Sichere Webanwendungen mit Java - wie fange ich an?

Dominik Schadow
BridgingIT GmbH
Stuttgart

Schlüsselworte
Java Web Security, sichere Webanwendungen, Cross-Site Scripting, Cross-Site Request Forgery

Einleitung
Auch wenn sichere Webanwendungen immer wichtiger wird und immer häufiger unter Entwicklern besprochen wird, stellt sich den meisten doch die Frage, wie man wirklich sicher entwickelt und dieses umfangreiche Thema in die tägliche Arbeit integriert. Gleichzeitig gilt es, möglichst schnell weitere Entwickler für das Thema sichere Softwareentwicklung zu interessieren und alle Komponenten einer Webanwendung gleich sicher zu entwickeln

Wenn man sich als Java-Entwickler zum ersten Mal mit dem Thema "Sichere Entwicklung von Webanwendungen" beschäftigt, stellt man sich zwangsläufig die Frage, wie man denn am besten mit der sicheren Entwicklung anfängt. Zu viele mögliche Sicherheitsprobleme buhlen um Aufmerksamkeit: weithin bekannte wie (SQL-) Injections, unzureichend gesicherte Daten (Passwörter) oder für jedermann erreichbare Seiten in eigentlich geschützten Bereichen. Dazu gesellen sich zahlreiche Spezialthemen und eher unbekanntere Sicherheitsprobleme wie direkte Objektreferenzen oder das Verwenden von fest im Code eingegebenen Credentials. Ranglisten wie die *OWASP TOP 10* [1] oder die *CWE/SANS Top 25 Most Dangerous Software Errors* [2] versuchen, Licht in dieses "Chaos" zu bringen und Entwicklern den Einstieg in die sichere Softwareentwicklung zu erleichtern. Gleichzeitig schaffen diese Listen eine gewisse Awareness für das Thema und sorgen für Gesprächsstoff rund um Sicherheitsprobleme in Webanwendungen. Für viele Entwickler ist eine solche Rangliste der erste Kontakt mit Sicherheitsproblemen in der Softwareentwicklung.

Auch wenn es sehr wahrscheinlich ist, muss selbst ein auf Position 1 gelistetes Sicherheitsproblem nicht zwangsläufig in der eigenen Webanwendung enthalten sein oder gleich gefährlich eingestuft werden. Die Kenntnis des Problems und seine Gegenmaßnahmen schaden sicherlich nicht, allerdings wäre ein anderer Start in die sichere Softwareentwicklung für die eigene Webanwendung effizienter gewesen. Vor allem wenn es zunächst noch gilt, Projektleiter und andere Verantwortliche vom Bedarf der sicheren Softwareentwicklung zu überzeugen. In diesem Fall sollte besser gleich die erste Codeänderung einen zumindest theoretisch nachweisbaren Nutzen bringen.

Ganz und gar Allgemeingültig ist es allerdings nahezu unmöglich, eine Empfehlung zum ersten zu behebenden Problem zu geben. Dazu sind Webanwendungen viel zu individuell; ebenso die möglicherweise folgenden Sicherheitsprobleme. Einen immerhin in den allermeisten Webanwendungen sinnvollen Einstieg in die sichere Entwicklung stellt das Problem *Cross-Site Scripting (XSS)* dar. Und das aus zwei Gründen: Zum einen ist XSS sehr weit verbreitet und damit in vielen Webanwendungen vorhanden. Zum anderen sind die möglichen Auswirkungen (Angriffe) enorm, dient XSS doch ebenfalls als Türöffner für andere Angriffe oder unterwandert den mühevoll implementierten Schutz vor anderen Angriffen. Die Behebung von XSS-Schwachstellen in der eigenen Webanwendung sollte daher stets an erster Stelle stehen.

Cross-Site Scripting (XSS)
Bevor man sich nun an die Behebung von XSS macht empfiehlt es sich, die Datenflüsse der Webanwendung genau zu analysieren und in einem Diagramm zu visualisieren. Ein dafür sehr gut geeignetes und kostenloses Tool ist das *Microsoft Threat Modeling Tool 2014* [3]. Mit diesem Tool lassen sich bereits in der Designphase ein- und ausgehende Datenflüsse in einer für Entwickler hilfreichen Art darstellen (Abbildung 1):

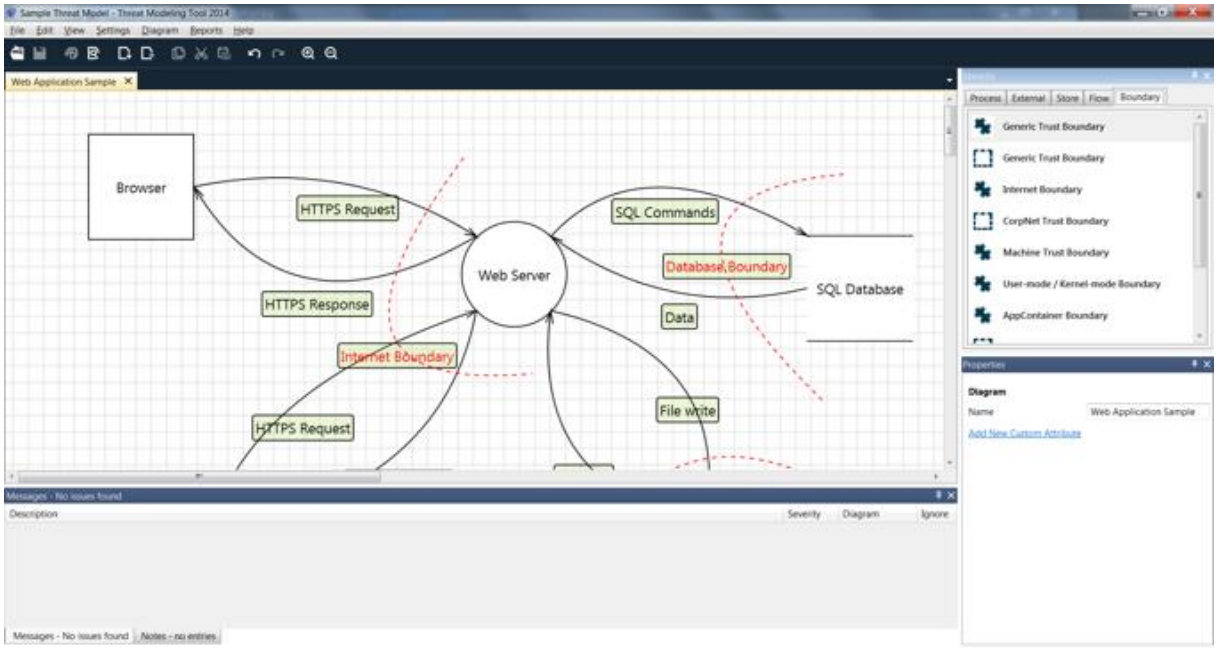

Abb. 1: Microsoft Threat Modeling Tool 2014

Anhand dieser modellierten Datenflüsse findet anschließend die Planung der *Boundaries* und damit sogenannter *Trust Zones* statt. Daten, die eine solche Boundary passiert haben, gelten als validiert und damit als sicher. Mit diesen Informationen lässt sich im Code die dazu notwendige Datenvalidierung an den korrekten Stellen umsetzen. Hier empfiehlt sich die Verwendung einer auf dem *JSR 303 Bean Validation* [4] basierenden Bibliothek wie *Hibernate Validator* [5]. Deren Annotationen machen es verhältnismäßig einfach, Eingabedaten zu validieren. Verhältnismäßig einfach deshalb, weil trotz der vielen im JSR enthaltenen Annotationen nur die *@Pattern* Annotation samt der Eingabe eines regulären Ausdrucks wirklich positive Auswirkungen auf die Sicherheit hat. Annotationen wie *@NotEmpty* oder *@Size(min = 10, max = 14)* dienen zwar der Datenqualität, haben aber gar keinen oder nur einen sehr geringen Einfluss auf die Sicherheit. Ausschließlich ein regulärer Ausdruck kann verhindern, dass z.B. anstelle des Vornamens JavaScript-Schadcode eingegeben wird. Eine Längenbegrenzung der Eingabedaten vermag einen Angriff nur zu erschweren, meist aber nicht vollständig zu verhindern.

Ein solches Threat Model zeigt weiterhin, wo die eingegebenen Daten verwendet und angezeigt werden. Hier gilt es, diese passend zum Kontext (in der Regel im Browser, also HTML, CSS oder JavaScript) zu escapen. Für diese Aufgabe muss eine entsprechende Bibliothek eingesetzt werden. Eine Eigenentwicklung wird niemals alle Fälle und Varianten abdecken. Wegen der zahlreichen unterstützten Kontexte und der einfachen Anwendbarkeit bietet sich der *OWASP Java Encoder* [6] an. Alle potentiell gefährlichen Daten, d.h. alle Daten aus fremden Quellen, werden mit dieser Bibliothek vor der Ausgabe z.B. im Browser neutralisiert. Von einem Angreifer eingegebene Befehle – sei es

HTML oder JavaScript – werden damit im Browser lediglich angezeigt, nicht aber ausgeführt (Listing 1):

```
@WebServlet(name = "OutputEscapedServlet", urlPatterns = {"/escaped"})
public class OutputEscapedServlet extends HttpServlet {
  private Logger logger = LoggerFactory.getLogger(getClass());

  @Override
  protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException {
    String name = request.getParameter("name");
    response.setContentType("text/html");

    try (PrintWriter out = response.getWriter()) {
      out.println("<html>");
      out.println("<body>");
      out.println("<p><strong>For HTML </strong>");
      Encode.forHtml(out, name);
      out.println("</p>");
      out.println("<p><strong>For CSS  </strong>");
      Encode.forCssString(out, name);
      out.println("</p>");
      out.println("<p><strong>For URI </strong>");
      Encode.forUri(out, name);
      out.println("</p>");
      out.println("</body></html>");
    } catch (IOException ex) {
      logger.error(ex.getMessage(), ex);
    }
  }
}
```

Eingabevalidierung und Ausgabeescaping klingen zwar einfach, sind aber aufwendig in der Umsetzung. Neben diesen beiden Fleißaufgaben ist eine simple Konfigurationsanpassung Pflicht. Dabei wird per *web.xml* festgelegt, dass die vom Server vergebene Session ID nur im Cookie übertragen wird und gleichzeitig der Zugriff clientseitiger Skripte auf diese ID blockiert. Das Stehlen einer Session ist auf diesem Weg damit nicht mehr möglich (Listing 2):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... version="3.1">
  <!-- ... -->
  <session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
      <http-only>true</http-only>
      <secure>true</secure>
    </cookie-config>
    <tracking-mode>COOKIE</tracking-mode>
  </session-config>
</web-app>
```

Eine weitere XSS-Gegenmaßnahme mit Hilfe des Browsers stellt die relativ neue *Content Security Policy (CSP)* [7] dar. Mit dieser Policy lässt sich einschränken, von welchen URLs der Browser Skripte, Grafiken oder andere Objekte laden darf. Vor allem aber blockiert der Browser damit immer die Verwendung von Inline-Skripten und Inline-Stilen (CSS). Zahlreiche XSS-Angriffe werden dadurch erschwert bzw. unmöglich. Die Konfiguration ist allerdings aufwendig und kann bei vorhandenen Webanwendungen zahlreiche Codeanpassungen nach sich ziehen. Gleichzeitig muss auch der Browser die Policy kennen und umsetzen. Man sollte sich daher nicht alleine auf die CSP verlassen, sondern diese als zusätzliche Schutzmaßnahme (im Sinne von *Defense-in-Depth*) begreifen.

Das Aktivieren der CSP ist einfach per Response-Header möglich (Listing 3):

```
response.addHeader("Content-Security-Policy", "default-src 'self'");
```

Die eigentliche Arbeit beginnt danach: Das Testen und Wiederherstellen der Funktionalität der eigenen Webanwendung. Mit dem Setzen des Headers ist es damit längst nicht getan, denn ohne Anpassungen im Applikationscode lässt sich die Content Security Policy selten einsetzen.

Die Wirksamkeit all dieser Maßnahmen lässt sich mit einem sogenannten *Intercepting Proxy* durch den Entwickler selbst testen. Dabei hängt sich der Proxy zwischen Browser und eigener Webanwendung und lässt die Manipulation von Requests und Responses zu. So lässt sich feststellen, wie das Backend auf vermeintlich im Frontend validierte – aber per Proxy nachträglich manipulierte –

Daten reagiert. Auch der umgekehrte Weg, also die Manipulation der Response, ist möglich. Hiermit testet man, wie der Browser vermeintlich sichere Daten anzeigt bzw. ob diese evtl. sogar ausgeführt werden. Eine korrekt konfigurierte CSP könnte hierbei noch die Sicherheit der Webanwendung retten. Das dazu speziell auch für Entwickler empfehlenswerte Tool ist *OWASP Zed Attack Proxy (ZAP)* [8] (Abbildung 2):

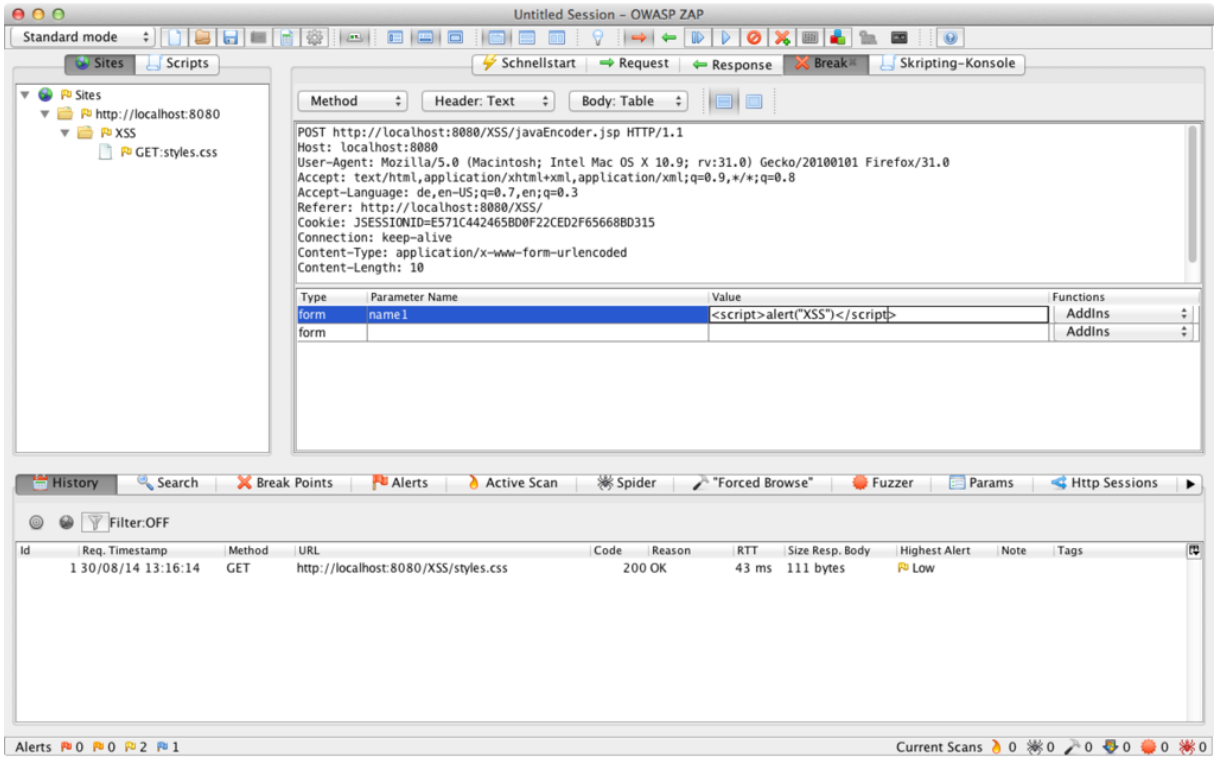

Abb. 2: OWASP ZAP bei der Manipulation eines bereits im Frontend validierten Requests

Cross-Site Request Forgery (CSRF)

Nachdem die Webanwendung vor XSS sicher ist kann man sich dem nächsten meist kritischen Sicherheitsproblem zuwenden: Cross-Site Request Forgery (CSRF). Da eine XSS-Verwundbarkeit jeden CSRF-Schutz unterwandert ist es wichtig, sich zunächst um Cross-Site Scripting zu kümmern. Anschließend gilt es, den direkten Aufruf von Backend-Operationen wie dem Anlegen eines neuen Benutzers oder eine Passwortänderung ohne Verwendung des Frontends zu verhindern. Hierfür muss das Backend in der Lage sein, ordnungsgemäß vom Benutzer ausgefüllte Formulare und dadurch ausgelöste Requests von denen zu unterscheiden, die ein Angreifer einem angemeldeten Benutzer zur Ausführung untergeschoben hat.

Die beste und für den Benutzer bei normaler Verwendung vollkommen unbemerkbare Variante setzt auf ein Token, das pro Nutzer pro Session oder gar pro Request zufällig berechnet wird. Dieses Token wird backendseitig in der Session gespeichert und gleichzeitig als versteckter Wert jedem Formular hinzugefügt. Bei jedem eingehenden Request überprüft das Backend dieses Token. Nur bei Übereinstimmung beider Werte wird die angeforderte Operation ausgeführt. Da ein Angreifer ohne XSS-Verwundbarkeit keinen Zugang zu dem in einem Formular versteckten Token erlangen kann, muss der Request wie vom Entwickler geplant über das Frontend ausgelöst worden sein.

Auch für diese Routineaufgabe gibt es Unterstützung von Frameworks. So bietet Spring Security [9] ab Version 3.2 einen integrierten und per Default aktiven Schutz vor CSRF. Für die Tokenberechnung

ist dabei nichts weiter zu tun. Lediglich das Hinzufügen zu den Formularen muss vom Entwickler selbst gemacht werden. Auch den Vergleich des übermittelten Tokens mit dem in der Session übernimmt Spring Security (Listing 4):

```
<form action="OrderServlet" method="post">
  <input type="hidden" name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
  <label for="product">Product</label>
  <input type="text" id="product" name="product"/>

  <!-- ... -->
  <input type="submit" value="Order" />
</form>
```

Unabhängig vom für den CSRF-Schutz ausgewählten Framework kann und muss der Entwickler die Wirksamkeit des Tokens testen. Auch hierfür ist OWASP ZAP hervorragend geeignet. Oft genügt auch ein simples Browser-Add-on, mit dem auch versteckte Formularwerte manipuliert werden können. Möglich ist das etwa mit den Firefox-Add-ons *Groundspeed* [10] oder *Firebug* [11] (Abbildung 3):

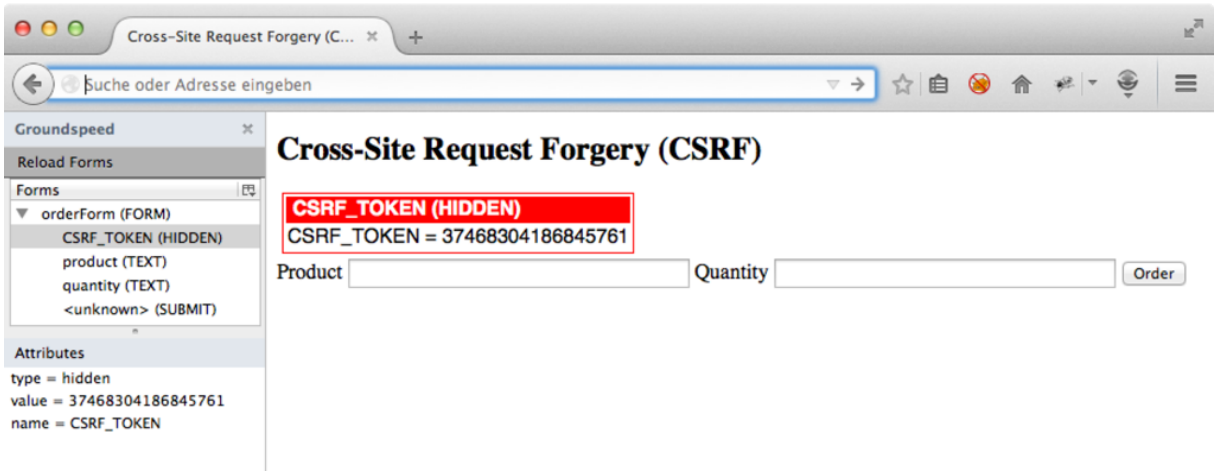

Abb. 3: Firefox Browser-Add-on Groundspeed zur Manipulation von versteckten Formularwerten

Risiko 3rd-Party-Bibliotheken

Ein vor allem im Java-Umfeld verbreitetes Sicherheitsproblem betrifft 3rd-Party-Bibliotheken. Gerade durch die zahlreiche Verwendung von fremder Funktionalität ist es wichtig, diese wie den eigenen Code regelmäßig zu aktualisieren und damit keine neuen Sicherheitsprobleme in die Webanwendung zu bringen.

Bei einer typischen Java-Webanwendung mit zahlreichen 3rd-Party-Bibliotheken ist es allerdings schwierig, stets auf dem Laufenden zu sein und verwundbare Bibliotheken zeitnah durch die neueste Version auszutauschen. Hier hilft *OWASP Dependency Check* [12] weiter, mit dessen Hilfe ein Entwickler auch lokal sämtliche Abhängigkeiten auf bekannte Sicherheitslücken überprüfen kann. Gefundene Sicherheitsprobleme werden per HTML-Report aufgezeigt und können anschließend vom Entwickler ausgetauscht werden.

Mit diesem Tool lässt sich der nächste Schritt tun: die Einbindung des Scans in den Jenkins-Build [13]. Durch die damit erfolgte Visualisierung per Diagramm und evtl. auch einem fehlschlagenden Build bemerken andere Entwickler das neue Security-Feature und fragen nach dessen Bedeutung. Die Sicherheit von Webanwendungen gerät ins Gespräch und wird idealerweise weiter verbessert.

Fazit

Viele Maßnahmen zur sicheren Entwicklung von Webanwendungen kann ein Entwickler alleine beginnen und in der eigenen Arbeit anwenden. In jedem Fall empfiehlt sich ein Blick in eine aktuelle Sicherheitsproblem-Rangliste wie den OWASP TOP 10. Einen guten Start stellt dann die Implementierung von XSS-Gegenmaßnahmen dar. Erst danach sollten weitere Sicherheitsprobleme angegangen werden. Durch die auch hierfür bereits zahlreich eingesetzten 3rd-Party-Bibliotheken lohnt es sich, einen Prozess zur regelmäßigen Überprüfung und Aktualisierung aller fremder Bibliotheken und Frameworks zu installieren.

Sicher wird eine Webanwendung allerdings erst, wenn alle Entwickler gemeinsam dieselben Entwicklungsarten anwenden und dieselben Codekonventionen beachten. Mit *OWASP ZAP* und *OWASP Dependency Check* sind dabei zwei Tools speziell für Entwickler verfügbar, die diesen Personen die sichere Entwicklung erleichtern und gleichzeitig auch die Wartung der Webanwendung unterstützen.

Ressourcen

[1] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[2] http://cwe.mitre.org/top25
[3] http://www.microsoft.com/en-us/download/details.aspx?id=42518
[4] http://beanvalidation.org/1.0/spec
[5] http://hibernate.org/validator
[6] https://www.owasp.org/index.php/OWASP_Java_Encoder_Project
[7] http://www.w3.org/TR/CSP
[8] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[9] http://projects.spring.io/spring-security
[10] https://addons.mozilla.org/de/firefox/addon/groundspeed
[11] https://addons.mozilla.org/de/firefox/addon/firebug
[12] https://www.owasp.org/index.php/OWASP_Dependency_Check
[13] http://jenkins-ci.org

Kontaktadresse:

Dominik Schadow
BridgingIT GmbH
Königstraße 42
D-70173 Stuttgart

Telefon: +(49) (0)711 7616 183 - 0
Fax: +(49) (0)711 7616 183 - 399

E-Mail dominik.schadow@bridging-it.de
Internet: www.bridging-it.de