="EncKeyId-1" >
5              <xenc:EncryptionMethod Algorithm="http://www.w3.org
                  /2001/04/xmlenc#rsa-1_5"/>
6              <ds:KeyInfo xmlns:ds="...">
7                 <wsse:SecurityTokenReference>
8                    <ds:X509Data>
9                       <ds:X509IssuerSerial>
10                         <ds:X509IssuerName>CN=example,C=com</
                              ds:X509IssuerName>
11                         <ds:X509SerialNumber>1234</ds:X509SerialNumber>
12                      </ds:X509IssuerSerial>
13                   </ds:X509Data>
14                </wsse:SecurityTokenReference>
15             </ds:KeyInfo>
16             <xenc:CipherData>
17                <xenc:CipherValue>
18                   V9Gy+PI0 ...
19                </xenc:CipherValue>
20             </xenc:CipherData>
21             <xenc:ReferenceList>
22                <xenc:DataReference URI="#EncDataId-1"/>
23             </xenc:ReferenceList>
24          </xenc:EncryptedKey>
25       </wsse:Security>
26    </env:Header>
27    <env:Body xmlns:wsu="...">
28       <xenc:EncryptedData Id="EncDataId-1" Type="http://www.w3.org
             /2001/04/xmlenc#Content">
29          <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/
               xmlenc#aes128-cbc"/>
30          <xenc:CipherData>
31             <xenc:CipherValue>
32                qsvxzDEz831z4sxRQ ...
33             </xenc:CipherValue>
34          </xenc:CipherData>
35       </xenc:EncryptedData>
36    </env:Body>
37 </env:Envelope>
```

Listing 2.7: Aufbau einer verschlüsselten SOAP-Nachricht

Listing 2.7 zeigt eine typische Ausprägung einer verschlüsselten SOAP-Nachricht. Hier sei erwähnt, dass eine Inhalts-Verschlüsselung (Content-Encryption) des `<env:Body>` stattgefunden hat. Wäre eine Element Encryption über `<env:Body>` vorgenommen worden, hätten wir keine gültige SOAP-Nachricht mehr vorliegen. Dies wird auch in der Nachricht durch das Attribut `Type` auf dem Element `<xenc:EncryptedData>` wiedergegeben. Die vorliegende Nachricht wurde mit einem symmetrischen Schlüssel verschlüsselt (Session-Key, Zeilen 17-19), welcher wiederum mit dem öffentlichen Schlüssel des Empfängers (Zeilen 7-14) verschlüsselt worden ist (Hybride Verschlüsselung). Mit der Deklaration von `<xenc:EncryptionMethod>` (Zeile 5) wird der Algorithmus angegeben, der verwendet wurde um den symmetrischen Key zu verschlüsseln, und nicht der Algorithmus zum Entschlüsseln der Nutzdaten. Die `<xenc:EncryptedKey>` Struktur enthält eine

<xenc:ReferenceList> (Zeilen 21-23), welche auf die Daten verweist, die mit diesem Schlüssel verschlüsselt wurden. Wird nun der
<xenc:DataReference URI="#EncDataId-1"/> gefolgt, kommt man zum verschlüsselten Teil der Nachricht (Zeilen 28-35). Hier wird durch Angabe von
<xenc:EncryptionMethod> (Zeile 29) der Algorithmus angezeigt der verwendet worden ist um die Nutzdaten in <xenc:CipherValue> (Zeilen 31-33) zu verschlüsseln.

Eine alternative Methode, wenn Sender und Empfänger "pre-shared keys" verwenden, zeigt Listing 2.8. In diesem Beispiel wird nur die <xenc:ReferenceList> im Security Header spezifiziert. Die Informationen zum verwendeten Schlüssel finden sich direkt im
<xenc:EncryptedData> Block. Dies genügt bereits um eine Nachricht erfolgreich verschlüsselt austauschen zu können.

```
<env:Envelope xmlns:env="..." xmlns:xenc="...">
   <env:Header>
      <wsse:Security xmlns:wsse="..." env:mustUnderstand="1">
         <xenc:ReferenceList>
            <xenc:DataReference URI="#EncDataId-1"/>
         </xenc:ReferenceList>
      </wsse:Security>
   </env:Header>
   <env:Body xmlns:wsu="...">
      <xenc:EncryptedData Id="EncDataId-1" Type="http://www.w3.org
         /2001/04/xmlenc#Content">
         <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/
            xmlenc#aes128-cbc"/>
         <ds:KeyInfo xmlns:ds="...">
            <ds:KeyName>CN=example,C=com</ds:KeyName>
         </ds:KeyInfo>
         <xenc:CipherData>
            <xenc:CipherValue>
               qsvxzDEz831z4sxRQ ...
            </xenc:CipherValue>
         </xenc:CipherData>
      </xenc:EncryptedData>
   </env:Body>
</env:Envelope>
```

Listing 2.8: Referenzierung von pre-shared-keys

2.3.4 Web Services Policy Framework

Mit Hilfe des Web Service Policy Frameworks [YHV+07] lassen sich Richtlinien bezüglich Sicherheit, Qualität, usw in einem Web Service Umfeld beschreiben und durchsetzen. Das Framework definiert ein allgemeines Modell und ein Syntax um Richtlinien (Policies) beschreiben zu können. In dieser Spezifikation sind daher keine Bedingungen (Assertions) definiert.

Aufbau Policies sind in der Normalform folgendermassen aufgebaut:

```
<wsp:Policy>
    <wsp:ExactlyOne>
```

```
        <wsp:All>
            <!--Liste von Assertions-->
        </wsp:All>
        <wsp:All>
            <!--Liste von Assertions-->
        </wsp:All>
        <!--beliebig viele weitere Elemente vom Typ wsp:All-->
    </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 2.9: Struktur einer Policy

Eine Policy beginnt immer mit dem Element `<wsp:Policy>`. Mit dem Element `<wsp:ExactlyOne>` wird eine Liste mit Policy Alternativen eingeleitet. Wobei ein `<wsp:All>` eine solche Alternative repräsentiert. Um eine gesamte Policy zu erfüllen, muss genau eine Alternative erfüllt sein. Eine Alternative ist erfüllt, wenn alle Assertions in einem `<wsp:All>` erfüllt sind. Die Elemente `<wsp:ExactlyOne>` und `<wsp:All>` können auch weiter in sich verschachtelt werden.

2.3.5 Web Services Policy Attachment

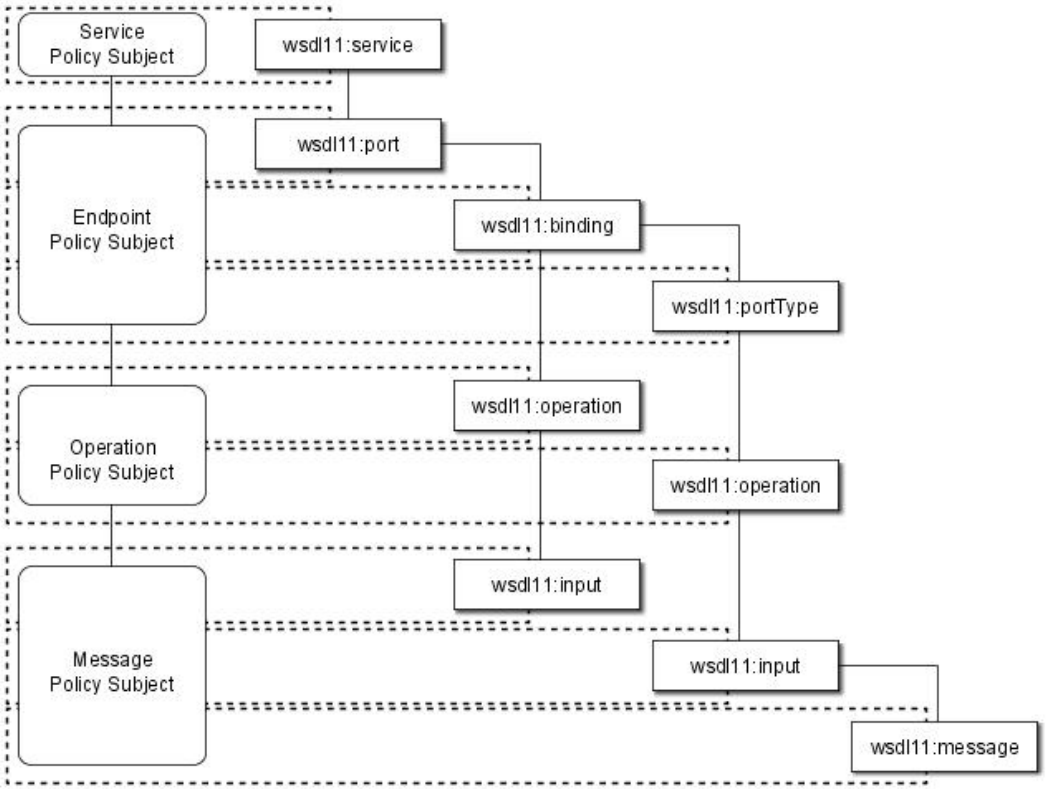


Abbildung 2.2: Policy Subjekte und Bereiche im WSDL (aus [VYO+07])

Das Web Service Policy Framework beschreibt wie man Policies ausdrücken kann. Nun gilt es, diese mit den Policy Subjekten (endpoint, message, resource, operation) eines Web Services zu verknüpfen. Genau dies ist die Aufgabe des Web Services Policy Attachment Standards

[VYO+07]. Der Standard beschreibt, wo man die Policies in einem WSDL anwenden kann [Abbildung 2.2].

Aus der Abbildung ist ersichtlich, dass es für eine eingehende Nachricht insgesamt 9 mögliche Punkte im WSDL gibt, wo Policies eingefügt werden können. So ist es möglich, Policies zu definieren, die für den ganzen Service gelten, oder aber auch nur für eine einzelne Nachricht. Werden gleichzeitig an mehreren Stellen im WSDL Policies definiert, müssen diese zu einer sogenannten **effektiven Policy** zusammengeführt werden (Merge). Die effektive Policy enthält alle Assertions die erfüllt werden müssen.

2.3.6 Web Services SecurityPolicy

Der Web Services SecurityPolicy Standard [NGG+09] definiert Policy Assertions im Bezug zu den Sicherheitsfunktionen die SOAP Message Security [LK06], WS-Trust [NGG+07b] und WS-SecureConversation [NGG+07a] bieten. Mit diesen Assertions kann man beschreiben, wie die Nachrichten gesichert werden müssen.

Web Services SecurityPolicy ist die Grundlage des zweiten Teils dieser Arbeit.

Die wichtigsten Assertion Kategorien sind:

- **Protection Assertions** bestimmen welche Elemente signiert, verschlüsselt oder einfach nur vorhanden sein müssen.
- **Token Assertions** beschreiben die Art der Tokens (Username, X509, Kerberos, SAML, et cetera), die vom System verwendet und allenfalls beim Nachrichtenaustausch mitgegeben werden müssen.
- **Security Binding Assertions** bestimmen ob die Sicherung der Nachricht auf Transport- oder Nachrichten- Ebene und mit welchen elementaren Sicherheitseinstellungen dies geschehen soll (Algorithmen, Zeitstempel, et cetera).
- **Supporting Token Assertions** beschreiben die zusätzlichen "unterstützenden" Tokens die in einer Nachricht erwartet werden. Dies kann zum Beispiel ein zusätzliches Token für die Benutzeranmeldung sein (UsernameToken).

Das Beispiel in Listing 2.10 zeigt eine Policy mit einem **AsymmetricBinding**, was bedeutet, dass asymmetrische Schlüssel (Public-Key Verfahren) zum Schutz der Nachricht angewendet werden müssen. Mit dem **InitiatorToken** wird ausgedrückt, welche Anforderungen an das Initiator-Token bestehen. In diesem Beispiel muss in jeder Anfrage ein X509 Zertifikat in der Version 3 mitgegeben werden. Das Initiator-Token wird für die Signatur vom Sender zum Empfänger und für die Verschlüsselung vom Empfänger zum Sender benutzt. Dem gleichen Zweck dient das **RecipientToken**, nur dass dieses Token zum Verschlüsseln vom Sender zum Empfänger und für die Signatur vom Empfänger zum Sender benutzt wird. Mit der **AlgorithmSuite** wird festgelegt, welche Algorithmen eingesetzt werden müssen. Die im Beispiel verwendete Base256 Suite besagt unter anderem, dass als Digest Sha1, für die Verschlüsselung Aes256, usw. verwendet werden muss. Die **EncryptedParts**, **EncryptedElements** und **ContentEncryptedElements** Assertions bestimmen welche Teile, Elemente und/oder

```
<wsp:Policy wsu:Id="SamplePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
          xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/
             securitypolicy">
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://schemas.xmlsoap.org/ws
                     /2005/07/securitypolicy/IncludeToken/
                     AlwaysToRecipient">
                <wsp:Policy>
                  <sp:WssX509V3Token11/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token
                  sp:IncludeToken="http://schemas.xmlsoap.org/ws
                     /2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token11/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:EncryptedParts>
        <sp:Body/>
      </sp:EncryptedParts>
      <sp:EncryptedElements>
        <sp:XPath xmlns:ex="http://example.com">//ex:Element</sp:XPath>
      </sp:EncryptedElements>
      <sp:ContentEncryptedElements>
        <sp:XPath xmlns:ex="http://example.com">//ex:Content</sp:XPath>
      </sp:ContentEncryptedElements>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 2.10: Beispiel WS-SecurityPolicy

Inhalte einer Nachricht verschlüsselt sein müssen.

2.4 Angriffsvektoren auf Web Services

Wie bereits gesehen, bieten Web Services eine plattformübergreifende Möglichkeit strukturierte Daten auszutauschen. Leider ergeben sich durch den Einsatz von Web Services auch neue Angriffsflächen, die im Folgenden besprochen werden. Es wird auf Angriffe eingegangen, die direkt auf das Security-Framework zielen. Andere Angriffe wie SQL, LDAP, XPath Injection werden in dieser Arbeit nicht besprochen.

Es soll hier nicht vorenthalten werden, dass das W3C ein Dokument mit dem Titel "XML Signature Best Practices" [HD10] veröffentlicht hat, das beim Implementieren eines Sicheren WS-Security Frameworks hilfreich sein kann.

2.4.1 Allgemeine DoS Attacken

Denial-of-Service Attacken zielen darauf ab, einen Dienst lahm zu legen. Im Speziellen bei Web Services sind folgende DoS Attacken denkbar, wie:

Senden von

- SOAP-Dokumenten mit sehr vielen XML Elementen, so dass der freie Arbeitsspeicher vollständig aufgebraucht wird.
- SOAP-Dokumenten mit sehr tiefer Struktur, ebenfalls bis der freie Arbeitsspeicher vollständig aufgebraucht wird.
- vielen signierten oder verschlüsselten SOAP-Dokumenten, welches den Web Services mit teuren kryptographischen Berechnungen auslastet/überlastet.
- verschlüsselten Dokumenten, welche im verschlüsselten Teil wiederum sehr viele Elemente oder tief verschachtelte Elemente enthält.

Begünstigt werden solche Angriffe durch den Einsatz von DOM Parsern, welche zuerst das ganze XML Dokument in den Arbeitsspeicher einlesen. Dies ist klassischerweise spätestens dann der Fall, wenn WS-Security zum Einsatz kommt.

2.4.2 XML Element Wrapping Attacken

Diese Art von Attacken basiert auf der Grundlage das XML Dokumente relativ frei bezüglich Inhalt sind. So lassen sich zum Beispiel signierte oder verschlüsselte XML Elemente im Dokument an eine andere Stelle kopieren oder sogar in andere Dokumente verschieben, ohne dass die Signatur zerstört wird und das verschlüsselte Element sich trotzdem entschlüsseln

lässt. XML Element Wrapping Attacken sind in [MA05] genau beschrieben. Eine Ausprägung des Szenarios in Bezug auf die Verschlüsselung ist zum Beispiel:

```
<env:Envelope>
  <env:Header>
     <wsse:Security>
        ...
        <xenc:EncryptedKey
           Id="EncKeyId-1">
           ...
        </xenc:EncryptedKey>
        <xenc:ReferenceList>
           <xenc:DataReference
              URI="#EncDataId-1"/>
        </xenc:ReferenceList>
     </wsse:Security>
  </env:Header>
  <env:Body wsu:Id="id-1">
     <xenc:EncryptedData
        Id="EncDataId-1">
        <!--
           <Bestellung>
              <Name>
                Hans Muster
              </Name>
              <Artikel1>
                 <Preis>
                   1500.00
                 </Preis>
              </Artikel1>
              <Kreditkarte>
                 123456-098
              </Kreditkarte>
           </Bestellung>
        -->
     </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

```
<env:Envelope>
  <env:Header>
     <wsse:Security>
        ...
        <xenc:EncryptedKey
           Id="EncKeyId-1">
           ...
        </xenc:EncryptedKey>
        <xenc:ReferenceList>
           <xenc:DataReference
              URI="#EncDataId-1"/>
        </xenc:ReferenceList>
     </wsse:Security>
     <Wrapper
        soap:mustUnderstand="0"
        soap:role=".../none">
        <env:Body wsu:Id="theBody">
           <xenc:EncryptedData
              Id="EncDataId-1">
              <!--
                 <Bestellung>
                    <Name>
                      Hans Muster
                    </Name>
                    <Artikel1>
                       <Preis>
                         1500.00
                       </Preis>
                    </Artikel1>
                    <Kreditkarte>
                       123456-098
                    </Kreditkarte>
                 </Bestellung>
              -->
           </xenc:EncryptedData>
        </env:Body>
     </Wrapper>
  </env:Header>
  <env:Body wsu:Id="newBody">
     <Bestellung>
        <Name>Hans Muster</Name>
        <Artikel1>
           <Preis>500.00</Preis>
        </Artikel1>
        <Kreditkarte>123456-098</Kreditkarte>
     </Bestellung>
  </env:Body>
</env:Envelope>
```

Stellen wir uns einen Online-Shop vor, bei dem sich der Benutzer anmelden muss. Der Kunde bestellt einen teuren Artikel. Der Shop kommuniziert mit dem Zahlungssystem über SOAP-Nachrichten bei denen der SOAP-Body verschlüsselt wird (linkes Beispiel). Das Zahlungssystem rechnet automatisch ab und löst die Bestellung aus. Schafft es der Kunde die SOAP-Nachricht abzufangen, kann er nun den verschlüsselten SOAP-Body “wrappen” und in

den SOAP-Header verschieben (rechtes Beispiel). Danach hängt er einen neuen SOAP-Body an, wo er den Preis des Artikels reduziert. Der neue SOAP-Body muss nicht mal verschlüsselt werden. Das Zahlungssystem wird diese Nachricht akzeptieren, wenn keine speziellen Massnahmen vorgenommen wurden, da die Referenzen zu den verschlüsselten Daten immer noch eindeutig auflösbar sind.

2.4.3 Rekursive Schlüssel Referenzen

Dieses Angriffsszenario gehört zu den DoS Attacken. Wie es der Name schon sagt, werden die Schlüssel rekursiv referenziert [IDS02, Kap. 6.4]

```
<xenc:EncryptedKey Id='Key1'>
    <xenc:EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#
       aes128-cbc'/>
    <ds:KeyInfo>
        <ds:RetrievalMethod URI='#Key2' Type="http://www.w3.org
           /2001/04/xmlenc#EncryptedKey"/>
        <ds:KeyName>Key2</ds:KeyName>
    </ds:KeyInfo>
    <xenc:CipherData>
        <xenc:CipherValue>ABC</xenc:CipherValue>
    </xenc:CipherData>
    <xenc:ReferenceList>
        <xenc:DataReference URI='#EncDataId-1'/>
    </xenc:ReferenceList>
    <xenc:CarriedKeyName>Key1</xenc:CarriedKeyName>
</xenc:EncryptedKey>
<xenc:EncryptedKey Id='Key2'>
    <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#
       aes128-cbc"/>
    <ds:KeyInfo>
        <ds:RetrievalMethod URI='#Key1' Type="http://www.w3.org
           /2001/04/xmlenc#EncryptedKey"/>
        <ds:KeyName>Key1</ds:KeyName>
    </ds:KeyInfo>
    <xenc:CipherData>
        <xenc:CipherValue>CBA</xenc:CipherValue>
    </xenc:CipherData>
    <xenc:ReferenceList>
        <xenc:DataReference URI='#EncDataId-2'/>
    </xenc:ReferenceList>
    <xenc:CarriedKeyName>Key2</xenc:CarriedKeyName>
</xenc:EncryptedKey>
...
<xenc:EncryptedData Id="EncDataId-2">
    ...
    <ds:KeyInfo>
        <wsse:SecurityTokenReference>
            <wsse:Reference URI="#EncKeyId-408
               A76AB670A19739C12884354753845"/>
        </wsse:SecurityTokenReference>
    </ds:KeyInfo>
    ...
```

```
</xenc:EncryptedData>
```

Listing 2.11: Rekursive Schlüssel Referenzen

Listing 2.11 zeigt eine solche Attacke. Das XML enthält zwei <EncryptedKey> Blöcke mit Keys die gegenseitig verschlüsselt wurden. Durch die gegenseitige Referenzierung mittels dem optionalen Element <RetrievalMethod> wird eine endlose Rekursion angestossen. Weitere Beispiele dieses Angriffes findet man in [Bid09] und [San02]

2.4.4 Externe URI Referenzen

Die Spezifikation [LK06] sieht vor, dass Referenzen zu Schlüsseln und Daten per "URI" Attribut referenziert werden. Die URI ist typischerweise eine Referenz, die im selben Dokument aufgelöst werden kann. Es ist aber auch vorgesehen, dass die referenzierten Ressourcen ausserhalb des Dokumentes liegen können. Dazu kann im URI Attribut eine absolute URI wie zum Beispiel "http://key-server/Key" angegeben werden:

Folgendes Beispiel demonstriert dies:

```
<ds:KeyInfo>
   <wsse:SecurityTokenReference>
      <wsse:Reference URI="http://www.kernel.org/pub/linux/kernel/v2.6/
         linux-2.6.23.tar.gz"/>
   </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Listing 2.12: Externe Token Referenz

Ein Angreifer hat nun die Möglichkeit, eine beliebige URI anzugeben und dadurch eine DoS Attacke zu starten [Listing 2.12]. Mit jeder Anfrage auf eine externe Ressource, werden lokale System Ressourcen benötigt.

2.4.5 Web Services und DTD Verarbeitung

In XML Dokumenten ist die Möglichkeit vorhanden eine sogenannte "Document Type Definition" anzugeben. Dadurch wird dem XML Parser mitgeteilt, wo er die Grammatik zum Dokument finden kann. Die DTD ist der Vorläufer zum heute bekannteren XML-Schema. Details zu DTD's und deren Benutzung findet man in [BPSM+08]. Die SOAP-Spezifikation [BEK+00, Kap. 3] sagt zu DTD's folgendes aus:
"A SOAP message MUST NOT contain a Document Type Declaration. A SOAP message MUST NOT contain Processing Instructions."

Trotzdem ist es im Jahr 2010 zu "Zwischenfällen" [1] [2] auf Web Service Frameworks gekommen.

[1] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1632
[2] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2076

Diese DTD Angriffe sind für die Arbeit insofern relevant, da das entschlüsselte XML wiederum eine DTD Deklaration enthalten könnte.

Durch die Angabe einer DTD im SOAP-Dokument lassen sich nun unter anderem diese Angriffe gegen einen Web Service starten [Her10]:

DTD Public Identifier DoS

Dieses Szenario ist die gleiche Problematik wie sie bereits im Abschnitt 2.4.4 besprochen wurde. Der Unterschied ist der, dass wir uns hier auf der XML-Parser Ebene befinden und nicht bereits schon im Web Service Security Stack. Durch die Angabe einer PUBLIC Identifier URI wird der Parser dazu angewiesen, die DTD von diesem Ort herunterzuladen um das XML Validieren zu können. Dies führt wiederum zu einer möglichen DoS Attacke [Listing 2.13].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE abbrev PUBLIC \"fsafasdf\" \"http://www.kernel.org/pub/linux/
    kernel/v2.6/linux-2.6.23.tar.gz\">"
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    ...
```

Listing 2.13: DTD Public Identifier DoS

Unrestricted Recursive Entity References in DTDs ('XML Bomb', 'Billion Laughs Attack')

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foobar [
        <!ENTITY x0 "A">
        <!ENTITY x1 "&x0;&x0;">
        <!ENTITY x2 "&x1;&x1;">
        <!ENTITY x3 "&x2;&x2;">
        ...
        <!ENTITY x63 "&x62;&x62;">
        <!ENTITY x64 "&x63;&x63;">
]>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
    ...
    <bye>&x64;</bye>
    ...
```

Listing 2.14: Unrestricted Recursive Entity References in DTDs ('XML Bomb', 'Billion Laughs Attack')

Löst nun der XML-Parser die Entities rekursiv auf, wird die Datenmenge explosionsartig vervielfältigt. Beim ersten Durchlauf wird &x64; zu &x63;&x63; aufgelöst. Beim darauffolgenden zu &x62;&x62;&x62;&x62; usw, welches folgende Sequenz ergibt:

2^0 = &x64;
2^1 = &x63;&x63;

2^2 = &x62;&x62;&x62;&x62;
2^3 = &x61;&x61;&x61;&x61;&x61;&x61;&x61;&x61;
2^4 = &x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;&x60;
usw.

Wenn wir nun davon ausgehen, dass wie im Listing 2.14 nur ein Zeichen bei x0 steht, dann hätten wir am Ende der Expandierung eine Zeichenkettenlänge von $2^{64} = 16$ Exabytes erreicht! Dies sind natürlich viel mehr Daten als erwartet und endet in einem effektiven DoS Angriff.

2.4.6 XSL Transform Exploitation

Die XML Encryption [IDS02] und XML Signature [RSH+08] Spezifikationen erlauben an diversen Orten eine XSL Transformation anzuwenden. Sie können dazu verwendet werden um zum Beispiel die signierten Daten in eine Form zu bringen in der die Signatur wieder eindeutig verifiziert werden kann. Im Bezug auf XML Verschlüsselung scheint eine XSL Transformation eher sinnlos zu sein, da die verschlüsselten Daten nicht in XML Form vorliegen. Da aber die XML Encryption Spezifikation die Definition von der XML Signature Spezifikation übernommen hat, ist es trotzdem möglich eine Transformation einzubetten. Folgende relative XPath Ausdrücke zeigen die möglichen Orte auf, wo man eine XSL Transformation anwenden kann:

- ds:KeyInfo/ds:RetrievalMethod/ds:Transforms/ds:Transform
- ds:Signature/ds:SignedInfo/ds:Reference/ds:Transforms/ds:Transform
- xenc:CipherData/xenc:CipherReference/xenc:Transforms/ds:Transform
- xenc:ReferenceList/xenc:DataReference/ds:Transforms/ds:Transform

Wie aus den XPath Ausdrücken ersichtlich ist, verwenden alle die gleiche Transform Definition aus [RSH+08, Kap. 6.6]

```
1 <Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116"
     xmlns="http://www.w3.org/2000/09/xmldsig#">
2     <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL
         /Transform"
3         xmlns:rt="http://xml.apache.org/xalan/java/java.lang.Runtime"
4         xmlns:ob="http://xml.apache.org/xalan/java/java.lang.Object"
5         exclude-result-prefixes="rt ob">
6         <xsl:template match="/">
7             <xsl:variable name="runtimeObject" select="rt:getRuntime()"
                 />
8             <xsl:variable name="command" select="rt:exec($runtimeObject
                 ,&apos;/bin/sh&apos;)"/>
9             <xsl:variable name="commandAsString" select="ob:toString($
                 command)"/>
10            <xsl:value-of select="$commandAsString"/>
11        </xsl:template>
12    </xsl:stylesheet>
13 </Transform>
```

Listing 2.15: XSL Transform Exploitation

Listing 2.15 demonstriert wie eine solche Transformation von einem Angreifer ausgenutzt werden könnte. Für dieses Beispiel wird die Xalan Java class Mapping Erweiterung benutzt, welche erlaubt Java Code aus dem Stylesheet aufzurufen. Auf den Zeilen 3 und 4 werden die benötigten Java Klassen in den XML-Namespace gemappt. Auf der Zeile 7 wird die Instanz der Runtime-Klasse über die statische Methode `getRuntime()` geholt. In der darauffolgenden Zeile 8 wird mit dem Aufruf der `exec()` Methode der System Befehl abgesetzt. Die Zeilen 9 und 10 sind nötig um einige Optimierungen von Xalan zu umgehen, so dass der Befehl auch wirklich abgesetzt wird.

Dieses Beispiel ist harmlos, da nur eine System-Shell geöffnet wird. Das Beispiel könnte aber problemlos erweitert werden, so dass ein beliebiger Systembefehl oder Java Code ausgeführt wird. Ebenfalls wäre es denkbar, dass beliebiger Programm-Code aus dem Internet nachgeladen und ausgeführt wird. Weitere Details findet man in [Bid09] und [Hil07]

2.4.7 SOAPAction spoofing

SOAPAction ist eine URI die vom Client auf der HTTP Ebene als Header mitgegeben wird.

Der Sinn und Zweck ist in der SOAP-Spezifikation [BEK+00, Kap 6.1.1] wie folgt beschrieben: "The SOAPAction HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The presence and content of the SOAPAction header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP."

Listing 2.16 zeigt, wo die SOAPAction im WSDL deklariert wird.

```
<definitions .... >
    <binding .... >
        <operation .... >
           <soap:operation soapAction="uri"? style="rpc|document"?>?
        </operation>
    </binding>
</definitions>
```

Listing 2.16: SOAPAction im WSDL

Der Client ist nun angewiesen, die SOAPAction beim Aufruf des entsprechenden Web Services mitzugeben. Auf der Web Service Server Seite kann dies verwendet werden um die richtige Web Service Implementierung zu finden.

Im Zusammenhang mit WS-Security-Policy kann sich nun folgendes Szenario ergeben, wenn der SOAPAction eines Clients unüberlegt gefolgt wird. Listing 2.17 zeigt einen Ausschnitt aus einem WSDL bei dem die Operation `getA` mit der SOAPAction `getA` die Policy `getAPolicy` referenziert (Zeilen 2-6). Die Operation `getB` referenziert keine Policy (Zeilen 7-10).

```
1 <wsdl:binding ... >
2     <wsdl:operation name="getA">
3         <wsp12:PolicyReference URI="#getAPolicy"/>
4         <soap:operation soapAction="getA" style="document"/>
5         ...
6     </wsdl:operation>
```

```
7       <wsdl:operation name="getB">
8           <soap:operation soapAction="getB" style="document"/>
9           ...
10      </wsdl:operation>
11  </wsdl:binding>
```

Listing 2.17: SOAPAction und SecurityPolicy im WSDL

```
1  POST /webservice HTTP/1.1
2  Content-Type: text/xml
3  SOAPAction: "getB"
4
5  ...
6  <soap:body>
7      <getA>...</getA>
8  </soap:body>
9  ...
```

Listing 2.18: Client SOAPAction spoofing

Macht der Client nun einen SOAP Aufruf wie in Listing 2.18, und der Server führt keine weiteren Überprüfungen bezüglich SOAPAction und der Operation durch, ermöglicht dies dem Client unter Umständen die SecurityPolicy zu umgehen. Auf der Zeile 3 setzt der Client die SOAPAction URL explizit auf `getB`, was auf dem Server dazu führen kann, dass keine Policy angewendet wird, da die Operation `getB` mit der SOAPAction `getB` keine Policy referenziert.

3 Konzept

In diesem Kapitel wird auf die Anforderungen und die Architektur der streaming-basierten SOAP-Message-Ver- und Entschlüsselung und die WSSecurityPolicy Validierung eingegangen.

3.1 Anforderungen und Ziele

Die folgenden Punkte beschreiben die Anforderungen und Ziele dieser Arbeit:

- Ressourcenverbrauch
  Auf dem Thema Ressourcenverbrauch liegt das Hauptaugenmerk dieser Arbeit. Der benötigte Arbeitsspeicher soll möglichst klein gehalten werden. Ebenso soll die Verarbeitung in möglichst kurzer Zeit erfolgen. Oft ist ein Kompromiss zwischen mehr Arbeitsspeicherverbrauch zugunsten von Prozessor-Zeit zu machen. Ein Grundprinzip, das sich durch die ganze Arbeit zieht, ist das sogenannte "fail-fast" Verhalten. Dies bedeutet, dass sobald ein Fehler auftritt, respektiv detektiert wird, die Verarbeitung abgebrochen wird um nicht unnötig Ressourcen zu verschwenden. Im Zusammenhang mit dieser Arbeit werden hier unter Fehlern Zustände verstanden, die nicht durch Programmierfehler hervorgerufen werden, sondern z.B. durch eine WS-SecurityPolicy Verletzung herbeigeführt werden.

- Sicherheit
  Die Sicherheit des Frameworks ist eine grundlegende Anforderung. Die Sicherheit darf nicht zugunsten von Geschwindigkeit geopfert werden. Schliesslich ist es auch die Aufgabe des Frameworks, Sicherheit zu schaffen. Es soll resistent gegen alle bekannten Angriffsvektoren sein.

- Erweiterbarkeit
  Um eine vollständige und standardkonforme Implementierung von SOAP Message Security zu erhalten, werden weitere Komponenten, die nicht Teil dieser Arbeit sind, benötigt. Zu diesen Komponenten zählen unter anderem Signatur-Erstellung/Verifizierung, Timestamp, UsernameToken, et cetera. Dies soll bei der Architektur berücksichtigt werden, so dass diese Komponenten nachgerüstet werden können.
  Ebenfalls sollten zukünftige technologische Entwicklungen und neue Standards auf möglichst einfache Weise in das Framework integriert werden können.

- Integrierbarkeit
  Das Framework sollte flexibel sein, um Adaptierungen in bestehende Systeme so einfach

wie möglich zu gestalten. Oft ist es notwendig, dass das Resultat der SOAP-Message-Security Verarbeitung der Web Service Implementierung zur Verfügung steht. Dies kann zum Beispiel dann der Fall sein, wenn der Web Service als Proxy fungiert und stellvertretend die Security Verifizierung übernimmt.

3.2 Architektur

Um den genannten Anforderungen gerecht zu werden, bedarf es einer soliden und flexiblen Architektur. Im vorherigen Kapitel wurden die verschiedenen Möglichkeiten erläutert wie ein XML Dokument verarbeitet werden kann. Die Wahl viel aus folgenden Gründen auf die StAX-API:

- StAX erlaubt das Lesen sowie Schreiben von XML Dokumenten.
- Die Implementierung kann die XML Elemente vom StAX Parser anfordern und werden nicht durch den Parser "aufgedrängt".
- Nicht zuletzt verwenden bekannte SOAP-Frameworks (Apache Axis2, Apache CXF, et cetera) die StAX API um SOAP-Nachrichten effizient zu verarbeiten.

Da keine Abhängigkeiten zu einem Webservie Framework bestehen, ist auch der eigenständige Betrieb wie in einem XML-Security-Gateway vorstellbar.

3.2.1 Integration

Die Integration in ein Web Service Framework, in diesem Fall Apache CXF [Apa], zeigt Abbildung 3.1. Apache CXF ist vollständig streaming-orientiert aufgebaut. Es verwendet die StAX-API um die SOAP-Nachrichten zu verarbeiten. Sobald aber WS-Security zum Einsatz gelangt, muss der Umweg über DOM genommen werden um die Sicherheitsfunktionen durchzuführen. Dieser Umstand wird mit dieser Arbeit "korrigiert". Apache CXF ist um einen sogenannten "Interceptor-Chain" aufgebaut[1]. Jeder "Interceptor" übernimmt eine bestimmte Funktionalität beim Verarbeiten einer SOAP-Nachricht. Die WS-Security Funktionalität wird einfach durch Hinzufügen eines weiteren Interceptors realisiert. Zu unterscheiden sind hier lediglich der ausgehende und eingehende Interceptor-Chain. Bei der ausgehenden Seite wird ein WS-Security **Out**-Interceptor eingefügt, dessen Aufgabe es ist, das ausgehende Streaming SOAP-Message Security-Framework aufzurufen und per WS-SecurityPolicy zu konfigurieren. Auf der eingehenden Seite wird ein WS-Security **In**-Interceptor eingefügt, der das eingehende Streaming SOAP-Message Security-Framework aufruft und die WS-SecurityPolicy übergibt.

[1] http://cxf.apache.org/docs/cxf-architecture.html

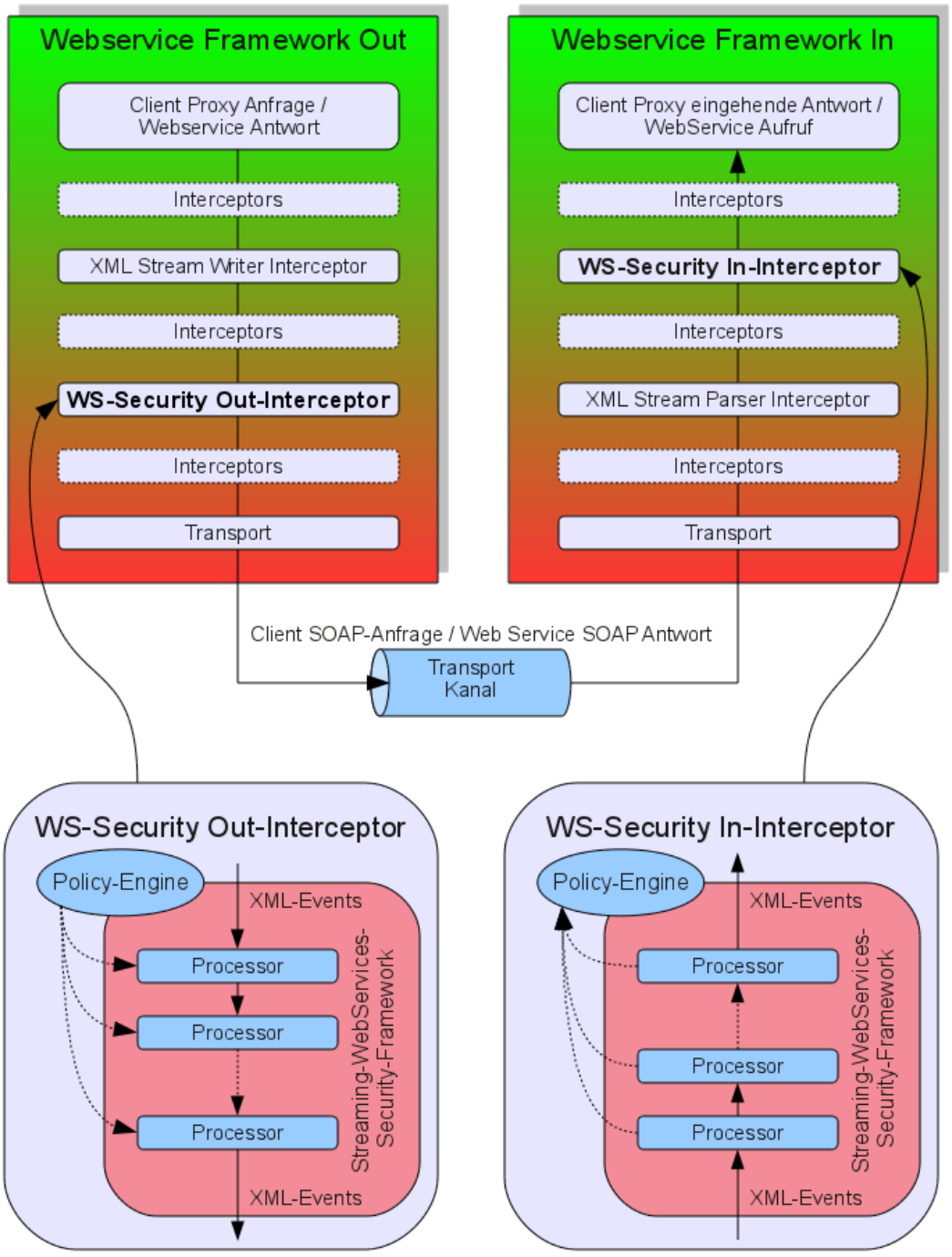


Abbildung 3.1: Web Service Framework Integration

3.2.2 Aufbau

Mit einem Augenmerk auf die Erweiterbarkeit wurde für den Aufbau ein Design gewählt, so dass eine Art Zuständigkeitskette (Chain of Responsibility) entsteht. Jedes Objekt (im folgenden Prozessor genannt) in der Kette ist genau für einen bestimmten Teil der SOAP-Nachricht zuständig. Die Prozessoren werden dynamisch zur Laufzeit hinzugefügt, je nachdem was für Einstellungen auf der Client-Seite vorgenommen wurden oder wie die von der Server-Seite empfangene Nachricht aufgebaut ist. Dieses Design ist in der Abbildung 3.2

ersichtlich.

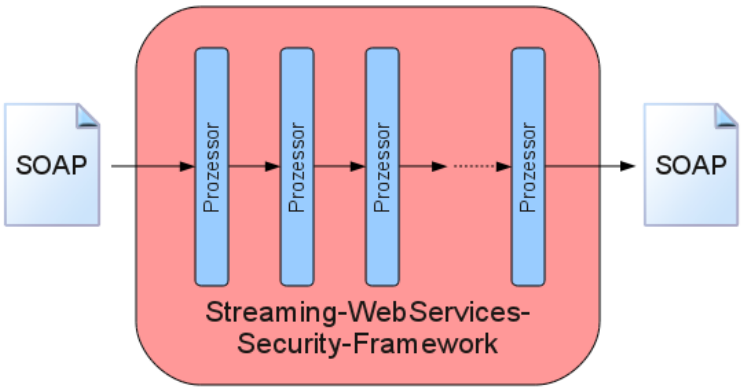


Abbildung 3.2: Streaming-WebServices-Security-Framework Design

Wie bereits gesehen, gibt es bei StAX zwei verschiedene Modelle um XML zu verarbeiten:

Verarbeitung mit dem Cursor-Modell Wird das Cursor-Modell in der Prozessor-Kette angewendet, kann jeder Prozessor direkt auf den Parser zugreifen und so sehr effizient das XML verarbeiten. Die Prozessoren können nach belieben weitere XML Fragmente einlesen. Dass dies ein schlechtes Design ist, zeigen folgende Nachteile:

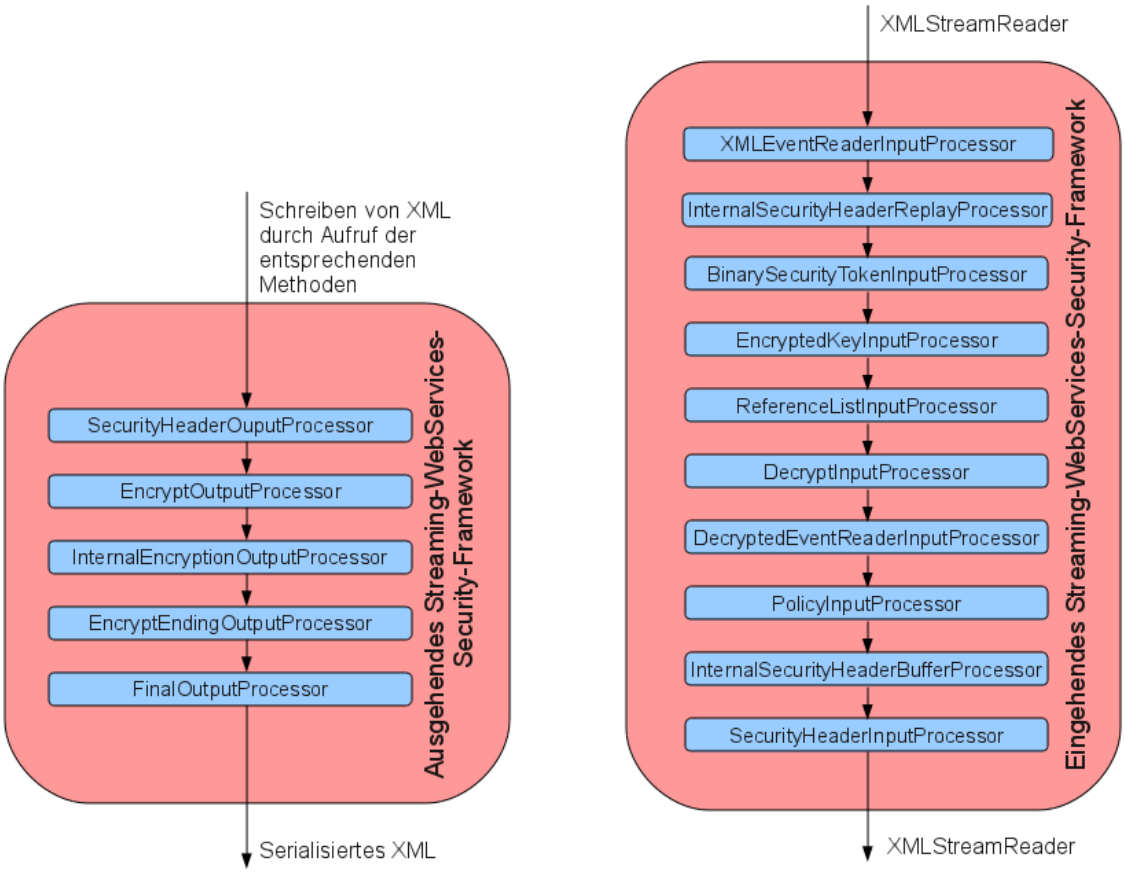
- Jeder Prozessor kann vom Parser weitere XML Teile anfordern. Dies führt dazu, dass die nachfolgenden Prozessoren nicht wissen wo sich nun der Cursor des Parser genau im XML befindet. Jede kleine Änderung in einem Prozessor führt dazu, dass alle nachfolgenden Prozessoren nicht mehr funktionieren.
- Überliest ein Prozessor ein Element, da es für ihn nicht interessant ist, haben auch die nachfolgenden Prozessoren keine Möglichkeit mehr dieses Element zu verarbeiten.
- Es können keine neuen XML Ereignisse eingeschleust werden. Das Erzeugen von neuen XML Elementen ist aber notwendig für das Ver- sowie das Entschlüsseln.

Verarbeitung mit dem Iterator-Modell Alle Nachteile die das Cursor-Modell aufweist sind beim Iterator-Modell nicht vorhanden. Es wird nur das jeweilige aktuelle XMLEvent-Objekt durch die Prozessor-Kette durchgereicht. Eine gegenseitige Beeinflussung des Parser ist daher schon von vornherein ausgeschlossen. Weiterhin besteht die Möglichkeit, XML-Events zu generieren, zu absorbieren, zu ignorieren oder auch zwischenzuspeichern.

3.2.3 Verschlüsselung

Auf der ausgehenden Seite, d.h. beim Verschlüsseln einer SOAP-Nachricht, bildet ein XML-StreamWriter die Schnittstelle zum Streaming-WebServices-Security-Framework. Dieser empfängt die XML Elemente, erzeugt daraus XMLEvent's und übergibt sie der Prozessor-Kette. Im vorliegenden Design kommen folgende Prozessoren zum Einsatz [Abbildung 3.3a]:

- SecurityHeaderOutputProcessor: Baut, falls nicht vorhanden, den `<soap:Header>` auf und fügt das `<wsse:Security>` Element ein.
- EncryptOutputProcessor: Generiert den symmetrischen Schlüssel und sucht nach den zu verschlüsselnden Teilen im Dokument. Sobald ein zu verschlüsselnder Teil gefunden wird, wird eine neue Instanz des InternalEncryptOutputProcessor in die Prozessor-Kette eingefügt.
- InternalEncryptOutputProcesor: Führt die Verschlüsselung des aktuell zu verschlüsselnden Teil-Dokuments durch.
- EncryptEndingOutputProcessor: Puffert die XML-Events bis zum Ende des Dokuments, baut danach den <wsse:Security> Header auf und reicht alle XML Events weiter durch die Prozessor-Kette.
- FinalOutputProcessor: Serialisiert die XML-Events in ein OutputStream.



(a) Ausgehend (b) Eingehend

Abbildung 3.3: Prozessoren des Streaming-WebServices-Security-Frameworks

3.2.4 Entschlüsselung

Die Schnittstelle zum Streaming-WebServices-Security-Framework auf der eingehenden Seite ist über einen XMLStreamReader gelöst. Dieser XMLStreamReader wird in ein XMLEventReader "gewrappt" der anschliessend die XMLEvents erzeugt. Am Ende der Kette steht wiederum ein XMLStreamReader, welcher die XML-Events durch die Kette "zieht" und der umschliessenden Applikation zur Verfügung stellt. Beim Entschlüsseln (eingehende Seite) kommen folgende Prozessoren zum Einsatz [Abbildung 3.3b]:

- XMLEventReaderInputProcessor: Liest angeforderte XML-Events vom darunterliegenden XMLStreamReader und reicht diese weiter.
- InternalSecurityHeaderReplayProcessor: Spielt die vom InternalSecurityHeaderBufferProcessor zwischengespeicherten XMLEvents wieder in die Kette ein.
- BinarySecurityTokenInputProcessor: Liest, falls im Dokument vorhanden, die `<BinarySecurityToken>` Struktur ein, und stellt das Token dem DecryptInputProcessor zur Verfügung.
- EncryptedKeyInputProcessor: Liest die `<EncryptedKey>` Struktur ein. Falls diese Struktur eine Liste mit Referenzen zu den verschlüsselten Teilen beinhaltet, wird ein DecryptInputProcessor instanziert und der Kette hinzugefügt.
- ReferenceListInputProcessor: Wenn der `<Security>` Header eine ReferenceList Struktur besitzt, wird diese mit dem ReferenceListInputProcessor eingelesen. Dieser Prozessor instanziert dann ebenfalls einen DecryptInputProcessor.
- DecryptInputProcessor: Sucht nach den referenzierten verschlüsselten Teilen im Dokument, instanziert pro verschlüsselten Teil einen DecryptedEventReaderInputProcessor und fügt diesen zur Kette hinzu. Ein Thread übernimmt für den aktuell zu entschlüsselnden Teil das Einlesen der chiffrierten Daten und reicht diese dem DecryptedEventReaderInputProcessor weiter.
- DecryptedEventReaderInputProcessor: Parst das entschlüsselte XML mit einem neuen XMLEventReader und reicht auf Anfrage die neu eingelesenen XMLEvents weiter durch die Kette.
- PolicyInputProcessor: Falls WS-SecurityPolicy konfiguriert wurde, überprüft dieser die Policy (3.2.5).
- InternalSecurityHeaderBufferProcessor: Speichert die XMLEvents bis zum Ende des `</Security>` Headers zwischen und übergibt sie danach dem InternalSecurityHeaderReplayProcessor. Dies ermöglicht auch das Entschlüsseln von Teilen des Dokuments die bereits vor dem Security Header vorgekommen sind.
- SecurityHeaderInputProcessor: Dies ist immer der letzte Prozessor in der Kette. Er instanziert anhand des `Security` Headers Prozessoren, wie zum Beispiel der EncryptedKeyInputProcessor oder ReferenceListInputProcessor, und fügt diese der Kette hinzu. Dieser Prozessor wird aus der Kette entfernt, sobald der ganze `<Security>` Header abgearbeitet wurde.

```
<env:Envelope xmlns:env="..." xmlns:xenc="...">
   <env:Header>
      <xenc:EncryptedData Id="EncDataId-1" Type="http://www.w3.org
         /2001/04/xmlenc#Element">
         ...
      </xenc:EncryptedData>
      <wsse:Security xmlns:wsse="..." env:mustUnderstand="1">
         <xenc:EncryptedKey Id="EncKeyId-1" >
            <xenc:EncryptionMethod Algorithm="http://www.w3.org
               /2001/04/xmlenc#rsa-1_5"/>
               ...
            <xenc:ReferenceList>
               <xenc:DataReference URI="#EncDataId-1"/>
            </xenc:ReferenceList>
         </xenc:EncryptedKey>
      </wsse:Security>
   </env:Header>
   <env:Body xmlns:wsu="...">
      ...
   </env:Body>
</env:Envelope>
```

Listing 3.1: Verschlüsselte SOAP-Header Elemente vor EncryptedKey

Problem verschlüsselte Security-Header Elemente

Oftmals ist gewünscht, dass Timestamps, Signaturen et cetera verschlüsselt übertragen werden um eine mögliche Angriffsfläche zu mindern. Wenn diese Teile aber verschlüsselt sind, kann der SecurityHeaderInputProcessor die zuständigen Prozessoren nicht instanzieren und als Folge daraus kann zum Beispiel auch keine Signatur- Verifizierung stattfinden. In dieser Arbeit wird davon ausgegangen, dass die `<EncryptedKey>` Struktur und die darin referenzierten Schlüssel vor den verschlüsselten Elementen deklariert wurden. Andere Elemente, wie SOAP-Headers, sind davon nicht betroffen. Darauf wird im nächsten Abschnitt eingegangen.

Problem verschlüsselte SOAP-Header Elemente

Es spricht nichts dagegen, ganze SOAP-Header Elemente zu verschlüsseln. Wenn nun wie im Beispiel 3.1 Rückwärts-Referenzen auftreten, ist es streaming-orientiert nicht mehr möglich diese Referenzen aufzufinden und zu entschlüsseln. Deshalb wird das SOAP-Dokument bis zum Ende des zuständigen Security Headers durch die Prozessor Kette “gejagt”, die notwendigen Prozessoren instanziert und alle bisher aufgetretenen XMLEvents zwischengespeichert. Danach werden die zwischengespeicherten XMLEvents erneut in die Kette eingespeist und mit dem restlichen Dokument normal weitergefahren.

Prozessor Lebensdauer

Um die Performance weiter zu steigern werden alle Prozessoren nur so lange wie nötig in der Kette gehalten. Es ist nicht notwendig, dass zum Beispiel ein BinarySecurityTokenIn-

putProcessor bis zum Ende des XML Dokuments in der Prozessor Kette gehalten wird. Dies verursacht nur unnötige Methoden-Aufrufe. Das flexible Design der Kette erlaubt dies problemlos.

Alle angesprochenen Aspekte (verschlüsselte SOAP- und Securiy Header, Prozessor Lebensdauer) beim Verarbeiten einer verschlüsselten Nachricht sind in Abbildung 3.4 ersichtlich.

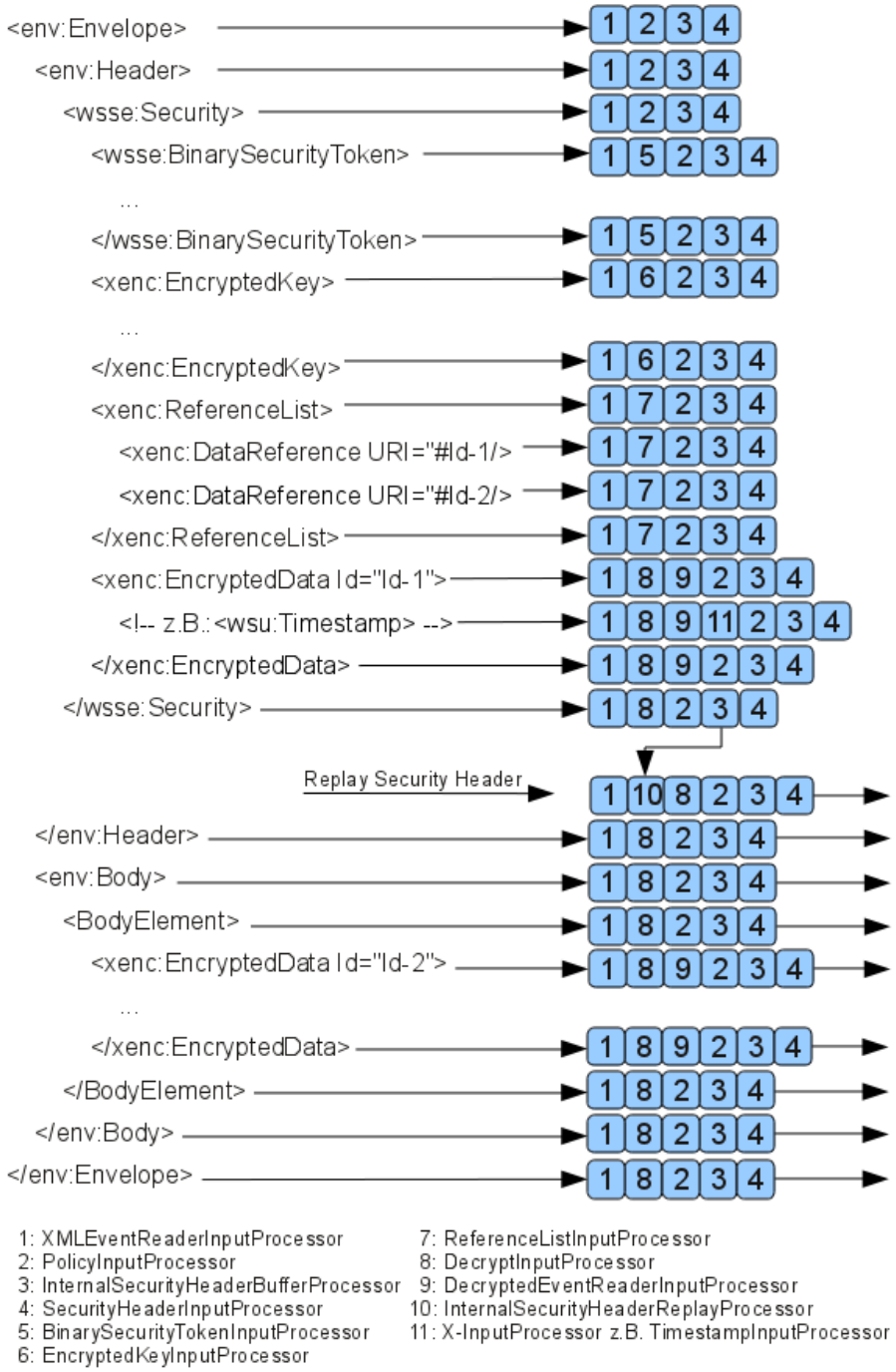


Abbildung 3.4: Input Prozessor Lebensdauer

Event basierte Entschlüsselung

Die verschlüsselte Daten sind Base64 kodiert im XML abgelegt. Dadurch werden die verschlüsselten Daten von einem XML-Parser als reinen Text angeschaut und als sogenannten Character-Event der aufrufenden Applikation zur Verfügung gestellt. Abbildung 3.2 zeigt ein solcher Ausschnitt mit dem umgebenden <CipherValue> Element.

```
<xenc:CipherValue>
 qsvxzDEz831z4sxRQFDt8pW2XNTRr9c39CLmMhm4fdQZDD1GPz9TTxhyBKHb0SL
 5q69sxJJRlZVxqKGKEvc09ZXpCwt71F1RmwrdjEU11YuuPBPIF0f+1y2dvxWHE=
</xenc:CipherValue>
```

Listing 3.2: Verschlüsselter Datenblock im XML

Die verschlüsselten Daten, also der ganze Text zwischen dem <CipherValue> Start- und End-Tag, kann beliebig lang sein, je nachdem wie viele Daten verschlüsselt wurden. Typischerweise wird nun ein streaming-orientierter XML-Parser nicht nur ein einzelner Character-Event für den Text generieren, sondern mehrere Character-Events nacheinander. Bei StAX-Parsern ist dieses Verhalten konfigurierbar (Stichwort Coalescing). Nehmen wir nun als Beispiel an, der Parser liefert uns ein Character-Event mit den ersten 32 Bytes an verschlüsselten und Base64 kodierten Daten. Der dazugehörige entschlüsselte Text laute (minus die angenommenen 16 Bytes für den IV) <Eleme. Wie man sieht, ist in diesem Beispiel nur ein Teil eines Start-Elements entschlüsselt worden. Für eine weitere Verarbeitung ist es aber notwendig, dass der entschlüsselte Text von einem Parser wieder eingelesen und als XMLEvents zur Verfügung gestellt werden kann. Um mit solchen fragmentierten Textteilen umgehen zu können, müssen diverse Eigenschaften berücksichtigt werden. Dazu zählt Base64 Kodierung mit Padding, Verschlüsselungsalgorithmus mit Padding, Initial-Vektor Länge, Länge der Text-Fragmente.

Lösungsansatz 1: Zwischenspeichern Die einfachste und nächstliegendste Lösung ist das Puffern aller Character Events vom </CipherValue> Start-Element bis zum abschliessenden
</CipherValue> Element. Im Puffer befindet sich dann der ganze Cipher-Text welcher in einem Rutsch Base64 dekodiert und entschlüsselt werden kann.

Noch einfacher geht es, wenn der StAX-Parser angewiesen wird "Coalescing" durchzuführen (als ein einzelner Character Event für den ganzen Text zwischen einem Start- und End-Element generieren). Dadurch entfällt das Puffern in der Applikation.

Der offensichtliche Nachteil dieser Lösung ist der Speicher-Verbrauch. Er wird auch zum limitierenden Faktor bezüglich der maximal zu verarbeitenden Grösse der Dokumente. Somit bewegen wir uns wieder in Richtung DOM Dokumente, wobei bei diesem Ansatz der Speicherverbrauch nicht ganz so dramatisch ist.

Lösungsansatz 2: Thread basiert Damit nicht der vorhandene Speicher zum limitierenden Faktor wird, wird hier ein Lösungsansatz präsentiert, der die einzelnen Character Events ebenso streaming-orientiert bearbeitet, wie bereits das ganze Dokument. Bei diesem Ansatz [Abbildung 3.5] wird ein neuer Thread verwendet, der über eine Pipe mit dem Main-Thread verbunden ist.

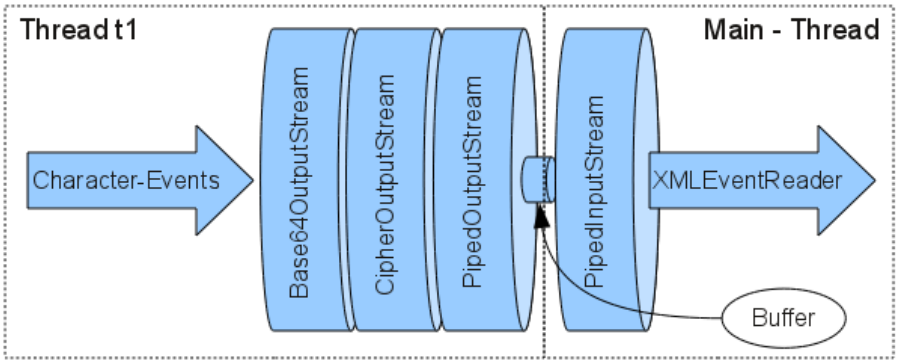


Abbildung 3.5: Thread basierte Entschlüsselung

Der neue Thread t1 übernimmt ab nun das Einlesen der Character-Events, überreicht diese dem OutputStream wo die Daten Base64 dekodiert, entschlüsselt und in den von den Threads gemeinsamen Speicherbereich abgelegt werden. Auf der anderen Seite der Pipe kann der Main-Thread die entschlüsselten Daten mit einem neu erzeugten StAX-Parser einlesen und wie gewohnt als XMLEvents weiterverarbeiten.

Dieses Design ermöglicht einen konstant kleinen Speicherverbrauch unabhängig von der Grösse des XML Dokuments. Nachteilig an dieser Lösung ist der zusätzlich benötigte Thread, welcher einen gewissen Overhead mit sich bringt, da dieser mit dem Main-Thread laufend synchronisiert werden muss. Wenn der Main-Thread Daten aus dem Puffer lesen will, muss er zuerst sicherstellen, dass Daten im Puffer vorhanden sind. Umgekehrt muss der Thread t1 sicherstellen, dass Platz für neue Daten im Puffer vorhanden ist. Aus diesem Grund spielt die Grösse des gemeinsamen Puffers eine wichtige Rolle, so dass beide Threads nicht dauernd aufeinander warten müssen. Zusätzlich birgt das Design die Gefahr einer DOS Attacke. Da bei jedem verschlüsselten Block ein neuer Thread benötigt wird, könnte ein möglicher Angreifer viele verschlüsselte Dokumente mit vielen Teilverschlüsselungen senden und so die Maschine mit Threads überlasten.

3.2.5 WS-SecurityPolicy Verifizierung

Einen Überblick über WS-SecurityPolicy wurde im Kapitel 2.3.6 gegeben. Im Folgenden werden wir nun besprechen wie WS-SecurityPolicy im streaming-basierten WS-Security-Framework integriert werden kann und was für Vorteile sich dadurch ergeben.

Der Ablauf in klassischen Web Service Frameworks mit DOM-basierter WS-Security Implementierung ist:

1. Parsen der Nachricht in eine DOM Struktur
2. Anwenden von WS-Security auf dem DOM
3. Speicherung der Resultate von WS-Security. Zu den Resultaten gehören Informationen wie verwendete Token, welche Elemente wurden verschlüsselt, ist ein Timestamp vorhanden et cetera.

4. WS-SecurityPolicy Validierung anhand der Resultate aus WS-Security

Dieses Vorgehen führt zu zwei Problemen:

1. Unvollständige Informationen. Oftmals sind unvollständige Informationen als Resultat vorhanden, so dass keine genaue Policy Validierung durchgeführt werden kann. Viele Informationen müssten nachträglich aus dem DOM herausgeholt werden um eine exakte Policy Verifizierung zu erreichen.

2. Späte Detektierung von Policy Verletzungen. Policy Verletzungen werden erst festgestellt nachdem die ganze Nachricht eingelesen und WS-Security angewendet wurde. Im Fall einer Verletzung wurden so bereits viele Computer-Ressourcen verschwendet.

In einer streaming-basierten Architektur ist es hingegen in vielen Fällen möglich, Policy Verletzungen direkt beim Auftreten von Policy relevanten Ereignissen zu Detektieren. Ein einfaches Beispiel für einen Fall, wo die Policy sofort zuschlagen kann, ist folgendes Szenario:

Gegeben sei folgende Policy:

```
<wsp:Policy wsu:Id="SamplePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:EncryptedParts>
        <sp:Body/>
      </sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Die Policy sagt aus, dass der ganze `<soap:Body>` verschlüsselt sein muss. Es gibt keine alternative Policy. Der Client sendet nun eine unverschlüsselte Nachricht. Sobald die Policy-Engine detektiert das der Inhalt des `<soap:Body>` unverschlüsselt ist wird eine Exception geworfen und die weitere Verarbeitung abgebrochen.

Ein Gegenbeispiel wo die sofortige Detektierung nicht mehr so einfach möglich ist:

```
<wsp:Policy wsu:Id="SamplePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding
          xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/
             securitypolicy">
        <wsp:Policy>
          <sp:AlgorithmSuite>
            <wsp:Policy>
              <sp:Basic256/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
        </wsp:Policy>
      </sp:AsymmetricBinding>
      <sp:EncryptedElements>
        <sp:XPath xmlns:ex="http://www.example.com">//ex:SomeElement</
           sp:XPath>
      </sp:EncryptedElements>
    </wsp:All>
```

```
      <wsp:All>
        <sp:AsymmetricBinding
            xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/
               securitypolicy">
          <wsp:Policy>
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp:TripleDes/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
          </wsp:Policy>
        </sp:AsymmetricBinding>
        <sp:EncryptedElements>
          <sp:XPath xmlns:ex="http://www.example.com">//ex:AnotherElement
             </sp:XPath>
        </sp:EncryptedElements>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
```

Diese Policy besteht aus zwei Policy-Alternativen von denen genau eine erfüllt werden muss. Die erste Policy-Alternative sagt aus, dass wenn ein Element mit dem Namen `{http://www.example.com}SomeElement` vorkommt, dies verschlüsselt sein muss. Zusätzlich wird gefordert, dass die Verschlüsselung mit AES-256 zu erfolgen hat. Bei der zweiten Policy-Alternative wird hingegen gefordert, dass wenn ein Element mit dem Namen `{http://www.example.com}AnotherElement` vorkommt, dies mit 3DES verschlüsselt sein muss. Sendet der Client nun eine Nachricht wo das Element `{http://www.example.com}SomeElement` mit 3DES verschlüsselt ist, passiert folgendes (In der Reihenfolge des Auftretens):

1. Die Policy-Engine bemerkt, dass das Element `{http://www.example.com}SomeElement` verschlüsselt ist und setzt dem entsprechend die Assertion in der ersten Policy-Alternative auf "true". Die zweite Policy-Alternative ist ebenfalls erfüllt, da weitere Elemente verschlüsselt sein dürfen, ohne dass dieses Element explizit in der Policy angegeben sein muss.

2. Die Verschlüsselung hat mit 3DES stattgefunden, somit wurde die Basic256-Assertion in der ersten Policy-Alternative nicht erfüllt. Hingegen die TripleDes-Assertion in der zweiten Policy-Alternative ist erfüllt.

3. Trifft nun die Policy-Engine auf ein unverschlüsseltes `{http://www.example.com}AnotherElement` ist die erste Policy-Alternative erfüllt da hier nicht gefordert wird dass dieses Element verschlüsselt sein muss. Die zweite Policy-Alternative wird hingegen verletzt.

Daraus resultiert folgender Assertion Status:

	1te Policy Alternative	2te Policy Alternative
AlgorithmSuite	∅	✓
EncryptedElements	✓	∅
Policy-Alternative erfüllt	∅	∅

Wenn mehrere Policy-Alternativen existieren ist oftmals kein sofortiger Abbruch bei einer Policy-Verletzung möglich. Bei sehr komplexen Policies mit mehrfach verschachtelten Policy-Alternativen und vielen Assertions zu Signatur, Verschlüsselung, Timestamps et cetera, kann es sogar dazu führen, dass eine Policy Verletzung erst ganz am Schluss festgestellt werden kann. Deshalb wird immer nach dem Verarbeiten des Dokuments eine vollständige Policy Validierung durchgeführt.

Abbildung 3.6 veranschaulicht wie WS-SecurityPolicy in das streaming-basierte WS-Security-Framework integriert ist.

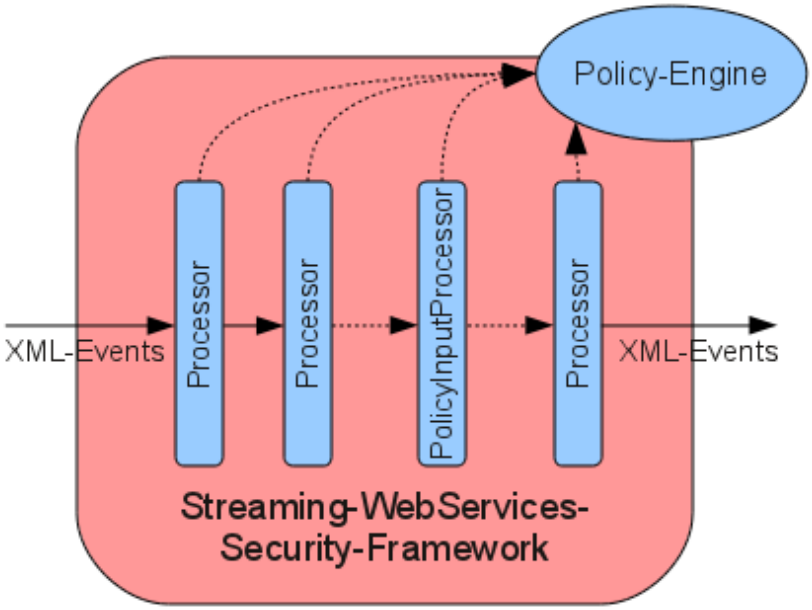


Abbildung 3.6: Policy-Engine Aufbau und Integration

Die Policy-Engine ist zuständig für

- das Parsen der Policy und vorbereiten der Assertions.
- das Auffinden der effektiven Policy anhand der eingehenden Nachricht.
- die Assertion Verifizierung in Echtzeit, das heisst beim Auftreten eines Policy relevanten Ereignisses die sofortige Überprüfung der Assertions.
- der Abschliessenden Policy Verifizierung.

Der PolicyInputProcessor hat die Aufgabe

- der Policy-Engine mitzuteilen, welche Service-Operation die aktuelle Nachricht aufruft um die effektive Policy ausfindig zu machen. Die Service-Operation ist gegeben durch das erste Element im `<soap:Body>`.
- der Policy-Engine mitzuteilen, welche XML-Elemente unverschlüsselt sind.
- in der Policy-Engine die finale Validierung anzustossen nachdem das ganze Dokument durchlaufen ist.

Alle anderen Prozessoren in der Kette informieren unmittelbar, beim Auftreten eines Policy relevanten Ereignisses , die Policy-Engine. So wird zum Beispiel vom DecryptInputProcessor ein Ereignis ausgelöst, das die Policy-Engine darüber informiert, welches Element mit welcher Granularität verschlüsselt wurde. Ein weiteres Ereignis vom DecryptInputProcessor ist das Token, das verwendet wurde, um die Nachricht zu verschlüsseln. Damit können Assertions wie AlgorithmSuite und RecipientToken verifiziert werden.

Reihenfolge der Policy relevanten Ereignisse Weil die SOAP-Nachrichten streaming-orientiert verarbeitet werden, treten auch die Ereignisse in Dokumentreihenfolge auf. Viele Policy relevante Ereignisse treten bereits beim Bearbeiten des SOAP-Headers auf wo sich die ganze WS-Security Struktur befindet. Zu diesem Zeitpunkt ist aber die effektive Policy noch nicht bekannt, da diese erst eindeutig durch die Service Operation bestimmt werden kann. Wie bereits erwähnt, ergibt sich die Service Operation durch das erste Kind Element im SOAP-Body. Solange also die Service Operation noch nicht bekannt ist, müssen die Ereignisse zwischengespeichert werden. Trifft die Policy-Engine später auf das Operations-Element werden alle zwischengespeicherten Ereignisse der Reihenfolge nach validiert. Es findet also keine WS-SecurityPolicy Validierung statt solange der SOAP-Header verarbeitet wird.

XPath Verarbeitung In WS-SecurityPolicy ist es an diversen Stellen möglich XPath Ausdrücke anzugeben. Ein Beispiel dafür ist:

```
<sp:EncryptedElements>
  <sp:XPath xmlns:ex="http://www.example.com">//ex:SomeElement</
     sp:XPath>
</sp:EncryptedElements>
```

Die XPath Verarbeitung im Zusammenhang mit Streaming ist aus Gründen der Komplexität nicht Teil dieser Arbeit. Überall wo XPath Ausdrücke verwendet werden können, wird in dieser Arbeit davon ausgegangen, dass nur das Element mit Namespace angegeben wurde. Anhand obigem Beispiel wird folgende Form erwartet:

```
<sp:EncryptedElements>
  <sp:XPath xmlns:ex="http://www.example.com">ex:SomeElement</sp:XPath>
</sp:EncryptedElements>
```

Somit "matchen" alle Elemente mit dem Namen `{http://www.example.com}SomeElement` egal wo sie im XML-Baum auftauchen. Für eine mögliche Erweiterung des Streaming-WebServices-Security-Framework um XPath Ausdrücke auswerten zu können sei hier SPEX [BCD+06] erwähnt.

Policy Abdeckung Wie wir bereits im Kapitel 2.3.6 gesehen haben, deckt WS-SecurityPolicy nicht nur die Nachrichten-Ebene ab, sondern auch die Transport-Ebene. Viele Assertions können auf beiden Ebenen erfüllt werden. Dazu gehören unter anderem die für diese Arbeit relevanten `<sp:EncryptedParts>` Assertions. So können am Beispiel von `<sp:EncryptedParts>` alle Assertions auch durch die SSL/TLS Verschlüsselung auf dem Transport-Layer erfüllt werden.

4 Implementierung

In diesem Kapitel wird genauer auf die Implementierung eingegangen. Es wird nicht jede Einzelheit beschrieben sondern nur spezifische Details herausgepickt, die besonders interessant sind oder Probleme verursachten.

4.1 Verwendete Frameworks

Entwickelt wurde das Streaming-WebServices-Security-Framework mit Java SE Development Kit 6. Die wichtigsten Bibliotheken die zusätzlich benötigt werden sind:

- Bouncy Castle [con10b]. Stellt die verschiedensten kryptographischen Algorithmen zur Verfügung. Durch Verwendung dieses Security Providers stehen unabhängig von der JVM alle benötigten Algorithmen zur Verfügung.
- Woodstox XML Processor [con10c]. Gilt als korrekter und sehr effizienter Parser.
- Apache Commons Codec [Apa10]. Aus dieser Bibliothek wird der Base64 Encoder / Decoder benötigt.
- Apache Neethi [con10a]. Neethi ist ein allgemeines WS-Policy Framework. Es kann Policies einlesen, normalisieren und schreiben.
- wsdl4j [wc10]. Ermöglicht das Einlesen von WSDL Dokumenten. Zusammen mit Apache Neethi können die im WSDL definierten Policies eingelesen und zu einer effektiven Policy zusammengeführt werden.

4.2 Framework Architektur

Der Aufbau des Frameworks ist bereits an diversen Orten [3.1, 3.2, 3.3a, 3.3b, 3.6] gezeigt worden.

4.2.1 Prozessor Phasen

Während dem Abarbeiten der SOAP-Nachricht werden fortlaufend, je nach Inhalt des Dokuments, neue Prozessoren instanziert und der Kette hinzugefügt. Jeder Prozessor hat die Möglichkeit weitere Hilfs-Prozessoren der Kette hinzuzufügen. Da die Reihenfolge sehr wichtig ist müssen die Prozessoren am richtigen Ort eingefügt werden. Die Prozessor-Kette ist in drei Phasen aufgeteilt: Phase 1 "**PREPROCESSING**", Phase 2 "**PROCESSING**" und Phase 3 "**POSTPROCESSING**". Zur Phase "PREPROCESSING" gehören Prozessoren die für das Einlesen des XML zuständig sind, zur Phase "PROCESSING" Prozessoren die die Nachricht bearbeiten und zur Phase "POSTPROCESSING" Prozessoren die abschliessende Arbeiten erledigen. Innerhalb dieser Phasen ist es wiederum möglich Abhängigkeiten unter den Prozessoren zu definieren. Durch diese Aufteilungen ist es möglich, exakt zu bestimmen, wo genau ein Prozessor in der Kette stehen muss.

Erreicht wird dies durch folgendes Interface, welches jeder Prozessor implementieren muss:

```
public interface InputProcessor {
    Set<String> getBeforeProcessors();
    Set<String> getAfterProcessors();
    Constants.Phase getPhase();
    ...
}
```

Jeder Prozessor bestimmt somit selber, wo genau er sich in der Kette einreihen soll. Mit `getBeforeProcessors()` und `getAfterProcessors()` kann der Prozessor bestimmen vor welchen, respektive nach welchen Prozessoren, er eingereiht werden soll. Gibt der Prozessor ein leeres Set zurück, wird er einfach am Ende der entsprechenden Phase eingehängt.

4.2.2 Schnittstelle zur Verarbeitung von SOAP-Nachrichten in Prozessoren

Die oben genannte Schnittstelle enthält auch Methoden zur Verarbeitung der Nachrichten. Die vollständige Input-Prozessor Schnittstelle sieht folgendermassen aus:

```
public interface InputProcessor {
    Set<String> getBeforeProcessors();
    Set<String> getAfterProcessors();
    Constants.Phase getPhase();

    XMLEvent processNextHeaderEvent(InputProcessorChain
        inputProcessorChain) throws XMLStreamException,
        WSSecurityException;
    XMLEvent processNextEvent(InputProcessorChain inputProcessorChain)
        throws XMLStreamException, WSSecurityException;
    void doFinal(InputProcessorChain inputProcessorChain) throws
        XMLStreamException, WSSecurityException;
}
```

Listing 4.1: InputProcessor Schnittstelle

Die Methode `processNextHeaderEvent(...)` wird vom SecurityHeaderInputProcessor solange aufgerufen bis der ganze Security-Header abgearbeitet wurde. Danach werden die XML-Events durch den "normalen" Pfad über die Methode `processNextEvent(...)` "gezogen". Die Methode `doFinal(...)` wird sobald das ganze Dokument verarbeitet ist einmal aufgerufen. Hier haben Prozessoren die Möglichkeit abschliessende Arbeiten zu verrichten. Jeder Prozessor fordert weitere XML-Events über den InputProcessorChain an:

```
XMLEvent xmlEvent = inputProcessorChain.processHeaderEvent();
und
XMLEvent xmlEvent = inputProcessorChain.processEvent();
```

Wie man sieht, werden die XML-Events nicht den Methoden übergeben sondern diese müssen explizit angefordert werden. Dadurch wird es auf einfache Weise möglich neue XML-Events zu generieren, zu überspringen oder einfach weiterzureichen.

Für die ausgehende Seite gibt es eine äquivalente Schnittstelle:

```
public interface OutputProcessor {
    Set<String> getBeforeProcessors();
    Set<String> getAfterProcessors();
    Constants.Phase getPhase();

    void processNextEvent(XMLEvent xmlEvent, OutputProcessorChain
        OutputProcessorChain) throws XMLStreamException,
        WSSecurityException;
    void doFinal(OutputProcessorChain OutputProcessorChain) throws
        XMLStreamException, WSSecurityException;
}
```

Bei den Output-Prozessoren fällt die Methode für die separate Header Verarbeitung weg. Im Gegensatz zum den Input-Prozessoren müssen hier die XMLEvents explizit durch die Kette weitergereicht werden:

```
outputProcessorChain.processEvent(xmlEvent);
```

Auch auf der ausgehenden Seite ist es dadurch sehr einfach möglich, neue XML-Events zu erzeugen, absorbieren oder ohne weitere Verarbeitung weiterzureichen.

4.3 Verschlüsselung

Die Verschlüsselung einer SOAP-Nachricht geschieht mit der Hilfe dreier Prozessoren:

EncryptOutputProcessor Pro Schlüssel gibt es einen EncryptOutputProcessor, dies Entspricht der `<xenc:EncryptedKey ...>` Struktur. Der Prozessor schaut bei jedem XMLEvent den er empfängt nach, ob dieser per Benutzer-Konfiguration verschlüsselt werden soll. Trifft dies zu, so wird eine neue Instanz des InternalEncryptionOutputProcessor der Kette hinzugefügt.

InternalEncryptionOutputProcessor Der InternalEncryptionOutputProcessor ist zuständig für die eigentliche Verschlüsselung. Bei der Instanziierung des Prozessors wird gleich die symmetrische Cipher und der Verschlüsselungs-OutputStream aufgebaut (hier in abgekürzter Form dargestellt):

```
Cipher symmetricCipher = Cipher.getInstance(jceAlgorithm);
symmetricCipher.init(Cipher.ENCRYPT_MODE, encryptionPartDef.
    getSymmetricKey());
byte[] iv = symmetricCipher.getIV();

CharacterEventGeneratorOutputStream characterEventGeneratorOutputStream
    =
    new CharacterEventGeneratorOutputStream(xmlEventNSAllocator);
Base64OutputStream base64EncoderStream =
    new Base64OutputStream(
        new BufferedOutputStream(characterEventGeneratorOutputStream),
        true, 76, new byte[]{'\n'}
    );

base64EncoderStream.write(iv);

CipherOutputStream cipherOutputStream = ;
OutputStreamWriter streamWriter = new OutputStreamWriter(
    new CipherOutputStream(base64EncoderStream, symmetricCipher)
);
```

Zu beachten ist hier, dass der Initial-Vektor (IV) direkt Base64 kodiert und nicht am CipherOutputStream übergeben wird. Deshalb findet hier auch ein Bruch zwischen dem CipherOutputStream und dem Base64OutputStream statt. Würde der IV auch verschlüsselt, könnte der Empfänger die Nachricht logischerweise nicht mehr entschlüsseln da der IV zum Initialisieren der Entschlüsselungs-Cipher wieder benötigt wird. Danach wird jeder über die oben beschriebene Methode

```
void processNextEvent(XMLEvent xmlEvent, OutputProcessorChain
    OutputProcessorChain) ...
```

eintreffende XMLEvent wird in den OutputStreamWriter geschrieben, am Ende der OutputStream-Kette als Character-XMLEvent abgefangen und als neuer XML-Event in die Prozessor-Kette eingeschleust.

EncryptEndingOutputProcessor Im EncryptEndingOutputProcessor werden alle über die Methode `processNextEvent(...)` eintreffenden XMLEvents zwischengespeichert. Sobald das ganze Dokument verarbeitet wurde, wird die Methode `doFinal(...)` aufgerufen in welcher der `<EncryptedKey>` Security-Header geschrieben wird:

```
createStartElement(outputProcessorChain, Constants.
    TAG_xenc_EncryptedKey, attributes);
createStartElement(outputProcessorChain, Constants.
    TAG_xenc_EncryptionMethod, attributes);
createEndElement(outputProcessorChain, Constants.
    TAG_xenc_EncryptionMethod);
```

Als nächstes wird die `SecurityTokenReference` geschrieben. Je nach Konfiguration des Frameworks sind verschiedene Arten möglich. Als Beispiel wird hier die sogenannte

IssuerSerial Referenzierung dargestellt:

```
createStartElement(outputProcessorChain, Constants.TAG_dsig_KeyInfo,
    null);
createStartElement(outputProcessorChain, Constants.
    TAG_wsse_SecurityTokenReference, null);
createStartElement(outputProcessorChain, Constants.TAG_dsig_X509Data,
    null);
createStartElement(outputProcessorChain, Constants.
    TAG_dsig_X509IssuerSerial, null);
createStartElement(outputProcessorChain, Constants.
    TAG_dsig_X509IssuerName, null);
createCharacters(outputProcessorChain, RFC2253Parser.normalize(
    x509Certificate.getIssuerDN().getName()));
createEndElement(outputProcessorChain, Constants.
    TAG_dsig_X509IssuerName);
createStartElement(outputProcessorChain, Constants.
    TAG_dsig_X509SerialNumber, null);
createCharacters(outputProcessorChain, x509Certificate.getSerialNumber
    ().toString());
createEndElement(outputProcessorChain, Constants.
    TAG_dsig_X509SerialNumber);
createEndElement(outputProcessorChain, Constants.
    TAG_dsig_X509IssuerSerial);
createEndElement(outputProcessorChain, Constants.TAG_dsig_X509Data);
createEndElement(outputProcessorChain, Constants.
    TAG_wsse_SecurityTokenReference);
createEndElement(outputProcessorChain, Constants.TAG_dsig_KeyInfo);
```

Danach wird der symmetrische Session-Key, der verwendet wurde um die Daten zu verschlüsseln, mit dem Public-Key des Empfängers verschlüsselt und geschrieben:

```
createStartElement(outputProcessorChain, Constants.TAG_xenc_CipherData,
     null);
createStartElement(outputProcessorChain, Constants.TAG_xenc_CipherValue
    , null);

String jceid = JCEAlgorithmMapper.translateURItoJCEID(
    getSecurityProperties().getEncryptionKeyTransportAlgorithm());
Cipher cipher = Cipher.getInstance(jceid);
cipher.init(Cipher.ENCRYPT_MODE, x509Certificate);
byte[] sessionKey = symmetricKey.getEncoded();
int blockSize = cipher.getBlockSize();
byte[] encryptedSessionKey = cipher.doFinal(sessionKey);

createCharacters(outputProcessorChain, new String(Base64.encode(
    encryptedSessionKey)));
createEndElement(outputProcessorChain, Constants.TAG_xenc_CipherValue);
createEndElement(outputProcessorChain, Constants.TAG_xenc_CipherData);
```

Zuletzt werden die Referenzen zu den verschlüsselten Daten gelegt:

```
createStartElement(outputProcessorChain, Constants.
    TAG_xenc_ReferenceList, null);

Iterator<EncryptionPartDef> encryptionPartDefIterator =
    encryptionPartDefList.iterator();
while (encryptionPartDefIterator.hasNext()) {
```

```
        EncryptionPartDef encryptionPartDef = encryptionPartDefIterator.
            next();

        attributes = new HashMap<QName, String>();
        attributes.put(Constants.ATT_NULL_URI, "#" + encryptionPartDef.
            getEncRefId());
        createStartElement(outputProcessorChain, Constants.
            TAG_xenc_DataReference, attributes);
        createEndElement(outputProcessorChain, Constants.
            TAG_xenc_DataReference);
    }

    createEndElement(outputProcessorChain, Constants.TAG_xenc_ReferenceList
        );
```

Somit ist der EncryptedKey Header vollständig aufgebaut und die restlichen XMLEvents können nun weitergereicht werden.

4.4 Entschlüsselung

Auf der Empfängerseite sind folgende Prozessoren an der Entschlüsselung der SOAP-Nachrichten beteiligt:

DecryptInputProcessor Der DecryptInputProcessor wird entweder von einem EncryptedKeyInputProcessor oder einem ReferenceListInputProcessor instanziert. Bei der Instanziierung wird die Liste mit den Referenzen zu den verschlüsselten Daten übergeben. Sobald der DecryptInputProcessor auf ein Start-Element mit dem Namen <EncryptedData> stösst, wird nachgeschaut ob dieses Element, ein Attribut Id hat, dessen Wert in der Liste vorkommt. Wenn diese Bedingungen zutreffen wird eine neue Instanz vom DecryptedEventReaderInputProcessor in die Prozessor-Kette eingehängt und der DecryptionThread gestartet. Ab diesem Zeitpunkt wird die Prozessor-Kette sozusagen in zwei Teile aufgeteilt. Die verschlüsselten Daten gelangen nur noch bis zum **DecryptionThread** und die entschlüsselten Daten werden vom **DecryptedEventReaderInputProcessor** in die Kette eingespiesen.

DecryptionThread Der DecryptionThread ist kein InputProcessor. Er übernimmt als selbständiger Thread das Einlesen der verschlüsselten Daten, entschlüsselt sie und stellt diese dem DecryptedEventReaderInputProcessor über einen Piped- Input / Ouput- Stream zur Verfügung [Abbildung 3.5].

Ungültiges XML im Entschlüsselungs-Kontext Im Kapitel 2.2.1 wurde beschrieben, auf welche Arten XML verschlüsselt werden kann. Das entschlüsselte Teil-XML muss wieder von einem Parser eingelesen werden können, um die XMLEvents durch die Prozessor-Kette weiterzureichen. Dies ist aber nicht in jedem Fall gegeben wie folgendes Beispiel zeigt. Ausgehend vom XML

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <Kreditkarte>
    <Nummer>1234</Nummer>
    <Aussteller>Institut $$$</Aussteller>
    <Ablaufdatum>05.05.2012</Ablaufdatum>
  </Kreditkarte>
</Zahlungsinformationen>
```

wird eine "Content"-Encryption auf dem Element <Nummer> vorgenommen:

```
<?xml version='1.0'?>
<Zahlungsinformationen>
  <Name>John Smith</Name>
  <Kreditkarte>
    <Nummer>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>Z63Y21XW</CipherValue>
        </CipherData>
      </EncryptedData>
    </Nummer>
    <Aussteller>Institut $$$</Aussteller>
    <Ablaufdatum>05.05.2012</Ablaufdatum>
  </Kreditkarte>
</Zahlungsinformationen>
```

Trifft nun der **DecryptionThread** auf eine solche Konstellation wird folgendes Fragment entschlüsselt:

```
1234
```

Dies kann selbstverständlich nicht von einem XML-Parser eingelesen werden, da es kein gültiges XML ist.

Ein weiterer Fall ergibt sich, wenn im obigen Beispiel der ganze Inhalt von <Kreditkarte> verschlüsselt wurde. Nach der Entschlüsselung hätte man folgendes XML-Fragment:

```
<Nummer>1234</Nummer>
<Aussteller>Institut $$$</Aussteller>
<Ablaufdatum>05.05.2012</Ablaufdatum>
```

Dieses XML kann auch wieder nicht eingelesen werden, da mehrere Wurzel-Elemente vorhanden sind.

Das ist aber noch nicht alles. Ein weiteres Problem wartet auf uns: Angenommen folgende SOAP-Nachricht wird auf dem Element <Kreditkarte> "Element"-Encrypted:

```
<?xml version='1.0'?>
<Zahlungsinformationen xmlns:"http://www.example.org" xmlns:a="http://
   www.a.com" xmlns:b="http://www.b.com">
  <Name>John Smith</Name>
  <Kreditkarte>
    <a:Nummer b:check="10">1234</a:Nummer>
    <a:Aussteller>Institut $$$</a:Aussteller>
```

```
    <a:Ablaufdatum>05.05.2012</a:Ablaufdatum>
  </Kreditkarte>
</Zahlungsinformationen>
```

Dann sieht das entschlüsselte Teil-XML folgendermassen aus:

```
<Kreditkarte>
  <a:Nummer b:check="10">1234</a:Nummer>
  <a:Aussteller>Institut $$$</a:Aussteller>
  <a:Ablaufdatum>05.05.2012</a:Ablaufdatum>
</Kreditkarte>
```

Dieses XML-Fragment ist auch wieder ungültig da die Namespaces für die Prefixe a, b und der "default-Prefix" nicht mehr aufgelöst werden können.

Um ungültige XML-Fragmente zu umgehen, wird ein <dummy> Element eingeführt, dass das entschlüsselte XML-Fragment "wrappt". Auf diesem Dummy-Element werden alle Namespaces vom aktuellen Namespace-Scope deklariert, so dass diese vom XML-Parser aufgelöst werden können:

```
//temporary writer for direct writing plaintext data
BufferedWriter tempBufferedWriter = new BufferedWriter(
  new OutputStreamWriter(
    pipedOutputStream,
    inputProcessorChain.getDocumentContext().getEncoding()
  )
);

final QName wrapperElementName = new QName("http://dummy", "dummy",
    uuid);

tempBufferedWriter.write('<');
tempBufferedWriter.write(wrapperElementName.getPrefix());
tempBufferedWriter.write(':');
tempBufferedWriter.write(wrapperElementName.getLocalPart());
tempBufferedWriter.write(' ');
tempBufferedWriter.write("xmlns:");
tempBufferedWriter.write(wrapperElementName.getPrefix());
tempBufferedWriter.write("=\"");
tempBufferedWriter.write(wrapperElementName.getNamespaceURI());
tempBufferedWriter.write('\"');

Iterator<ComparableNamespace> comparableNamespaceIterator =
    comparableNamespacesToApply.iterator();
while (comparableNamespaceIterator.hasNext()) {
  ComparableNamespace comparableNamespace = comparableNamespaceIterator
      .next();
  tempBufferedWriter.write(' ');
  tempBufferedWriter.write(comparableNamespace.toString());
}

tempBufferedWriter.write('>');
tempBufferedWriter.flush();
```

Wie man aus dem Programm-Ausschnitt sehen kann, wird das <dummy> Element direkt am PipedOutputStream übergeben. Auf der anderen Seite der Pipe, im DecryptedEventReaderInputProcessor, wird dann das Empfangene <dummy> Element zwar vom Parser eingelesen,

der daraus generierte XMLEvent aber wird verworfen. Sobald das <dummy> Element in die Pipe geschrieben ist, kann begonnen werden die Daten zu entschlüsseln und in die Pipe zu schreiben. Dafür wird auch wieder eine Verkettung von OutputStreams benötigt. Eine vereinfachte Darstellung wie die vielen OutputStreams verknüpft sind zeigt folgendes Listing:

```
decryptOutputStream = new Base64OutputStream(
                          new ReplaceableOuputStream(
                              new IVSplittingOutputStream(
                                  new CipherOutputStream(
                                      new FilterOutputStream(
                                          pipedOutputStream)
                                  )
                              )
                          )
                      )
```

Die detailierte Version, wie sie implementiert ist, werden wir nun besprechen:

```
1  IVSplittingOutputStream ivSplittingOutputStream = new
       IVSplittingOutputStream(
2    new CipherOutputStream(new FilterOutputStream(pipedOutputStream) {
3      //die ueberschriebenen write(...) Methoden sind aus lebarkeit
           weggelassen worden
4
5      @Override
6      public void close() throws IOException {
7        out.flush();
8      }
9    }, symmetricCipher),
10 symmetricCipher, secretKey);
11
12 ReplaceableOuputStream replaceableOuputStream = new
       ReplaceableOuputStream(ivSplittingOutputStream);
13 OutputStream decryptOutputStream = new Base64OutputStream(
       replaceableOuputStream, false);
14 ivSplittingOutputStream.setParentOutputStream(replaceableOuputStream);
15
16 boolean finished = false;
17 while (!finished) {
18   XMLEvent xmlEvent = processNextEvent();
19
20   switch (xmlEvent.getEventType()) {
21     case XMLStreamConstants.END_ELEMENT:
22       //this must be the CipherValue EndElement.
23       finished = true;
24       break;
25     case XMLStreamConstants.CHARACTERS:
26       decryptOutputStream.write(xmlEvent.asCharacters().getData().
             getBytes(inputProcessorChain.getDocumentContext().getEncoding
             ()));
27       break;
28     default:
29       throw new WSSecurityException("Unexpected event: " + Utils.
             getXMLEventAsString(xmlEvent));
30   }
31 }
32
33 //close to get Cipher.doFinal() called
```

```
34 decryptOutputStream.close();
```

Listing 4.2: Streaming-basierte Entschlüsselung

Der Base64OutputStream und der CipherOutputStream auf den Zeilen 2 und 12 übernehmen das Dekodieren und das Entschlüsseln der Daten. Interessanter wird es auf der Zeile 1 beim IVSplittingOutputStream. Dieser initialisiert die Cipher mit dem Initalvektor (IV) sobald genug Bytes angekommen sind. Wenn der IV vollständig ist und die Cipher initialisiert wurde, wird mit Hilfe des ReplaceableOuputStream (Zeile 11 und 13) der IVSplittingOutputStream aus der OutputStream Kette entfernt, so dass dieser nicht die Performance negativ beeinflusst, da es sich beim IV nur um die ersten paar Bytes im Datenstrom handelt. Nachdem der OutputStream aufgesetzt ist, werden im while-loop (Zeilen 16 - 29) solange die nächsten XMLEvents von der Prozessor-Kette angefordert und in den decryptOutputStream geschrieben, bis das `<CipherValue>` End-Element auftaucht. Auf der Zeile wird 34 wird der decryptOutputStream geschlossen, so dass gegebenenfalls gepufferte Daten Base64 kodiert, und `Cipher.doFinal()` aufgerufen wird um die Verschlüsselung abzuschliessen. Leider zieht sich der Aufruf von `close()` durch alle verschachtelten OutputStream bis zum PipedOutputStream durch. Dieser würde auch geschlossen und der XMLEventReader auf der anderen Seite der Pipe würde einer EndOfStream-Exception werfen, da Ihm noch das Abschliessende <dummy> Element fehlt. Da das `close()` aber zwingend notwendig ist um die Verschlüsselung abzuschliessen, wird nun das `close()` im FilterOutputStream abgefangen und durch ein flush() ersetzt (Zeile 7). Als abschliessende Arbeit wird nun nur noch das End </dummy> Element in den PipedOutputStream geschrieben und nun das Finale `close()` aufgerufen:

```
tempBufferedWriter.write("</");
tempBufferedWriter.write(wrapperElementName.getPrefix());
tempBufferedWriter.write(':');
tempBufferedWriter.write(wrapperElementName.getLocalPart());
tempBufferedWriter.write('>');
//real close of the stream
tempBufferedWriter.close();
```

DecryptedEventReaderInputProcessor Der entschlüsselte Datenstrom muss in Form von XMLEvents wieder in die Prozessor-Kette eingeschleust werden. Dafür ist der DecryptedEventReaderInputProcessor zuständig. Anstelle der Aufrufe von

```
XMLEvent xmlEvent = inputProcessorChain.processEvent();
```

für das Anfordern der nächsten XMLEvents über die Prozessor-Kette werden die XMLEvents über einen neuen XMLEventReader eingelesen und weitergereicht:

```
XMLEvent xmlEvent = xmlEventReader.nextEvent();
```

Der XMLEventReader zum Einlesen des entschlüsselten Datenstrom wird über den Konstruktor mitgegeben und erhält als Input-Quelle den PipedInputStream vom DecryptedEventReaderInputProcessor.

Exceptions Exception die im Thread auftreten, da zum Beispiel nicht entschlüsselt werden konnte, gehen verloren und der Thread “stirbt” danach. Dies führt dazu, dass wir auf der lesenden Seite der Pipe ein ein IOEXception “Write end dead” sehen und sonst nichts.

Irgendwie muss der Applikation mitgeteilt werden können, dass etwas schief gelaufen ist. Für das gibt es in Java die Möglichkeit einen sogenannten "UncaughtExceptionHandler" anzugeben. Alle Exception die nicht abgefangen wurden, werden nun an diesen Händler weitergereicht.

Wir verwenden dies um die Exception im Entschlüsselungs-Thread sauber zum Main-Thread weiterzureichen und dementsprechend darauf reagieren zu können:

```
1  class DecryptedEventReaderInputProcessor implements Thread.
       UncaughtExceptionHandler {
2    private Throwable thrownException;
3    ...
4    private XMLEvent processEvent(InputProcessorChain inputProcessorChain
         , boolean headerEvent) throws XMLStreamException,
         WSSecurityException {
5      testAndThrowUncaughtException();
6      XMLEvent xmlEvent = xmlEventReader.nextEvent();
7      ...
8    }
9    public void uncaughtException(Thread t, Throwable e) {
10     this.thrownException = e;
11   }
12   public void testAndThrowUncaughtException() throws XMLStreamException
         {
13     if (this.thrownException != null) {
14       if (this.thrownException instanceof UncheckedWSSecurityException)
             {
15         UncheckedWSSecurityException uxse = (
               UncheckedWSSecurityException) this.thrownException;
16         throw new XMLStreamException(uxse.getCause());
17       } else {
18         throw new XMLStreamException(this.thrownException.getCause());
19       }
20     }
21   }
22 }
```

Listing 4.3: Abfangen von Exceptions in Threads

Um unbehandelte Exceptions empfangen zu können, muss das Interface `Thread.UncaughtExceptionHandler` implementiert werden (Zeile 1). Sobald eine Exception auftritt, wird die Methode `uncaughtException(...)` aufgerufen (Zeilen 9 - 11). Die Exception wird in einer globalen Variable gespeichert. Sobald der nächste XMLEvent über die Methode `processEvent(...)` (Zeile 4) angefordert wird, wird zuerst geprüft (Zeile 5 und Zeilen 12 - 21) ob eine Exception vorhanden ist. Falls dem so ist, wird diese hier erneut geworfen. Diese wird dann durch die ganze Prozessor-Kette hindurch propagiert.

Damit das ganze funktioniert, muss dem Thread nur noch gesagt werden, wer für unbehandelte Exceptions zuständig ist. In unserem Fall ist der Händler der DecryptedEventReaderInputProcessor:

```
receiverThread.setUncaughtExceptionHandler(
    decryptedEventReaderInputProcessor);
receiverThread.start();
```

4.5 WS-SecurityPolicy Validierung

Das Policy-Framework ist ein eigenständiges Framework. Es sind nur sehr weniger Berührungspunkte zwischen dem Streaming-WebServices-Security-Framework und dem Policy-Framework vorhanden. Das Einbinden des Policy-Framework ist sehr einfach gehalten:

```
PolicyEnforcerFactory policyEnforcerFactory = PolicyEnforcerFactory.
    newInstance("msg.wsdl"));
PolicyEnforcer policyEnforcer = policyEnforcerFactory.newPolicyEnforcer
    (null);
SecurityProperties securityProperties = new SecurityProperties();
securityProperties.addInputProcessor(new PolicyInputProcessor(
    policyEnforcer, null));
InboundWSSec wsSec = WSSec.getInboundWSSec(securityProperties);
XMLStreamReader outXmlStreamReader = wsSec.processInMessage(
    xmlStreamReader, policyEnforcer);
```

Listing 4.4: Einbinden des Policy-Frameworks

Die **PolicyEnforcerFactory** liest eine WSDL Datei ein und sucht zu allen Web Service Operationen die definierten Policies heraus. Die gefundenen Policies werden in einer Map den Operationen zugeordnet und im Speicher gehalten um wiederholtes Einlesen der Policies zu vermeiden. Danach kann über die PolicyEnforcerFactory eine neue Instanz des **PolicyEnforcer** geholt werden.

Der **PolicyEnforcer** hat das SecurityEventListener Interface implementiert, welches ihm erlaubt SecurityEvents von den Prozessoren zu erhalten. Daher ist es möglich, den PolicyEnforcer der `processInMessage(...)` Methode zu übergeben. Die SecurityEvents werden später noch genauer besprochen.

Über die **SecurityProperties** (Die Konfiguration des Streaming-WebServices-Security-Frameworks) kann der **PolicyInputProcessor** zusätzlich in die Prozessor-Kette eingefügt werden. Der PolicyInputProcessor überprüft ob unverschlüsselte Elemente in der SOAP-Nachricht vorkommen die eigentlich verschlüsselt sein müssten.

SecurityEvents SecurityEvent ist ein Konzept das für diese Arbeit entwickelt wurde. Jeder Prozessor hat jederzeit die Gelegenheit SecurityEvents abzufeuern. Ein SecurityEvent ist definiert durch folgende abstrakte Klasse:

```
public abstract class SecurityEvent {
    public enum Event {
        Operation,
        Timestamp,
        SignedPart,
        SignedElement,
        InitiatorEncryptionToken,
        RecipientEncryptionToken,
        AlgorithmSuite,
        EncryptedPart,
        EncryptedElement,
        ContentEncrypted,
    }
```

```
    private Event securityEventType;

    protected SecurityEvent(Event securityEventType) {
        this.securityEventType = securityEventType;
    }

    public Event getSecurityEventType() {
        return securityEventType;
    }

    public void setSecurityEventType(Event securityEventType) {
        this.securityEventType = securityEventType;
    }
}
```

Um SecurityEvents empfangen zu können, muss das Interface SecurityEventListener implementiert werden:

```
public interface SecurityEventListener {
    public void registerSecurityEvent(SecurityEvent securityEvent)
        throws WSSecurityException;
}
```

und über die API der Klasse InboundWSSec registriert werden:

```
public XMLStreamReader processInMessage(XMLStreamReader xmlStreamReader
    , SecurityEventListener securityEventListener) throws
    XMLStreamException, WSSecurityException {
```

Die Schnittstelle wurde so gewählt, das einerseits ein PolicyFramework wie das unsere die Ereignisse möglichst losgelöst vom Streaming-WebServices-Security-Framework erhalten kann und andererseits ist möglicherweise auch der Web Service daran interessiert was genau gelaufen ist während der Verarbeitung. Die SecurityEvents werden sozusagen in Echtzeit von den Prozessoren ausgelöst sobald ein relevantes Ereignis auftritt. Dies ermöglicht das erwünschte “fail-fast” Verhalten in vielen Fällen.

Ein Beispiel von einem konkreten SecurityEvent ist folgende Klasse:

```
public class EncryptedPartSecurityEvent extends SecurityEvent {

    //todo xpath:
    private QName element;
    private boolean notEncrypted; //if true this element is not
        encrypted.

    public EncryptedPartSecurityEvent(Event securityEventType, boolean
        notEncrypted) {
        super(securityEventType);
        this.notEncrypted = notEncrypted;
    }

    public QName getElement() {
        return element;
    }

    public void setElement(QName element) {
        this.element = element;
```

```
    }

    public boolean isNotEncrypted() {
        return notEncrypted;
    }

    public void setNotEncrypted(boolean notEncrypted) {
        this.notEncrypted = notEncrypted;
    }
}
```

Dieser EncryptedPartSecurityEvent wird vom DecryptInputProcessor ausgelöst sobald er auf ein Encrypted-Part stösst, wie den `<env:Body>` oder ein SOAP-Header Element.

Ist nun der PolicyEnforcer als SecurityEventListener registriert, erhält dieser den Security-Event.

PolicyEnforcer Der PolicyEnforcer ist der Dreh und Angelpunkt der ganzen WS-SecurityPolicy Validierung. Er übernimmt die Validierung der eintreffenden SecurityEvents und vergleicht diese mit den konfigurierten Assertions. Sobald die effektive Policy anhand von der Web Service Operation ausgemacht wurde, wird eine Map mit den Events und den dazugehörenden AssertionState's gebildet. Das Bilden der Map geschieht so:

```
1  private void buildAssertionStateMap(PolicyComponent policyComponent,
       Map<SecurityEvent.Event, Collection<AssertionState>>
       assertionStateMap) throws WSSPolicyException {
2      if (policyComponent instanceof PolicyOperator) {
3          PolicyOperator policyOperator = (PolicyOperator)
               policyComponent;
4          List<PolicyComponent> policyComponents = policyOperator.
               getPolicyComponents();
5          for (int i = 0; i < policyComponents.size(); i++) {
6              PolicyComponent curPolicyComponent = policyComponents.get(i
                   );
7              buildAssertionStateMap(curPolicyComponent,
                   assertionStateMap);
8          }
9      } else if (policyComponent instanceof AbstractSecurityAssertion) {
10         AbstractSecurityAssertion abstractSecurityAssertion = (
               AbstractSecurityAssertion) policyComponent;
11         abstractSecurityAssertion.getAssertions(assertionStateMap);
12     } else {
13         throw new WSSPolicyException("Unknown PolicyComponent: " +
               policyComponent + " " + policyComponent.getType());
14     }
15 }
```

Es wird rekursiv über alle `PolicyComponent` (Policy, ExactlyOne, All, Assertion) iteriert (Zeile 7) und aus allen Assertions (Zeile 11) ein entsprechendes AssertionState Objekt erzeugt und der Map hinzugefügt. Folgende Beispiele beschreiben den Validierungsvorgang anhand der AlgorithmSuite Assertion.

Abbildung 4.1 zeigt eine Übersicht der Klassen:

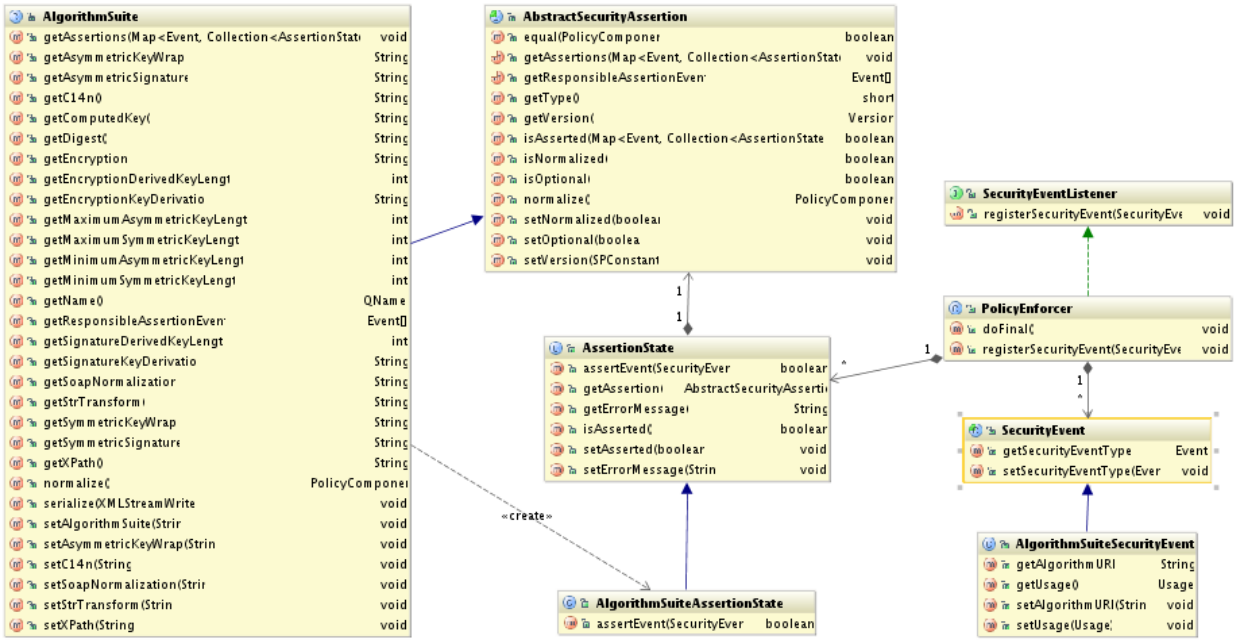


Abbildung 4.1: AlgorithmSuiteAssertion Paketdiagramm

```
1 public class AlgorithmSuite extends
      AbstractConfigurableSecurityAssertion {
2     ...
3     public void getAssertions(Map<SecurityEvent.Event, Collection<
          AssertionState>> assertionStateMap) {
4         Collection<AssertionState> assertionStates = assertionStateMap.
              get(SecurityEvent.Event.AlgorithmSuite);
5         AlgorithmSuiteAssertionState algorithmSuiteAssertionState = new
               AlgorithmSuiteAssertionState(this, true);
6         assertionStates.add(algorithmSuiteAssertionState);
7     }
8 }
```

Ein AssertionState Objekt überprüft einzelne Assertions anhand der zugehörenden SecurityEvents und hält das Resultat der Überprüfung lokal fest:

```
1 public class AlgorithmSuiteAssertionState extends AssertionState {
2
3     public AlgorithmSuiteAssertionState(AbstractSecurityAssertion
          assertion, boolean asserted) {
4         super(assertion, asserted);
5     }
6
7     @Override
8     public boolean assertEvent(SecurityEvent securityEvent) {
9         AlgorithmSuiteSecurityEvent algorithmSuiteSecurityEvent = (
              AlgorithmSuiteSecurityEvent) securityEvent;
10        AlgorithmSuite algorithmSuite = (AlgorithmSuite) getAssertion()
              ;
11
12        switch (algorithmSuiteSecurityEvent.getUsage()) {
13            case Enc:
```

```
14                  if (!algorithmSuite.getEncryption().equals(
                        algorithmSuiteSecurityEvent.getAlgorithmURI())) {
15                      setAsserted(false);
16                      setErrorMessage("Encryption algorithm " +
                            algorithmSuiteSecurityEvent.getAlgorithmURI() +
                             " does not meet policy");
17                  }
18                  break;
19              ...
20          }
21          return isAsserted();
22      }
23  }
```

Wird nun zum Beispiel vom DecryptInputProcessor ein AlgorithmSuiteSecurityEvent ausgelöst, der den Verschlüsselungsalgorithmus enthält der verwendet wurde, werden vom PolicyEnforcer alle AlgorithmSuiteAssertionState aus der Map abgefragt, ob sie verletzt werden. Wenn alle "false" zurückgeben kann die weitere Verarbeitung abgebrochen werden. Gibt mindestens ein AlgorithmSuiteAssertionState "true" zurück, wurde mindestens eine Policy-Alternative erfüllt und die Verarbeitung kann noch nicht abgebrochen werden.

Ist bis zum Ende der SOAP-Nachricht keine Policy-Verletzung aufgetreten, wird vom PolicyInputProcessor die finale Policy-Validierung angestossen. Hier wird ebenfalls wieder rekursiv über alle `PolicyComponent` iteriert und die AssertionStates den konkreten Assertions zur Verifizierung übergeben (Zeile 20):

```
1  private boolean verifyPolicy(PolicyComponent policyComponent) throws
       WSSPolicyException, PolicyViolationException {
2      if (policyComponent instanceof PolicyOperator) {
3          PolicyOperator policyOperator = (PolicyOperator)
               policyComponent;
4          boolean isExactlyOne = policyOperator instanceof ExactlyOne;
5          List<PolicyComponent> policyComponents = policyOperator.
               getPolicyComponents();
6
7          boolean isAsserted = false;
8          for (int i = 0; i < policyComponents.size(); i++) {
9              PolicyComponent curPolicyComponent = policyComponents.get(i
                   );
10             isAsserted = verifyPolicy(curPolicyComponent);
11             if (isExactlyOne && isAsserted) {
12                 return true; //a satisfied alternative is found
13             } else if (!isExactlyOne && !isAsserted) {
14                 return false;
15             }
16         }
17         return isAsserted;
18     } else if (policyComponent instanceof AbstractSecurityAssertion) {
19         AbstractSecurityAssertion abstractSecurityAssertion = (
               AbstractSecurityAssertion) policyComponent;
20         return abstractSecurityAssertion.isAsserted(assertionStateMap);
21     } else if (policyComponent == null) {
22         throw new WSSPolicyException("Policy not found");
23     } else {
24         throw new WSSPolicyException("Unknown PolicyComponent: " +
               policyComponent + " " + policyComponent.getType());
25     }
```

```
26 }
```

Listing 4.5: Policy Verifizierung

Die Verifizierung jeder einzelnen Assertions kann generell in der AbstractSecurityAssertion Klasse durchgeführt werden:

```
1  public abstract class AbstractSecurityAssertion implements Assertion {
2      ...
3      public abstract SecurityEvent.Event[] getResponsibleAssertionEvents
           ();
4
5      public abstract void getAssertions(Map<SecurityEvent.Event,
           Collection<AssertionState>> assertionStateMap);
6
7      public boolean isAsserted(Map<SecurityEvent.Event, Collection<
           AssertionState>> assertionStateMap) {
8          boolean asserted = true;
9          SecurityEvent.Event[] secEvents = getResponsibleAssertionEvents
               ();
10         for (int i = 0; i < secEvents.length; i++) {
11             SecurityEvent.Event securityEvent = secEvents[i];
12
13             Collection<AssertionState> assertionStates =
                   assertionStateMap.get(securityEvent);
14             for (Iterator<AssertionState> assertionStateIterator =
                   assertionStates.iterator(); assertionStateIterator.
                   hasNext();) {
15                 AssertionState assertionState = assertionStateIterator.
                       next();
16                 if (assertionState.getAssertion() == this) {
17                     asserted &= assertionState.isAsserted();
18                 }
19             }
20         }
21         return asserted;
22     }
23 }
```

Listing 4.6: Assertion Validierung

Eine entscheidende Zeile ist die Zeile 16. Über `assertionState.getAssertion() == this` wird entschieden, ob die aktuelle Assertion zu aktuellen durchlaufenen Policy-Alternative gehört. Falls die Methode bei einer Assertion “false” zurückgibt, ist die aktuell zu validierende Policy-Alternative nicht erfüllt.

5 Auswertung

Streaming-basierte Verarbeitung von XML-Dokumenten verspricht höhere Performance im Vergleich zu DOM. Wir wollen hier zeigen ob sich dies bewahrheitet und wie gross die Unterschiede sind. Als weiteren Punkt wollen wir zeigen wie es mit der Resistenz gegen die bekannten Angriffe aussieht.

5.1 Umgebung und Messverfahren

Die Messungen wurden auf einem Computer mit Intel Core2Duo T7250 2.00GHz Prozessor und 4GB Arbeitspeicher durchgeführt. Als Plattform kam ein 64-Bit Linux 2.6.35.4 und das Sun-JDK 1.6.0_22 zum Einsatz. Die maximale Heap-Size der Java-VM wurde mit "-Xmx2000m" auf 2 Gigabytes limitiert.

Für die Messungen des Heap-Speicherverbrauchs musste tief in die Trickkiste gegriffen werden. Dies stellte sich als grössere Herausforderung dar als zuerst angenommen. Diverse Strategien sind in Betracht gezogen worden um den Speicherverbrauch messen zu können:

Java Runtime Über die Klasse Runtime kann der aktuelle Heap-Speicherverbrauch ausgelesen werden. Über folgende Rechnung erhält man den aktuell benutzten Speicher:

```
System.gc();
Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
```

Wichtig ist hierbei, dass immer vorher der Garbage Collector aufgerufen wird. Ansonsten werden auch unreferenzierte (nicht mehr gebrauchte Objekte) mitgemessen. Ob der Speicher wirklich freigegeben wird, hängt wahrscheinlich von der JVM Implementierung ab. Die API-Dokumentation zum Sun-JDK 1.6 äussert dazu:
"When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects."
Die nächste Frage die sich hier stellt ist, wo soll dieser Aufruf stattfinden? Am Ende der Verarbeitung, während dem Einlesen des nächsten XML-Events?

Profiler Mit einem Profiler kann ziemlich viel aus einer laufenden VM ausgelesen werden. Die Erfahrung hat gezeigt, dass die Werte zum aktuellen Speicherverbrauch ziemlich inakkurat sind. Viele Objekte können während der Messung immer noch auf dem Heap liegen und sind noch nicht dem GC (Garbage Collector) zum Opfer gefallen. Somit werden auch "tote" Objekte mitgemessen.

Thread Alle bisher vorgestellten Möglichkeiten den Speicherverbrauch zu messen, waren unbefriedigend. Daher kam die Idee mit Hilfe eines Threads die Messungen durchzuführen:

```
Runnable myRunnable = new Runnable() {
  public void run() {
    int sleepTime = 100;
    while (!threadStopper.isStop()) {
      try {
        Thread.sleep(sleepTime);
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
      System.gc();
      memory.add(((int)((Runtime.getRuntime().totalMemory() - Runtime.
          getRuntime().freeMemory()) / 1024 / 1024)));
    }
  }
};
```

Listing 5.1: Methode zum Messen des Speicherverbrauchs

Der Thread wurde jeweils vor den Messungen mit einer leicht höheren Priorität gestartet, was zu einer relativ konstanten Sampling-Rate führte. Die längeren Laufzeiten, die sich durch den zusätzlichen Thread ergaben, sind für die Speichermessungen irrelevant. Am Schluss, wenn das Dokument verarbeitet worden ist, wurde aus allen Samples den grössten Wert ermittelt: Der maximale Speicherverbrauch zu einem bestimmten Zeitpunkt. Die Aussage, die daraus getroffen werden kann: Wie viele SOAP-Nachrichten können gleichzeitig verarbeitet werden bis der Heap-Speicher voll ist? Angenommen wir haben 100 Megabytes an Heap-Speicher zur Verfügung. Die Nachrichten werden per DOM verarbeitet. Jede Nachricht benötigt in der Spitze 50 Megabytes Speicher. Somit können maximal 2 Nachrichten gleichzeitig verarbeitet werden.

5.2 DOM versus Streaming

Um Vergleiche machen zu können, wurde das Streaming-WebServices-Security-Framework mit Apache WSS4J [con10d] verglichen. WSS4J baut auf Apache Santuario (xmlsec) [sc10] auf und verwendet DOM als Datenmodell. Bei den Tests wurden bei den SOAP-Nachrichten jeweils der ganze SOAP-Body mit mit AES-256 verschlüsselt. Die maximale Tiefe des XML im SOAP-Body lag bei 9. Der SOAP-Body wurde mit sich wiederholenden Strukturen gefüllt.

5.2.1 Entschlüsselung

Abbildung 5.1 zeigt die Verarbeitungszeit beider Modelle beim Entschlüsseln. Auffällig ist die Linearität beim streaming-basierten Modell. Das DOM Modell scheint auch eine gewisse Linearität vorhanden zu sein, die aber vermutlich gestört wird durch die Speicherverwaltung da viel Speicher alloziert und wieder freigegeben werden muss. Je mehr man sich der maximalen Heap-Grösse nähert, umso grösser wird der Overhead, da nun viel häufiger der Garbage

Collector zum Einsatz kommt und die Suche nach freien Speicherblöcken immer aufwändiger wird. Ersichtlich ist, dass der Laufzeitunterschied bei kleinen Dokumenten immer geringerer wird. Je grösser die Dokumente werden, desto grösser wird der Laufzeitunterschied zu Gunsten der streaming-basierten Implementierung. Die letzte Messung mit etwa 460'000 XML Elementen konnte bei der DOM Implementierung nicht mehr durchgeführt werden, da eine OutOfMemory Exception auftrat. Dies entspricht ungefähr einem 43 Megabytes grossem XML Dokument.

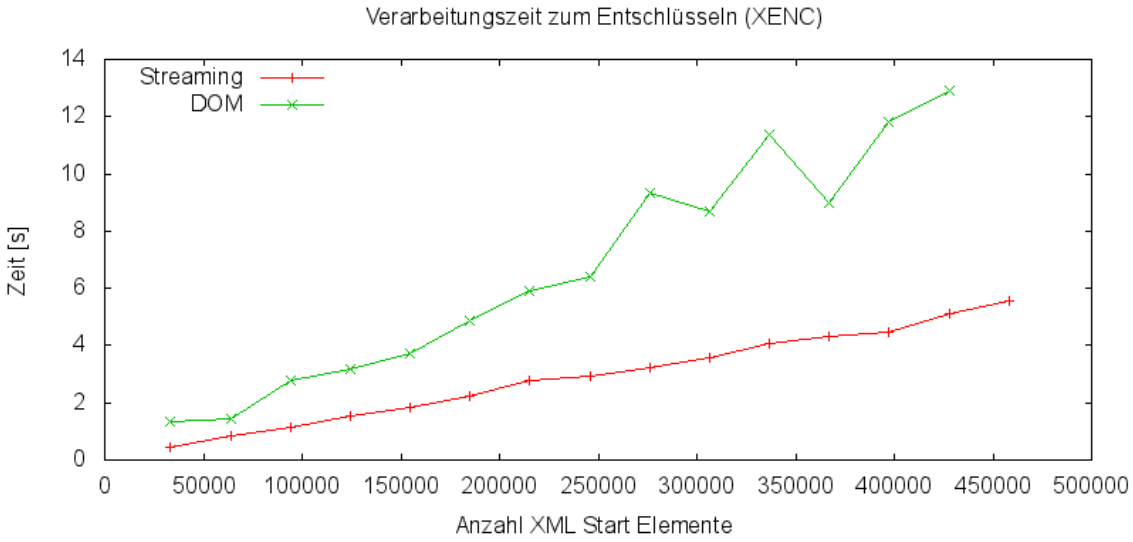


Abbildung 5.1: Vergleich Geschwindigkeit beim Entschlüsseln

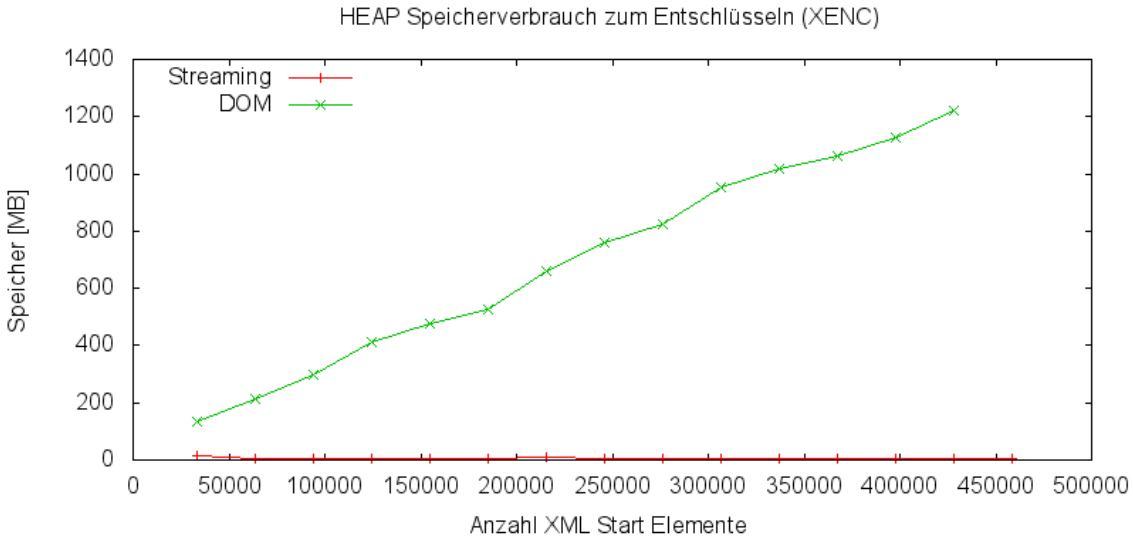


Abbildung 5.2: Vergleich Speicherverbrauch beim Entschlüsseln

Betrachtet man die Abbildung 5.2, erkennt man sofort den grossen Unterschied beider Modelle bezüglich Speicherverbrauchs. Beim DOM Modell steigt der Speicherverbrauch linear mit der Anzahl XML-Elementen an. Dabei wird klar, dass der vorhandene Speicher zum limitierenden Faktor wird. Im Gegensatz dazu zeigt die streaming-basierte Implementierung durchgängig einen sehr kleinen Speicherverbrauch. Egal wie gross die XML Dokumente wa-

ren, es wurde im Schnitt etwa 5 Megabytes, mit "Ausrutschern" maximal 13 Megabytes gemessen.

5.2.2 Verschlüsselung

Auf der ausgehenden Seite, also bei der Verschlüsselung, sieht es sehr ähnlich aus. Während die streaming-basierte Implementierung eine lineare Kurve aufweist, zeigt das DOM Modell wieder eine gewisse "Unruhe" in der Verarbeitungszeit [Abbildung 5.3]. Erklärbar dürfte dies ebenfalls wieder durch den hohen Speicherverbrauch und den öfteren Einsatz des Garbage Collector sein.

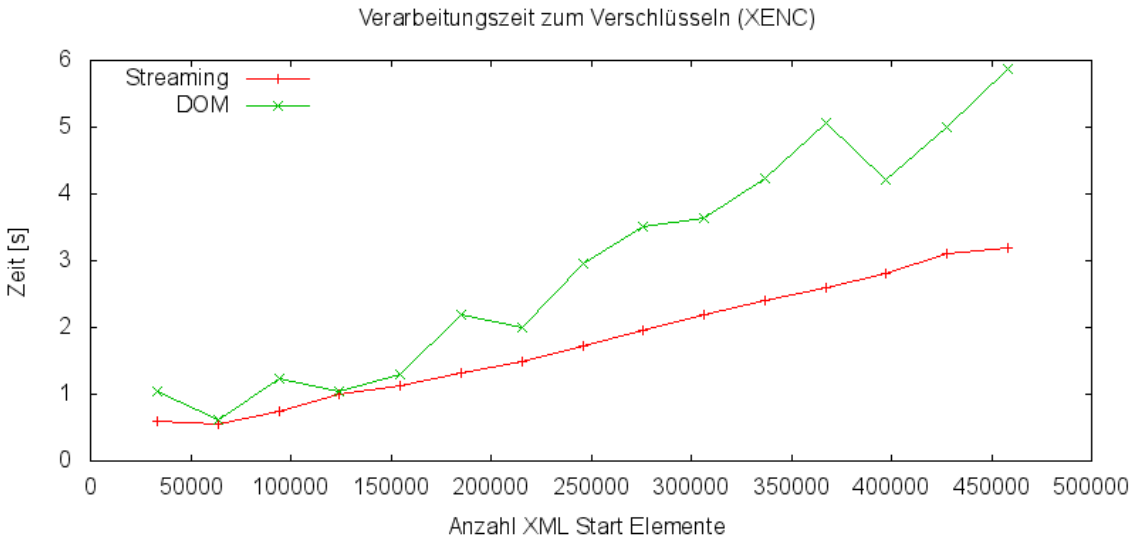


Abbildung 5.3: Vergleich Geschwindigkeit beim Verschlüsseln

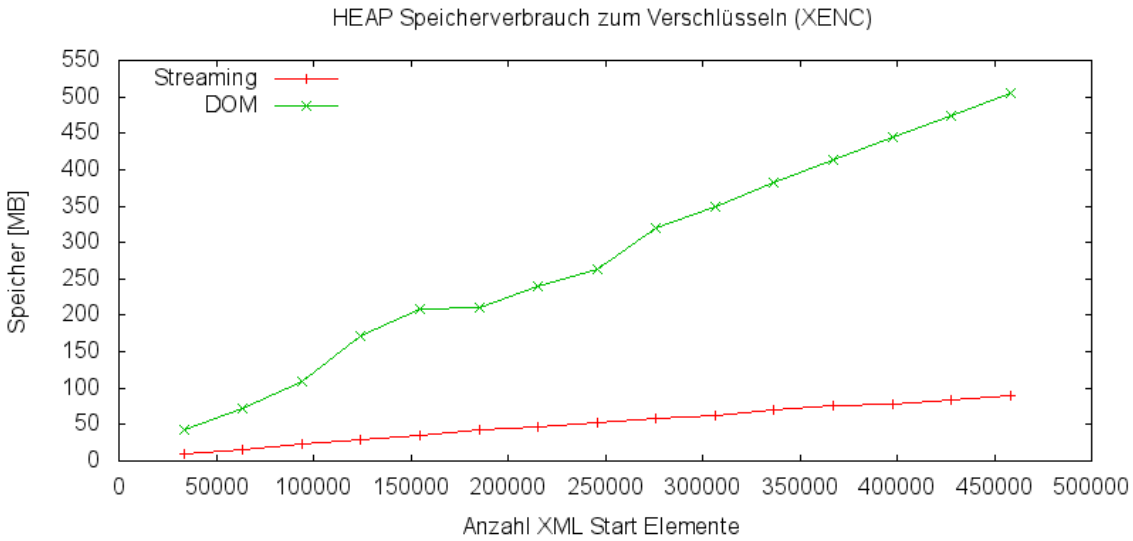


Abbildung 5.4: Vergleich Speicherverbrauch beim Verschlüsseln

In der Abbildung 5.4 ist der Speicherverbrauch auf der ausgehenden Seite zu sehen. Obwohl bei beiden Implementierungen das ganze XML im Speicher gehalten werden muss, scheint die streaming-basierte Lösung trotzdem um einiges effizienter mit dem Speicher umzugehen. Begründung dafür dürfte sein, dass bei der streaming-basierten Implementierung "nur" die unabhängigen XMLEvents im Speicher gehalten werden, im Unterschied zu DOM wo ein ganzer navigierbarer Baum mit allen Metainformationen im Speicher gehalten wird.

5.3 Policy-Validierung Performance

Erwartungsgemäss ist eine der grossen Stärken der streaming-basierten Implementierung wenn es um das "fail-fast" Verhalten geht. Abbildung 5.5 zeigt die Zeit bis die Policy Verletzung detektiert wird. Bei diesem Test Szenario wurde eine Policy spezifiziert, die die Verschlüsselung des ganzen SOAP-Body erzwingt. Die gesendete SOAP-Nachricht wurde aber nur partiell verschlüsselt. Aus dem Graph lässt sich ablesen, dass beim DOM Modell immer zuerst die ganze Nachricht eingelesen, danach entschlüsselt und erst dann die Policy überprüft wird/werden kann. Daher die stetig wachsenden Verarbeitungszeiten mit immer grösseren Dokumenten. Beim streaming-basierten Modell hingegen wird unmittelbar beim Verarbeiten des SOAP-Body detektiert, dass dieser nicht verschlüsselt ist aber sein müsste. Dies führt zum sofortigen Abbruch der Verarbeitung, was Computer-Ressourcen spart und somit die Gefahr einer DoS-Attacke minimiert.

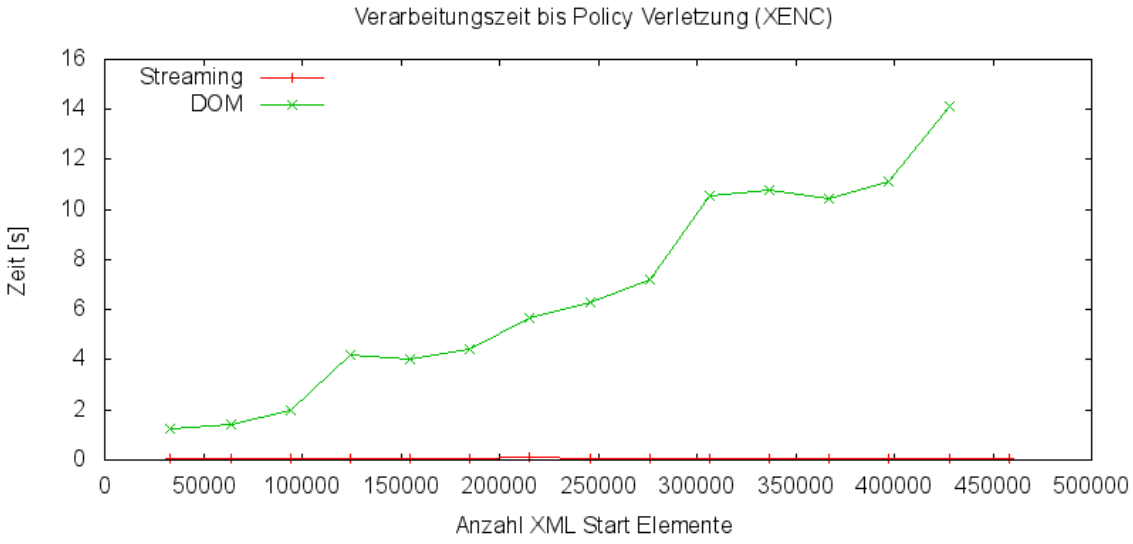


Abbildung 5.5: Zeit bis zum Abbruch bei einer Policy-Verletzung

5.4 Schutz gegen die bekannten Angriffe

Im Bezug auf die im Kapitel 2.4 erwähnten Angriffsarten wird im Folgenden besprochen, wie sich das Streaming-WebServices-Security-Framework in diesen Situationen verhält.

5.4.1 Allgemeine DoS Attacken

Die allgemeinen DoS Attacken werden durch die streaming-basierte Architektur gemildert aber sicher nicht vollständig verhindert. Da mit dem StAX-Parser das XML "portionsweise" eingelesen wird, ist jederzeit die Möglichkeit vorhanden, den Vorgang abzubrechen. Dies kann erreicht werden indem zum Beispiel die Anzahl Elemente und die Tiefe des XML mitgezählt wird. Eine Limitierung der Anzahl gleichzeitig zu verarbeitender Dokumenten, um eine Überlastung zu verhindern, ist schwierig im Streaming-Framework zu verhindern. Dies wird am besten mit einer vorgeschalteten Firewall gelöst.

5.4.2 XML Element Wrapping Attacken

Bei der Anwendung von WS-Security ohne weitere Massnahmen ist es nicht möglich diese Angriffsart zu verhindern. Die Referenzen zu den verschlüsselten Daten sind immer noch gültig und eindeutig. Aus der Sicht von WS-Security ist alles normal abgelaufen. Das Dokument [MA05] beschreibt Gegenmassnahmen die getroffen werden können, um "Wrapping-Angriffe" zu verhindern. Eine Massnahme davon ist der Einsatz von einer gut ausgearbeiteten WS-SecurityPolicy. Im Streaming-WebServices-Security-Framework kann dieser Angriff durch die entsprechende Policy verhindert werden:

```
<wsp12:Policy>
  ...
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
  <sp:EncryptedParts>
    <sp:Body/>
  </sp:EncryptedParts>
  ...
</wsp12:Policy>
```

Zusätzlich sollte vom Web Service Framework eine XML-Schema Validierung der SOAP-Nachricht durchgeführt werden, um sicherzustellen dass der SOAP-Body nicht verdoppelt wurde.

5.4.3 Rekursive Schlüssel Referenzen

In der aktuellen Version des Frameworks werden die Schlüssel Referenzen sofort aufgelöst, um die Schlüssel für die Entschlüsselung bereit zu halten. Daher läuft dieser Angriff, mit einer Exception dass der Schlüssel nicht gefunden wurde, ins Leere. Dieser Angriff muss aber in zukünftigen Versionen im Auge behalten werden. Zusätzliche Vorsichtsmassnahme wäre die Limitierung der Anzahl von Referenzierungen.

5.4.4 Externe URI Referenzen

Externe URI-Referenzen werden in der aktuellen Version des Frameworks nicht unterstützt. Daher ist dieser Angriff wirkungslos. Falls diese Eigenschaft trotzdem einmal benötigt würde, wäre eine Gegenmassnahme das Whitelisting. Nur bekannte und Vertrauenswürde URL's zulassen.

5.4.5 Web Services und DTD Verarbeitung

Gegen die DTD Angriffe gibt es eine wirkungsvolle Lösung: Beim XML-Parser deaktivieren. Wie bereits in der Beschreibung der Angriffe gesehen, ist eine DTD Deklaration in SOAP-Nachrichten nicht erlaubt. Deshalb ist es legitim die DTD Verarbeitung einfach zu deaktivieren.

5.4.6 XSL Transform Exploitation

XSLT Unterstützung ist in der aktuellen Version des Frameworks nicht eingebaut und somit ist dieser Angriff nicht möglich. Ob dies jemals unterstützt werden soll, ist fraglich da dies sehr gefährlich sein kann, wie man im Beispiel gesehen hat. Dies sollte, wenn überhaupt, nur in vertrauenswürdigen Umgebung eingesetzt werden.

5.4.7 SOAPAction spoofing

SOAPAction spoofing ist im Zusammenhang mit Policies "interessant". Um die Möglichkeit der Umgehung von Policies zu verhindern, wird in der aktuellen Version immer auch zusätzlich die Web Service Operation verglichen. Nur wenn ein passendes Paar SOAPAction und Operation gefunden wird, wird die Policy angewendet. Im anderen Fall wird die weitere Verarbeitung abgebrochen.

6 Schlussfolgerung und Ausblick

In dieser Arbeit wurde die streaming-basierte Verarbeitung von gesicherten SOAP-Nachrichten behandelt. Im Kapitel 2 wurden die verschiedenen Standards und Technologien vorgestellt auf denen diese Arbeit aufbaut. Ebenso wurden bekannten Angriffsmöglichkeiten aufgezeigt, die bei einer Implementierung zu berücksichtigen sind. In der konzeptionellen Phase (Kapitel 3) der Arbeit wurden Rahmenbedingungen, Anforderungen und Architektur festgelegt, welche dann in der Implementierung umgesetzt wurden. Die entstandene Implementierung des Streaming-WebServices-Security-Framework beherrscht das Verschlüsseln, das Entschlüsseln und die Policy-Validierung. Um die Korrektheit und Interoperabilität zu Gewährleisten wurde mit dem WSS4J [con10d] Framework getestet und verglichen.

Mit der erarbeiteten Implementierung des Streaming-WebServices-Security-Framework wurden Vergleiche mit dem DOM-basierten WS-Security Modell WSS4J durchgeführt die die erwarteten Performance Vorteile bestätigen sollen. Dabei haben wir gesehen, dass der streaming-basierte Ansatz einen spürbaren Vorteil bringt bezüglich Verarbeitungszeit und Speicherverbrauch. Je grösser die Dokumente werden umso mehr lohnt sich der streaming-basierte Ansatz.

Bei der Implementierung von WS-SecurityPolicy wurde eine lose gekoppelte Architektur gewählt, um eine gute Integration in Web Service Frameworks zu ermöglichen. Die Integration in die Web Service Frameworks ist ebenso wichtig wie die Integration in die WS-Security-Frameworks, da wie bereits erwähnt wurde [3.2.5], viele Assertions nur mit Hilfe des "Wissens" des Web Service Framework korrekt erfüllt werden können. Trotzdem ist es gelungen die gesetzten Ziele zu erreichen. Der schnelle Abbruch ("fail-fast") der Verarbeitung bei Policy Verletzungen ist gegeben.

Mit der Architektur des Streaming-WebServices-Security-Framework wurden die Grundbausteine gelegt um es einfach erweitern zu können. Erweiterungen wie Signaturen, Tokens (SAML Token, UsernameToken) et cetera lassen sich Problemlos integrieren und wurden teilweise zu Demonstrationszwecken bereits umgesetzt.

Sinnvolle zukünftige Ergänzung des Frameworks könnte die Unterstützung für XPath Ausdrücke sein. Mit XPath Ausdrücken könnten einerseits die verschlüsselten Teile des XML referenziert werden und andererseits Policies mit XPath Ausdrücken verifiziert werden.

Da in dieser Arbeit nur die Policy Verifizierung behandelt wurde, d.h die Policy Validierung auf der eingehenden Seite, wäre eine weitere Ergänzung das automatische Einrichten und Konfigurieren des Web Service Clients via Policy. Dazu könnte ähnlich vorgegangen werden wie auf der Server Seite. Der Client müsste aus allen Policy-Alternativen die Policy finden, die er anhand von seinen bestehenden Gegebenheiten, wie Schlüsselmaterial, et cetera erfüllen kann und diese auf das SOAP-Dokument anwenden.

Literaturverzeichnis

[Apa] The Apache Software Foundation. *Apache CXF*. http://cxf.apache.org.

[Apa10] Apache. *Apache Commons Codec*, 2010. http://commons.apache.org/codec.

[BCD+06] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. *SPEX: XPath Evaluation against XML Streams*, 2006. http://spex.sourceforge.net.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. W3C note, W3C, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[Bid09] Renaud Bidou. Attacks on web services. Technical report, The OWASP Foundation, may 2009. http://www.owasp.org/images/6/6b/2009-05-06-OWASPFR-WebServices.pdf.

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0. W3C recommendation, W3C, November 2008. http://www.w3.org/TR/xml.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[con10a] The Apache Neethi contributors. *Apache Neethi*, 2010. http://ws.apache.org/commons/neethi/.

[con10b] The Bouncy Castle contributors. *Bouncy Castle Crypto APIs*, 2010. http://www.bouncycastle.org.

[con10c] The Woodstox contributors. *Woodstox - High-performance XML processor*, 2010. http://woodstox.codehaus.org.

[con10d] The WSS4J contributors. *Apache WSS4J*, 2010. http://ws.apache.org/wss4j.

[HBN+04] Hugo Haas, David Booth, Eric Newcomer, Mike Champion, David Orchard, Christopher Ferris, and Francis McCabe. Web services architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[HD10] Frederick Hirsch and Pratik Datta. XML signature best practices. W3C working draft, W3C, August 2010. http://www.w3.org/TR/xmldsig-bestpractices/.

[Her10] Sascha Herzog. Xml external entity attacks (xxe). Technical report, The OWASP Foundation, oct 2010. http://www.owasp.org/images/5/5d/XML_Exteral_Entity_Attack.pdf.

[Hil07] Bradley W. Hill. Command injection in xml signatures and encryption. Technical report, Information Security Partners, jul 2007. https://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf.

[IDS02] Takeshi Imamura, Blair Dillaway, and Ed Simon. XML encryption syntax and processing. W3C recommendation, W3C, December 2002. http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[LK06] Kelvin Lawrence and Chris Kaler. Web services security: Soap message security 1.1 (ws-security 2004). Technical report, OASIS Open, February 2006. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[MA05] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 20–27, New York, NY, USA, 2005. ACM.

[Meg] D. Megginson. SAX 2.0: The Simple API for XML. http://www.saxproject.org.

[NGG+07a] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Ws-secureconversation 1.3. oasis standard. Technical report, March 2007. http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html.

[NGG+07b] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Ws-trust 1.3. oasis standard. Technical report, March 2007. http://docs.oasis-open.org/ws-sx/ws-trust/v1.3/ws-trust.html.

[NGG+09] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Ws-securitypolicy 1.3. oasis standard. Technical report, February 2009. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/os/ws-securitypolicy-1.3-spec-os.html.

[OAS] Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/home/index.php.

[RSH+08] Thomas Roessler, David Solo, Frederick Hirsch, Donald Eastlake, and Joseph Reagle. XML signature syntax and processing (second edition). W3C recommendation, W3C, June 2008. http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/.

[San02] Aleksey Sanin. possible dos attack, apr 2002. http://lists.w3.org/Archives/Public/xml-encryption/2002Apr/0055.html.

[sc10] The santuario contributors. *Apache santuario*, 2010. http://santuario.apache.org/Java/index.html.

[StA] Streaming API for XML. http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/tutorial/doc/StAX.html#wp69937.

[VYO+07] Asir S Vedamuthu, Ümit Yalçinalp, David Orchard, Toufic Boubez, Frederick Hirsch, Prasad Yendluri, and Maryann Hondo. Web services policy 1.5 - attachment. W3C recommendation, W3C, September 2007. http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904.

[W3Ca] The World Wide Web Consortium. http://www.w3.org.

[W3Cb] Document object model (DOM). http://www.w3.org/DOM/.

[wc10] The wsdl4j contributors. *The Web Services Description Language for Java Toolkit*, 2010. http://sourceforge.net/projects/wsdl4j/.

[Web] Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[YHV+07] Prasad Yendluri, Maryann Hondo, Asir S Vedamuthu, Frederick Hirsch, David Orchard, Ümit Yalçinalp, and Toufic Boubez. Web services policy 1.5 - framework. W3C recommendation, W3C, September 2007. http://www.w3.org/TR/2007/REC-ws-policy-20070904.