Java Security Mythen

Dominik Schadow
BridgingIT GmbH
Stuttgart

Schlüsselworte
Java, Webapplikation, Webanwendung, Sicherheit, Security, Cross-Site Scripting, Cross-Site Request Forgery, HTTP Strict Transport Security Header

Einleitung

Java hatte lange Zeit einen geradezu fantastischen Ruf in Bezug auf die Sicherheit. Für Entwickler war es mehr oder weniger ausreichend, sich um die Geschäftslogik zu kümmern, für den Rest sorgte die Java Laufzeitumgebung. Wirklich kritische Sicherheitsprobleme traten immer nur in anderen Programmiersprachen und deren Laufzeitumgebungen auf. Auch wenn diese Aussage etwas überspitzt daherkommt, hat es doch einige Zeit gedauert, bis das Thema Sicherheit im Bewusstsein der Java-Entwickler und –Architekten, aber auch der Projektverantwortlichen, angekommen ist. Vor allem die Entwickler stehen seither vor dem Problem, Wahres von Unwahrem zu trennen und bei der Sicherheit ihrer Webanwendung nicht auf verbreitete Mythen hereinzufallen. Drei dieser Mythen werden im Folgenden vorgestellt und auf ihren Wahrheitsgehalt hin überprüft.

Sichere Entwicklung von Java-Webanwendungen

Es hat einige Zeit gedauert, bis das Thema „sichere Entwicklung“ im Umfeld von Java-Webanwendungen angekommen ist. Sicherlich haben sich schon immer einige Entwickler und andere Projektbeteiligte Gedanken über die Sicherheit ihrer Java-Webanwendung gemacht. Der breiten Masse von Entwicklern wurde dieses wichtige Thema aber erst in den letzten Jahren ins Bewusstsein gerückt. Dazu beigetragen haben neben zahlreichen spektakulären Datendiebstählen (die in der Webanwendung verwendete Programmiersprache spielte hier eigentlich keine Rolle) vor allem die überaus kritischen Sicherheitslöcher der Java Laufzeitumgebung des vergangenen Jahres. Paradoxerweise haben diese Sicherheitslöcher keinerlei direkte Auswirkungen auf die Sicherheit von Java-Webanwendungen, benötigt bei diesen browserbasierten Applikationen der Client doch überhaupt keine Java Runtime. Dennoch taten diese Skandale zumindest eine Zeit lang auch ihr Gutes und rückten das Thema Sicherheit nicht nur in den Fokus von Architekten und Entwicklern, sondern auch in die von IT-Verantwortlichen.

Die Aussage „Java ist sicher“ ist seither nicht mehr so häufig zu hören wie vor einigen Jahren. Häufig ist sogar das Gegenteil „Java sei vollständig unsicher“ zu hören. Dafür kümmern sich Entwickler jetzt aktiv(er) um die Sicherheit von Java-Webanwendungen und wollen ihre Anwendung frei von Sicherheitsproblemen ausliefern. Allerdings taucht nun mit den teils weit verbreiteten Java Security Mythen ein neues Problem auf. Worauf kann man sich als Entwickler verlassen, worauf nicht? Wo sollte man Aussagen zur Sicherheit hinterfragen und in Tests überprüfen? Gewiss sind einige dieser Mythen wahr, andere gehören allerdings ganz oder zumindest teilweise ins Reich der Legenden. Drei dieser Mythen rund um die sichere Datenübertragung sowie zu zwei der größten Probleme in Webanwendungen, Cross-Site Scripting und Cross-Site Request Forgery, sollen im Folgenden daher näher betrachtet werden.

Sichere Datenübertragung

Wohl kaum noch jemand bezweifelt die Notwendigkeit, kritische Benutzerdaten während ihrer Übertragung zu verschlüsseln. Weit verbreitetes Mittel der Wahl ist hier Secure Sockets Layer (SSL) bzw. Transport Layer Security (TLS), also die Verwendung von HTTPS. Alle Daten werden dabei über eine verschlüsselte Verbindung zwischen Client und Server ausgetauscht, u.a. Benutzername und

Passwort werden so sicher übertragen. Und auch die nach einer erfolgreichen Anmeldung generierte Session-ID (JSESSIONID in Java) wird damit vor fremden Zugriff geschützt übermittelt.
Da HTTPS allerdings je nach Benutzerzahlen durchaus spürbare Performance-Nachteile mit sich bringt, hält sich hartnäckig die Behauptung, dass die Umstellung von ungesichertem HTTP auf das sichere HTTPS nach dem Anmeldeformular genügt. Benutzername und Passwort werden damit auf einer nicht durch HTTPS gesicherten Seite eingegeben, das Formular übermittelt allerdings an eine HTTPS-URL. Sicherlich werden kritische Daten erst ab diesem Formular übertragen, eine dem Benutzer zugeordnete (nicht anonyme) Session-ID steht ebenfalls erst nach der Anmeldung zur Verfügung. Die offensichtlich schützenswerten Daten werden damit nur gesichert übertragen.
Verloren geht dabei allerdings die Integrität der Anmeldeseite bzw. des Anmeldeformulars. Ein Angreifer kann teils mit geringem Aufwand und unter Ausnutzung weiterer Sicherheitsprobleme wie Cross-Site Scripting das Anmeldeformular austauschen, die eingegebenen Benutzerdaten auf eine von ihm kontrollierte URL umleiten und dem Benutzer anschließend eine Fehlerseite wegen eines fehlerhaften Logins präsentieren. Dieser glaubt an einen Tippfehler und trägt seine Daten erneut ein. Da der Anmeldevorgang dieses Mal nicht abgefangen wird, wird der Benutzer angemeldet und kann wie gewünscht mit der Webanwendung arbeiten. Die Benutzerdaten sind damit in den Händen des Angreifers gelandet, unbemerkt vom Benutzer.
Dieser Mythos ist ein klassisches Beispiel für eine halbherzig durchgeführte sichere Entwicklung. Die Erkenntnis, dass kritische Daten übertragen und deshalb gesichert werden müssen, ist vorhanden. Alleine die Umsetzung wurde nicht konsequent durchgeführt. Bereits das Login-Formular muss immer per HTTPS gesichert ausgeliefert werden. Die Herausforderung ist, dies auch entsprechend durchzusetzen. Und hier kann ein Java-Entwickler durchaus etwas unternehmen.
Da die meisten Anwender kein http:// mehr einer URL voranstellen, wird aus Gewohnheit auch selten https:// eingegeben. Hier hilft der „HTTP Strict Transport Security Header“ (HSTS) (http://tools.ietf.org/html/rfc6797) weiter. Dieser sorgt dafür, dass die Seiten einer Webanwendung nur noch über eine per HTTPS gesicherte Verbindung ausgeliefert werden. Bei Eingabe einer http:// URL wird automatisch eine Umleitung auf https:// durchgeführt. Unsichere Zertifikate (abgelaufen oder selbst erstellt) sowie gemischte Inhalte (also per HTTP referenzierte Inhalte innerhalb einer HTTPS-Seite) werden geblockt und führen zum Abbruch der Kommunikation. Im Gegensatz zu bisher wird hier keine obskure Warnung angezeigt, die von den meisten Benutzern ohnehin ignoriert wird. Die Verwendung der Webanwendung ist bei derartigen Verstößen nicht mehr möglich.
Die einzige Aufgabe für den Java-Entwickler ist es, diesen Header in der Server-Response zu setzen:

```
HttpServletResponse response //...

response.setHeader("Strict-Transport-Security", "max-age=2592000;
includeSubDomains");
```

Der Einbezug von Subdomains ist optional. Wichtiger ist der `max-age` Parameter. Dieser funktioniert als eine Art Countdown angegeben in Sekunden. Innerhalb dessen Gültigkeit wird immer eine Umleitung auf HTTPS durchgeführt. Eine weitere Response mit diesem Parameter lässt den Countdown dabei wieder von vorne beginnen. Hier ist es immens wichtig, eine genügend große Zeitspanne zu definieren. Auch normalerweise täglich verwendete Webanwendungen bleiben über das Wochenende unbenutzt, ein Urlaub lässt eine noch größere Zeitspanne verstreichen. Bei einem zu geringen Wert findet der erste Aufruf dann wieder ohne eine sofortige Umleitung auf HTTPS statt. Ein Angreifer hat somit eine theoretische Chance, die Benutzerdaten abzugreifen. Werte in der Größenordnung von einem halben oder einem ganzen Jahr ergeben somit durchaus Sinn und haben keine negativen Nebenwirkungen.
Wie jede clientseitige Policy muss allerdings auch der HSTS-Header vom Browser unterstützt werden. Google Chrome und Mozilla Firefox gehen hier mit gutem Beispiel voran, andere Browser werden über kurz oder lang nachziehen. Da der Response-Parameter keine negativen Auswirkungen bei

fehlender Browserunterstützung hat, sollte er in jedem Fall in eigenen Webanwendungen gesetzt werden.

Cross-Site Scripting

Cross-Site Scripting (XSS) belegt seit vielen Jahren einen der Spitzenplätze der OWASP Top 10 (https://www.owasp.org). Dabei gelingt es einem Angreifer, Skript-Code (meist JavaScript) einzugeben und diesen im Browser eines anderen Besuchers zur Ausführung zu bringen. Das Skript kopiert dann beispielsweise die Session-Informationen des Opfers (seine Session-ID) und übermittelt diese Daten an den Angreifer. Dieser kann sich damit gegenüber der Webanwendung als sein Opfer ausgeben und in dessen Namen Operationen ausführen.

Input-Validierung, aber vor allem das Output-Escaping, sind hier die richtigen Gegenmaßnahmen. Wo die Input-Validierung in jeder Webanwendung sehr individuell ist und so oft für jedes Eingabefeld die gültigen Zeichen festgelegt werden müssen, kann das Output-Escaping weitgehend standardisiert werden. So muss beim Output-Escaping im wohl meistverwendeten HTML-Kontext u.a. ein < in *<* und > in *>* konvertiert werden. Für den Browser ist dies die Aufforderung, die folgenden Zeichen nicht als auszuführenden Code, sondern als gewöhnlichen Text zu betrachten. Egal welche Eingabe ein Angreifer somit eingibt, der Browser zeigt diese immer nur als normalen Text an.

Genau dieses Output-Escaping versprechen dann beispielsweise Frameworks wie die bekannten Java Server Faces (JSF) automatisch auszuführen. Cross-Site-Scripting-Angriffe haben bei Verwendung von JSF somit keine Chance, der Entwickler muss sich nur um die Eingabe-Validierung kümmern.

Wirklich? Sicherlich denken die meisten Entwickler bei der Ausgabe von Benutzereingaben im Browser an ein `outputText`-Element. Tatsächlich ist dies die am weitesten verbreitete Form. Hier führt die Anzeige mit

```
<h:outputText value="#{bean.input}" />
```

oder auch

```
#{bean.input}
```

immer ein Output-Escaping durch. Eine Eingabe wie

```
<script>alert('XSS with JSF')</script>
```

wird damit als einfacher Text, und nicht als Dialogbox angezeigt. Und zwar ganz ohne Zutun des Entwicklers. Allerdings landen Benutzereingaben gelegentlich auch in Drop-Down-Listen, gefüllt mit dem Inhalt einer Map:

```
<h:selectOneMenu>
  <f:selectItems value="#{bean.input}" />
</h:selectOneMenu>
```

Hier erscheint bei Eingabe des obigen JavaScripts eine Dialogbox, JSF führt in diesem Fall also kein automatisches Output-Escaping durch. Auch das explizite Setzen des `itemLabelEscaped="true"` Attributs hat bei der verwendeten Variante keinerlei Wirkung.

An einer Stelle macht das Framework somit einen kritischen Fehler, bei Drop-Down-Listen ist der Standardwert des Escaping-Attributs auf false gesetzt. Vermittelbar ist das nur sehr schwer, JSF hat nun einmal den Ruf, vor XSS zu schützen. Die einzig wirksame Gegenmaßnahme an dieser Stelle ist entweder die Umstellung auf einen Array als Input samt der Angabe des `itemLabelEscaped="true"` Attributs (bei dieser Variante ist das Attribut wirksam), oder die zusätzliche Verwendung einer speziellen Security-Output-Escaping-Bibliothek wie die OWASP Enterprise Security API (ESAPI)

(https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API) oder die Coverity Security Library (https://github.com/coverity/coverity-security-library).
JSF ist auch bei weitem nicht das einzige Framework, das einen sicherheitskritischen Fehler aufweist. Als Java-Entwickler muss man sich hier angewöhnen, auch für Frameworks, d.h. eigentlich „fremden" Code, innerhalb seiner Anwendung Verantwortung zu übernehmen. Das heißt einerseits, solche Funktionalität und Werbeaussagen durch Tests regelmäßig zu überprüfen (z.B. mit den eigenen JUnit-Tests), als auch bei neuen Framework- bzw. Bibliotheksversionen diese umgehend in den eigenen Anwendungen zu integrieren und zu testen. Kritische Fehler in weit verbreiteten Frameworks sprechen sich schnell herum und werden entsprechend schnell in Angriffen ausgenutzt. Bei der in der Java-Entwicklung weit verbreiteten Nutzung von Frameworks kann man es sich daher nicht leisten, keine Verantwortung für den Code in Frameworks zu übernehmen und diese blind zu verwenden.

Cross-Site Request Forgery
Auch Cross-Site Request Forgery (CSRF) ist ein sehr weit verbreitetes Sicherheitsproblem in vielen Webanwendungen. Ein Angreifer macht sich bei diesem Angriff zu Nutze, dass ein Browser mit einem bei einer Webanwendung angemeldeten Benutzer automatisch immer dessen Credentials mit zum Server überträgt. Ein Request zu einem bekannten Server schickt so automatisch alle vorhandenen Cookies mit, die Webanwendung weiß also, dass sie es mit einem bestimmten Benutzer zu tun hat. Allerdings ist das auch bei CSRF der Fall. Hier löst ein Angreifer den Request heimlich aus. Der Browser schickt die Benutzerdaten mit, die Webanwendung erkennt den vermeintlichen Benutzer wieder und führt die angeforderte Operation aus.
Hier hört man bei vielen Gelegenheiten, dass man einfach seine Formulare von GET-Requests auf POST-Requests umstellen müsste, wodurch die Webanwendung vor CSRF geschützt sei. Kann das wirklich so einfach sein? Tatsächlich lässt sich ein POST-Formular nicht mehr mit einem simplen

```
<img src="http://www.sample.de/servlet?param1=wert">
```

auslösen. Bei GET-Requests führt dieses „Image" bereits zum Erfolg und ruft das dahinterliegende Servlet auf. Allerdings lassen sich Requests ja auch mit JavaScript und dessen XMLHttpRequest auslösen. Oder auf die klassische Art mit einem durch Cascading Style Sheets (CSS) in einem unsichtbaren Layer versteckten Formular, das automatisch ausgefüllt, aber von einem anderen Formular verdeckt wird. Beim Abschicken des vermeintlichen Formulars schickt der Benutzer in Wahrheit das darunterliegende ab. Die Umstellung auf POST hält also die einfachste aller CSRF-Angriffsformen auf, nicht aber die (geringfügig) komplexeren Varianten.
Als Lösung des CSRF-Problems ist neben einer ständigen Re-Authentifizierung vor der Ausführung einer Operation nur die für den normalen Benutzer völlig transparente Verwendung eines Anti-CSRF-Tokens möglich. Dabei wird zu Session-Beginn ein für jeden Benutzer individueller Zufallswert berechnet und in der Session gespeichert. Gleichzeitig wird dieser Wert als versteckter Parameter jedem Formular hinzugefügt. Jeder beim Server eingehende Request vergleicht zunächst, ob der versteckte Wert mit dem in der Session übereinstimmt. Erst dann wird die Operation ausgeführt. Ein fehlender oder falscher Wert zeigt an, dass der Request gefälscht und dem Benutzer heimlich untergeschoben wurde. Als Folge wird die Verarbeitung unmittelbar abgebrochen. Diese auch als „Synchronizer Token Pattern" bekannte Gegenmaßnahme ist bereits in vielen Java-Frameworks automatisch aktiv. Unter anderem auch im der kommenden Spring Security Version 3.2.0 (http://spring.io/blog/2013/08/21/spring-security-3-2-0-rc1-highlights-csrf-protection). Wie bei XSS gezeigt, sollte allerdings auch dieses Vertrauen in ein Security-Framework durch eigene Tests bestätigt werden.
Im Umfeld des als falsch entlarvten CSRF-POST-Request-Mythos taucht häufig auch die Behauptung auf, dass rein im Intranet betriebene Webanwendungen keine besondere Sicherheit erfordern. Schließlich ist das interne Netzwerk nach außen hin ja mit einer Firewall geschützt. Die zusätzlichen Kosten für eine sichere Entwicklung können daher problemlos eingespart werden. Auch dieser Mythos erweist sich bei genauerer Betrachtung als falsch. Initiiert wird der Request auf die Webanwendung

vom Browser des Benutzers, der sich innerhalb der Firewall befindet. Der CSRF-Request wird daher ohne Einschränkungen ausgeführt, selbst eine im Intranet vermeintlich geschützte Webanwendung kann somit zum Ziel einer CSRF-Attacke werden. Für den Java-Entwickler bedeutet das, dass Webanwendungen unabhängig ihres Einsatzortes sicher entwickelt und entsprechend geschützt werden müssen. Eine Intranetanwendung benötigt damit den gleichen Grundschutz, wie eine deutlich exponiertere Internetanwendung.

Fazit

Auch wenn sich ein erfreulicher Sinneswandel beim Thema sichere Entwicklung von Java-Webanwendungen abzeichnet, müssen noch zahlreiche teils weit verbreitete Mythen rund um deren Sicherheit aus der Welt geschafft werden. Für den Endanwender ist es letztlich egal, ob ein Programmierfehler oder der Glaube in einen unwahren Mythos zum Verlust seiner Daten oder in seinem Namen ausgeführter Aktionen geführt haben. Er erwartet zu Recht eine sicher entwickelte Webanwendung. Für die Java-Entwickler ist es daher immens wichtig, verbreiteten (Irr-)Glauben in Frage zu stellen und sich selbst von dessen Wahrheitsgehalt zu überzeugen.

Kontaktadresse:

Dominik Schadow
BridgingIT GmbH
Königstraße 42
D-70137 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