Wer bin ich?

SAML 2.0, ein Tutorium – Teil 2: Praxis

Michael Kain, Guido Keller

Die Security Assertion Markup Language 2.0 (SAML) ermöglicht den Austausch von Authentifizierungs- und Autorisierungsinformationen über Organisationsgrenzen hinweg. Trotz der umfangreichen OASIS-Spezifikation ist ein schneller Einstieg in die SAML 2.0 möglich. Dieses zweiteilige Tutorium begann mit einer allgemeinen, theoretischen Einführung und endet in diesem zweiten Teil mit einem anschaulichen Beispiel.

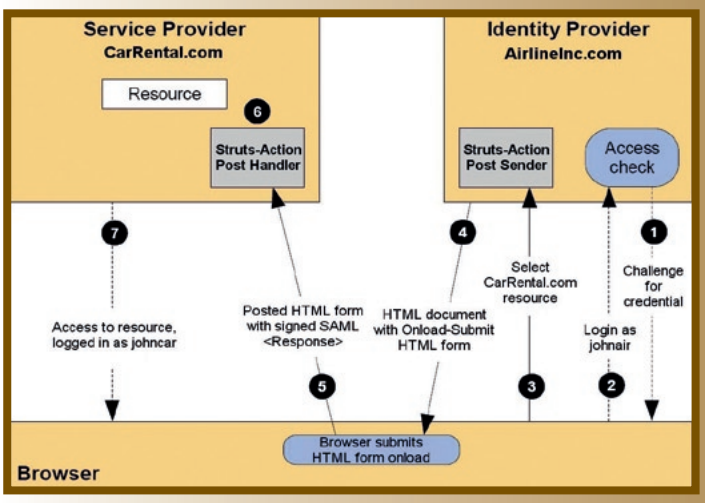

Abb. 1: Übersichtsdiagramm

Motivation

Nachdem Teil 1 dieses Tutoriums [KaiKel07] sich sehr breit mit vielen theoretischen Aspekten der SAML 2.0 befasst hat, beschränkt sich Teil 2 fast ausschließlich auf ein praktisches Anschauungsbeispiel. Ziel dieses Beispieles ist es, die theoretischen Begriffe der SAML 2.0 in ein konkretes Implementierungsszenario zu überführen. Dem eher praktisch orientierten Leser soll dadurch das Verständnis erleichtert werden.

Des Weiteren wird dem Leser eine mögliche Anleitung an die Hand gegeben, selbst eine Web Browser Single Sign-On(SSO)-Lösung zu realisieren: sicher, auf Basis eines Standards und ohne Kauf eines bestimmten Werkzeuges. Allein die Bordmittel der Java-Web-Welt und das Web-Framework Apache Struts sind dazu erforderlich.

Auf der Homepage des *Security Services Technical Commitee (SSTC)* unter [OasisSSTC] finden sich dazu leider keine praktischen Beispiele. Diese Lücke soll mit diesem zweiten Teil des Tutoriums nun geschlossen werden.

Szenario

Das folgende Szenario soll realisiert werden: Ein Benutzer mit dem Namen Joe hat sich beim Identity Provider (*IdP*) unter AirlineInc.com mit seinem Account *johnair* authentifiziert und ist dort eingeloggt. Nachdem er einen Flug gebucht hat, möchte er im Anschluss einen Leihwagen mieten. Dazu klickt er in seinem Browser auf einen Link unter AirlineInc.com, der ihn direkt zu CarRental.com weiterleitet und dort einloggt, ohne dass er sich erneut authentifizieren muss. CarRental.com ist in diesem Szenario der Service Provider (*SP*).

Aus Gründen der Vereinfachung wird in diesem Szenario vorausgesetzt, dass sich AirlineInc.com und CarRental.com bereits im Vorfeld auf eine Art Pseudonym verständigt haben, anhand dessen sie jeweils in ihren Systemen einen Benutzer wiederfinden können. Überträgt in diesem Szenario z. B. AirlineInc.com das Kürzel *azqu3H7* an CarRental.com, so weiß CarRental.com, dass es sich dabei nur um seinen eigenen Benutzer *johncar* handeln kann. Dieses Kürzel könnte z. B. im Voraus mit Hilfe der Identity Federation von SAML abgeglichen und vereinbart worden sein. Ebenso ist es möglich, dass im Zuge eines Kundendatenabgleichs oder einer Kundendatenmigration solche Kürzel vergeben wurden, anhand derer die Benutzer in dem jeweiligen *Customer Relationship Management System* (CRM) der kooperierenden Firmen aufgefunden werden können.

In der Terminologie von SAML 2. 0 wird dieses Szenario mit Hilfe des Web Browser SSO Profile *IdP-initiated* realisiert; auf Basis von HTTP Post Binding und des Authentication Request Protocol.

Übersichtsdiagramm

Das Diagramm in Abbildung 1 stellt das *HTTP Post Binding* in Verbindung mit den notwendigen Komponenten dar, die in diesem Beispiel für die technische Realisierung verwendet werden. Sowohl der Single Sign-On Service (*Post Sender*) des IdP als auch der Assertion Consumer Service (*Post Handler*) des SP werden als Apache Struts Action-Klassen implementiert. Als Alternative wäre eine Implementierung mit Java Servlets, Java Server Faces oder eine Kombination aus beiden ebenfalls ohne weiteres möglich. Wir haben uns in diesem Beispiel für Struts entschieden aufgrund dessen weiter Verbreitung unter geschäftsrelevanten Webanwendungen. Dies basiert vor allem auf der Annahme, dass bestehende Webanwendungen um eine Single Sign-On-Funktionalität erweitert werden sollen, die bisher schon auf Basis von Struts entwickelt wurden und damit ihren Login-Prozess realisieren.

Digitale Signaturen

Eine weitere elementare technische Komponente wurde in dem Übersichtsdiagramm in Abbildung 1 bewusst vernachlässigt: die digitalen Signaturen und deren Verwendung. Diese überprüft der SP CarRental.com, um sich sicher zu sein, dass der erhaltene SAML Response auf dem Weg vom IdP nicht manipuliert wurde und tatsächlich von AirlineInc. com ausgestellt wurde. Dies setzt aber voraus, dass sowohl der IdP als auch der SP in ihren Struts-Actions Zugriff auf einen Java Keystore haben. Diese sind in dem Diagramm nicht eingezeichnet, um es zu vereinfachen.

Der SP hat in seinem Keystore den öffentlichen Schlüssel des IdP importiert und kann mit dessen Hilfe die Signaturen überprüfen. Der IdP verwendet seinen privaten Schlüssel, um die gesendete SAML Response mit einer Signatur zu versehen. Weiter soll an dieser Stelle nicht auf diese Thematik eingegangen werden. Der interessierte Leser sei deshalb an Sun unter den Links [SunDigSig] und [SunKeytool] verwiesen.

Post Sender

Listing 1 stellt exemplarisch die Implementierung der Struts-Action Post Sender dar, die in Abbildung 1 im Kästchen des IdP AirlineInc.com eingebettet ist. In deren `execute`-Methode wird eine HTML-Seite mittels eines StringBuffers zusammengesetzt und direkt an den Browser des Aufrufers zurückgegeben. Die HTML-Seite enthält ein Formular, das dynamisch zusammengebaut wird. Dieses HTML-Formular wird mittels JavaScript beim Laden der Seite (*onload*) durch den Browser verschickt. Dies geschieht im Hintergrund und ohne Eingreifen des Benutzers.

Das Formular wird an die URL gepostet, die als Wert im `action`-Attribut des Formulars hinterlegt ist. Diese URL wird hier als Request-Parameter übergeben, der den Namen `SERVICE_PROVIDER_URL` hat. Der Wert dieses Parameters enthält die URL der Struts-Action Post Handler des SP, die in Abbildung 1 im linken Kästchen eingebettet ist. Das Formular beinhaltet zwei *hidden-fields*, deren Wert-Attribute *base64* kodiert sind: eine SAML 2.0 Response und deren Signatur. In der hier nicht näher ausgeführten `buildSAMLAssertion`-Methode werden die Benutzerdaten in der SAML Assertion abgelegt.

Listing 2 stellt den Inhalt der HTML-Seite dar, die der Post Sender an den Browser zurückgibt. Listing 3 stellt ausschnittsweise die im HTML-Formular enthaltene SAML 2.0 Response dar. Diese bescheinigt, dass das Subjekt *azqu3H7@AirlineInc.com* erfolgreich durch AirlineInc.com (*Issuer*) authentifiziert wurde und ein Kreditlimit in Höhe von 500 Euro hat. Dieser Nachweis der Authentifizierung ist bestimmt für CarRental.com (*Audience*).

JAXB

Die in Listing 1 in der Methode `buildSAMLResponse` verwendeten Klassen `SAMLResponse` und `SAMLAssertion` wurden mit JAXB (Java Architecture for XML Binding) erzeugt. Sie dienen dazu, das XML-Objekt-Mapping zu vereinfachen. Der ANT-Task in Listing 4 erzeugt aus der SAML 2.0 XML Schema-Datei alle notwendigen Java-Klassen, um später auf Java-Ebene bequem Objekte erzeugen zu können und diese Schema-konform zu serialisieren. Umgekehrt kann damit Schema-konformes XML in Java-Objekte deserialisiert werden.

Der ANT-Task referenziert die in Listing 5 aufgeführte .xjb-Datei. Diese Datei hat die Aufgabe, das Binding überhaupt zu ermöglichen, indem es Namenskonflikte bei der Generierung auflöst. In diesem Fall sollen bei der Erzeugung der Java-Klassen diese in definierte Pakete gemappt werden, die der zugrunde liegenden Schema-Datei entsprechen. Die Schema-Dateien in Listing 5 können auf der Homepage des Standards unter [OasisSSTC] heruntergeladen werden.

Post Handler

Der Post Handler in Listing 6 ist beinahe das umgekehrte Gegenstück des Post

```
...
public class PostSender extends Action {

public ActionMapping execute(final HttpServletRequest request,
                             final ActionMapping mapping,
                             final ActionForm form,
  ...
  final StringBuffer pageBuffer = new StringBuffer ();
  pageBuffer.append(
    "<html><meta http-equiv='content-type' content='text/html'>");
  pageBuffer.append("<body onload='document.forms[0].submit()'>");
  pageBuffer.append("<form method='POST' action='"
    + request.getParameter("SERVICE_PROVIDER_URL") + "'>");

  String samlResponse;
  byte[] signature;
  ...
    samlResponse = buildSAMLResponse(request);
    signature = getSignature(samlResponse);
  ...
  pageBuffer.append("<input type='hidden' name='SAMLResponse' value='"
    + Util.byteArrayToBase64(samlResponse.getBytes()) + "'>");
  pageBuffer.append("<input type='hidden' name='Signature' value='"
    + Util.byteArrayToBase64(signature) + "'>");
  pageBuffer.append("</form></body></html>");

  final PrintWriter writer = response.getWriter();
  writer.print(pageBuffer.toString());
  writer.flush();
  writer.close();

  return null;
}
private String buildSAMLResponse(final HttpServletRequest request)
throws SAMLException {
  final SAMLResponse response = new SAMLResponse();
  response.setStatusCode(SAMLResponse.URN_SUCCESS);
  ...
  final SAMLAssertion assertion = buildSAMLAssertion(request);
  response.setAssertion(assertion);
  ...
  final String xml = response.toXml();
  return xml;
}
...
```

Listing 1: Struts-Action Post Sender

```
<html>
  <meta http-equiv=Content-Type content='text/html'>
    <body onload='document.forms[0].submit()' >
      <form method='POST'
            action='http://www.carrental.com/posthandler.do'>
        <input type='hidden' name='SAMLResponse'
               value='PD94bWwgdmVyc2206U39.....'>
        <input type='hidden' name='Signature'
               value='FSDh3DFzqALuPDDHQG6uLxPM.....'>
      </form>
    </body>
</html>
```

Listing 2: Post Sender HTML-Seite

```
<saml:Response
...
  <saml:Status>
    <saml:StatusCode
      Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
  </saml:Status>

  <saml:Assertion
  ...
    <saml:Issuer>AirlineInc.com</saml:Issuer>
    <saml:Subject>
      <saml:NameID Format=
        "urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
          azqu3H7@AirlineInc.com
      </saml:NameID>
      ...
    <saml:AudienceRestriction>
      <saml:Audience>CarRental.com</saml:Audience>
    </saml:AudienceRestriction>
    ...
    <saml:AttributeStatement
    ...
      <saml:Attribute Name="CreditLimit"
                      NameFormat="urn:x-airline:attributes">
        <saml:AttributeValue xsi:type="xsd:string">500</saml:AttributeValue>
      </saml:Attribute>
    </saml:AttributeStatement>
  </saml:Assertion>
</saml:Response>
```

Listing 3: SAML 2.0 Response

Sender, was die Arbeitsschritte seiner Implementierung betrifft. Er empfängt in seinem Request das HTML-Formular, das der Browser an ihn gepostet hat. Daraus entnimmt er den SAML 2.0 Response und dessen Signatur, hebt bei beiden die *base64*-Kodierung auf und prüft die Signatur gegen den Response. Im Anschluss erfolgt die Deserialisierung. Mit den daraus erhaltenen Kundendaten kann der Kunde anschließend aus dem Subjekt-Element der SAML Assertion im eigenen CRM aufgefunden werden. Wie in diesem Beispiel durch sein Kürzel *azqu3H7*. Nach dem Durchlaufen dieser Action sollte der Kunde vollständig eingeloggt und z. B. das Kreditlimit von 500 Euro berücksichtigt sein.

Fazit

Wie das Implementierungsbeispiel in diesem Artikel zeigt, muss man nur sehr wenig Code selbst schreiben, um eine solche Single Sign-On-Lösung zu realisieren. Die Schwierigkeit liegt eher darin, mehrere Werkzeuge präzise miteinander kombinieren zu müssen. Wer sich bereits mit JAXB und dem digitalen Signieren in Java auskennt, sollte damit allerdings keine Probleme haben. Der Einarbeitungsaufwand in die SAML 2.0 sollte sich ebenfalls – anhand der Code-Beispiele in den vorangegangenen Listings und der theoretischen Inhalte in Teil 1 dieses Tutoriums – in einem sehr überschaubaren Rahmen bewegen. Alles in allem also Grund genug, Single Sign-On auf diese Art und Weise umzusetzen.

Sollte man sich als Unternehmen für diese Art der Kooperation mit einem anderen Unternehmen entscheiden, so ist dies zu begrüßen. Technisch betrachtet schafft man sich damit eine solide Basis, um solche Kooperationen zukünftig ebenfalls mit anderen Unternehmen zu etablieren. Bei der zeitlichen Planung allerdings sollte man niemals den kommunikativen und organisatorischen Aufwand eines solchen Softwareprojektes unterschätzen. Allein das Austauschen und Beantragen von Zertifikaten für mehrere Serverinstanzen kann sich über Unternehmensgrenzen hinweg als zeitaufwändig gestalten. Dies gilt ebenfalls für Netzwerk-technische Konfigurationen und das Testen als solches.

Links

[KaiKel07] M. Kain, G. Keller, SAML 2.0, ein Tutorium – Teil 1: Theorie, in: JavaSPEKTRUM, 5/2007

[OasisSSTC] OASIS Security Services Technical Commitee, `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`

[SunDigSig] Lesson: Generating and Verifying Signatures, The Java Tutorials, `http://java.sun.com/docs/books/tutorial/security/apisign/index.html`

[SunKeytool] Key and Certificate Management Tool, `http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html`

```
<xjc target="../" extension="true">
  <schema  dir="conf" includes="saml-schema-protocol-2.0.xsd"/>
  <binding dir="conf" includes="*.xjb"/>
</xjc>
```

Listing 4: JAXB ANT-Task

```
<?xml version="1.0"?>
<jxb:bindings version="1.0"
              xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

<jxb:bindings
  schemaLocation=
"http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-core-schema.xsd"
  node="/xs:schema">
  <jxb:schemaBindings>
    <jxb:package name="saml.core"/>
  </jxb:schemaBindings>
</jxb:bindings>

<jxb:bindings schemaLocation=
"http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/xenc-schema.xsd"
  node="/xs:schema">
  <jxb:schemaBindings>
    <jxb:package name="saml.xenc"/>
  </jxb:schemaBindings>
</jxb:bindings>

<jxb:bindings schemaLocation="saml-schema-protocol-2.0.xsd"
              node="/xs:schema">
  <jxb:schemaBindings>
    <jxb:package name="saml.protocol"/>
  </jxb:schemaBindings>
</jxb:bindings>

<jxb:bindings schemaLocation="saml-schema-assertion-2.0.xsd"
              node="/xs:schema">
  <jxb:schemaBindings>
    <jxb:package name="saml.assertion"/>
  </jxb:schemaBindings>
</jxb:bindings>

</jxb:bindings>
```

Listing 5: JAXB Binding-Datei

```
...
public class PostHandler extends Action {
...
public ActionForward execute(final ActionMapping mapping,
                             final ActionForm form,
  final HttpServletRequest request,
  final HttpServletResponse response)
  throws IOException, ServletException {
  ...
    String samlResponse = request.getParameter("SAMLResponse");
    byte[] samlResponseByteArray =
      Util.base64ToByteArray(samlResponse);
    final String signature = request.getParameter("Signature");
    byte[] signatureByteArray = Util.base64ToByteArray(signature);
    ...
    if (validateSignature(samlResponseByteArray,
                          signatureByteArray)) {
      if (validateSAMLResponse(samlResponseByteArray)) {
        final SAMLResponse response = new SAMLResponse(samlResponse);
        final SAMLAssertion assertion = response.getAssertion();
        // Get customer data from CRM and log customer in.
        ...
      }
    }
  ...
  }
}
...
```

Listing 6: Post Handler

Michael Kain ist Consultant im Bereich Enterprise Application Development bei der Unternehmensberatung Resco in Hamburg. Er beschäftigt sich mit der Konzeption und Entwicklung von Webanwendungen.
E-Mail: Michael.Kain@resco.de.

Guido Keller ist freiberuflicher Consultant im Bereich Enterprise Application Development. Sein Schwerpunkt liegt in der Konzeption und Entwicklung von Webanwendungen.
E-Mail: Keller.Guido@resco.de.