03-2016 | Herbst | www.ijug.eu

# Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

## Java wächst weiter


**Microservices**
Schnell und einfach implementieren

**Container-Architektur**
Verteilte Java-Anwendungen mit Docker

**Java-Web-Anwendungen**
Fallstricke bei der sicheren Entwicklung

4 191978 304903 03

iJUG Verbund


Neues von den letzten Releases


Sich schnell einen Überblick über bestehende Strukturen und deren Beziehungen verschaffen




Daten in unterschiedlichen Formaten in der Datenbank ablegen

# Fallstricke bei der sicheren Entwicklung von Java-Web-Anwendungen

**Dominik Schadow, BridgingIT GmbH**

*Bei der sicheren Entwicklung von Web-Anwendungen zeichnet sich eine ähnliche Entwicklung ab wie bei deren Entwurf: eine gewisse Standardisierung und empfohlene sowie erprobte Vorgehensweisen. Wo Empfehlungen existieren, sind die zugehörigen Antipatterns und Fallstricke allerdings nicht weit. Diese bedrohen die mühsam errungene Sicherheit einer Web-Anwendung und können die umgesetzten Maßnahmen gleich wieder zunichtemachen.*

Zur sicheren Entwicklung von Java-Web-Anwendungen existieren mittlerweile zahlreiche Empfehlungen und Vorgehensweisen, etwa vom Open Web Application Security Project (OWASP) *[1]*. Man denke nur an das Output Escaping zur Vermeidung von Cross-Site Scripting (XSS) oder an Prepared Statements zur Verhinderung von SQL-Injections. Derartig konkrete Empfehlungen sind meist vergleichsweise einfach in vielen Web-Anwendungen umsetzbar. Sie sind stark standardisiert und mehr oder weniger überall gleich umzusetzen. Als Entwickler muss man sich nur der Bedrohung bewusst sein und die zugehörige Gegenmaßnahme kennen und umsetzen.

An anderer Stelle werden Empfehlungen rund um die sichere Entwicklung notgedrungen sehr viel allgemeiner. Beispiele hierfür sind die Benutzerverwaltung oder das Session Management. Zwar hat nahezu jede Web-Anwendung mit diesen Aufgaben zu tun; die konkrete Implementierung, Konfiguration und angebundene Systeme unterscheiden sich allerdings meist deutlich. Direkt einsetzbare Empfehlungen werden damit schwierig beziehungsweise zielen immer nur auf bestimmte Komponenten oder Frameworks ab.

Trotz dieser Einschränkung haben sich einige Muster herauskristallisiert, die man vor, während und nach der Entwicklung berücksichtigen sollte. Nur ist es – wie auch bei den normalen Design-Patterns – allzu leicht möglich, einmal falsch abzubiegen und aus einem eigentlich sicheren Pattern ein unsicheres zu machen.

Erschwerend kommt im Umfeld der sicheren Software-Entwicklung das subjektive Sicherheitsgefühl hinzu. Als Entwickler merkt man ja durchaus, ob eine Web-Anwendung sich sicher anfühlt oder nicht. Wendet man nun ein Security-Pattern bei der Entwicklung an, verstärkt sich das Gefühl, dass man alles Notwendige zur Absicherung der Web-Anwendung getan hat. Warnzeichen – etwa eine bekanntgewordene Sicherheitslücke in einer anderen Web-Anwendung – werden in der Folge leichter ignoriert; man hat die Web-Anwendung ja sicher entwickelt. Der Schein trügt allerdings, sie ist längst nicht so sicher wie gedacht und eigentlich möglich.

Der Artikel stellt einige ausgewählte Fallen und Antipatterns vor, die häufig die Sicherheit einer Web-Anwendung bedrohen. Diese Liste ist natürlich bei Weitem nicht vollständig. Zum einfacheren Verständnis lassen sich die folgenden Antipatterns grob in die Projektphasen „Architektur", „Implementierung" und „Wartung" einteilen. Da für Entwickler der Schwerpunkt auf der Implementierung einer Web-Anwendung liegt, liegt dort auch der Schwerpunkt dieses Artikels. Ganz ohne Architektur und Wartung geht es aber auch bei der Sicherheit nicht.

## Architektur

Security-Antipatterns lassen sich in den meisten Projekten schon in der ersten Phase finden. Was sich beim Design einer Web-Anwendung längst durchgesetzt hat – etwa UML-Diagramme oder Dokumentation mit „arc42" – steckt bei der Planung der Sicherheit noch immer in den Kinderschuhen.

Tatsächlich wird die Sicherheit einer Web-Anwendung allzu häufig bis kurz vor dem Release aufgeschoben; Sicherheit verlangsamt schließlich die Entwicklung und macht alles unnötig kompliziert. Von einer inhärenten Sicherheit der Anwendung kann daher keine Rede sein. Stattdessen wirkt die Sicherheit wie ein zusätzlicher Layer, der die Anwendung möglichst vollständig umgeben und absichern soll. Ganz im Sinne einer Firewall, die einen Schutzkreis um das Netzwerk (in diesem Fall die Anwendung) zieht. Security Flaws, also tiefliegende Sicherheitsprobleme aufgrund mangelnder Planung, sind damit nahezu vorprogrammiert. Die Gefahr ist groß, dass man bei dieser späten Absicherung einen Eingangskanal wie etwa einen konsumierten fremden Webservice übersieht und Daten ungeprüft in die eigene Anwendung gelangen.

Dabei ist die Vermeidung dieses Antipattern so einfach: Die Sicherheit einer Web-Anwendung muss wie deren Design vor der Implementierung geplant werden. Die UML hilft hier nur sehr begrenzt weiter, notwendig sind Threat Models *[2]*. Diese sind im Gegensatz zur UML wenig standardisiert; als Architekt oder Entwickler trifft man daher in jedem

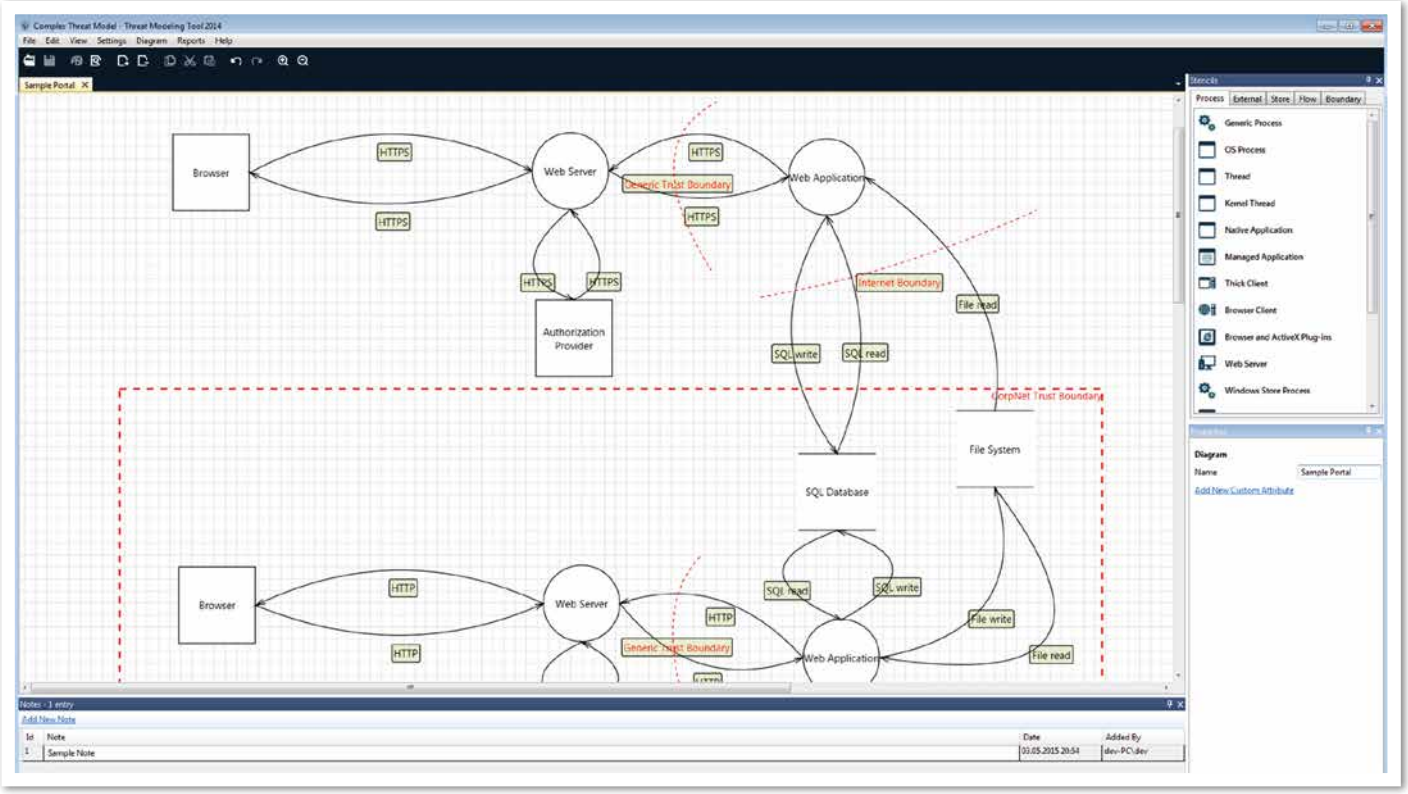


*Abbildung 1: Microsoft Threat Modeling Tool 2014*

Tool auf andere grafische Elemente. Auch wenn ohnehin nur wenige produktionsreife Tools zur Verfügung stehen, sollte man sich daher auf ein Tool beschränken und diesem möglichst immer treu bleiben. Das selbst für Java-Web-Anwendungen empfehlenswerte Tool ist das Microsoft Threat Modeling Tool 2014 *[3] (siehe Abbildung 1)*.

Dieses Tool steht im Rahmen des Microsoft Secure Development Lifecycle kostenlos zur Verfügung und ist – zumindest bei der Modellierung – unabhängig von der eingesetzten Programmiersprache. Bei der Erstellung des Modells folgt man idealerweise den Daten, in einer Web-Anwendung ist der Startpunkt daher häufig der Webbrowser (ebenso wie der Endpunkt). Die vom Benutzer eingegebenen Daten wandern durch das System, kommunizieren/interagieren mit anderen Entitäten oder werden mit weiteren Informationen angereichert. Alle diese Entitäten – Datenbanken, Webserver, Benutzerverwaltung etc. – werden in das Threat Model eingetragen.

Im nächsten Schritt entstehen Trust Boundaries, im Beispiel die rot gestrichelten Linien. An diesen Stellen müssen etwa aus Benutzereingaben vertrauenswürdige Daten werden, damit diese sicher weiterverarbeitet werden können. Die Eingabedaten müssen also validiert werden. Eine weitere Trust Boundary stellt einen Kommunikationsvorgang dar, an dem Daten aus dem Internet in das Intranet übertragen werden. Das Threat Model kann dadurch bei großen Web-Anwendungen mit zahlreichen externen Entitäten durchaus komplex werden und sollte daher in mehrere Threat Models aufgeteilt werden.

Gerade an den Boundaries wird deutlich, dass zwingend alle externen Entitäten im Model erfasst werden müssen. Ein Angrei-

```
byte[] hash(char[] password, byte[] salt) throws Exception {
  SecretKeyFactory skf =
     SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
  PBEKeySpec spec = new PBEKeySpec(password, salt, 10000, 512);

  return skf.generateSecret(spec).getEncoded();
}
```

*Listing 1*

```
@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder(10);
}
```

*Listing 2*

```
public void doFilter(ServletRequest req, ServletResponse res,

  FilterChain filterChain) throws Exception {
  response.addHeader("Strict-Transport-Security",
     "max-age=31536000; includeSubDomains");
  filterChain.doFilter(req, response);
}
```

*Listing 3*

fer steht andernfalls nur vor der Aufgabe, die eine vergessene (und damit ungeschützte) Entität zu finden und für einen Angriff auszunutzen.

Das vollständige Threat Model legt die Basis für die nun folgende Implementierung. Viele konkrete Bedrohungen der eigenen Web-Anwendung sind dadurch bekannt, beginnend bei XSS und SQL-Injections über die Eingabevalidierung bis hin zu applikationsspezifischen Bedrohungen. Gleich wie bei UML-Diagrammen hat der Architekt in der Regel auch hier die Aufgabe, das Model zu pflegen und bei Änderungen der Anwendung zu aktualisieren.

## Implementierung

Bei der Implementierung ist es aufgrund der Vielzahl unterschiedlicher Anwendungen, Frameworks und Systeme schwierig, verbreitete Antipatterns und deren korrektes Pattern allgemeingültig aufzuzeigen. Die folgenden Abschnitte konzentrieren sich daher auf Aufgaben rund um das Login eines Benutzers bis hin zur Erstellung seiner Session. Der vollständige Quellcode steht auf GitHub *[4]*.

```
@WebServlet
public class LoginServlet extends HttpServlet {
  protected void doPost(HttpServletRequest req,
     HttpServletResponse res) throws ServletException {
     request.changeSessionId();
  }
}
```

*Listing 4*

```
<reporting>
  <plugins>
     <plugin>
        <groupId>org.owasp</groupId>
        <artifactId>dependency-check-maven</artifactId>
        <version>1.3.3</version>
        <reportSets>
           <reportSet>
              <reports>
                 <report>aggregate</report>
              </reports>
           </reportSet>
        </reportSets>
     </plugin>
  </plugins>
</reporting>
```

*Listing 3*

## Passwörter speichern

Trotz Single Sign-on (SSO) besitzen viele Web-Anwendungen ihre eigene Benutzerverwaltung. Wo eine Umstellung auf ein zentrales SSO-System nicht möglich ist, haben diese Anwendungen die Aufgabe, Benutzer-Passwörter sicher zu speichern. Die Zeiten, in denen diese im Klartext gespeichert wurden, sind zum Glück längst vorbei. Auch dass die Passwort-Verschlüsselung ein Antipattern ist, hat sich längst herumgesprochen: Ein verschlüsseltes Passwort kann schließlich wieder entschlüsselt werden, und das ist niemals notwendig. Das Hashen von Passwörtern vor dem Speichern ist daher die einzig richtige Vorgehensweise.

Nun gilt es, unter den vielen verfügbaren Hashing-Algorithmen den richtigen auszuwählen. MD5, SHA1 und die gesamte SHA2-Familie sind hierfür nicht die richtige Wahl (und wurden teilweise auch schon längst als unsicher entlarvt). Diese auf Geschwindigkeit optimierten Hashing-Algorithmen machen es einem Angreifer zu leicht, viele Millionen Passwörter pro Sekunde per Brute-Force-Angriff auszuprobieren. Stattdessen gilt es – wohl nahezu einmalig in der Software-Entwicklung –, den Hashvorgang gezielt zu verlangsamen und so einen Angriff auszubremsen. Für einen einzelnen Benutzer ist eine Verzögerung um einige Millisekunden nicht spürbar, für einen Angreifer dagegen schon. Schließlich treffen ihn die beispielsweise 100 Millisekunden Verzögerung nicht nur einmalig, sondern gleich millionenfach.

Zum sicheren Hashen von Passwörtern müssen daher die Algorithmen Password-Based Key Derivation Function 2 (PDKDF2), „bcrypt" oder „scrypt" verwendet werden. PBKDF2 steht ohne weitere Bibliotheken direkt in plain Java zur Verfügung und lässt sich sehr einfach einsetzen *(siehe Listing 1)*.

„10000" bestimmt dabei die Anzahl der Iterationen, „512" die Hash-Länge. Gerade bei der Anzahl der Iterationen tappt man gerne in die nächste Falle. Dieser Wert muss sich mit Verbesserung der Hardware immer weiter erhöhen, damit Angriffe weiterhin wirksam aufgehalten werden können. Wichtig ist bei einer Erhöhung der Anzahl von Iterationen, dass sämtliche Benutzer-Passwörter innerhalb einer vorgegebenen Zeit aktualisiert werden.

Sobald sich ein Benutzer erfolgreich eingeloggt hat, muss sein Passwort mit einem neuen Salt mit der neuen Anzahl von Iterationen gehasht und anschließend als neuer Hash gespeichert werden. Wer sich innerhalb dieser Zeitspanne nicht anmeldet, sollte deaktiviert werden und muss seinen Benutzer später etwa per „Passwort vergessen" erneut aktivieren. Das endlose Speichern von potenziell unsicher gehashten Passwörtern birgt ansonsten ein enormes Risiko, sollte ein Angreifer doch einmal Zugang zu den Benutzerdaten erlangen.

Vereinfachen lässt sich das sichere Speichern von Benutzer-Passwörtern mit Spring Security *[5]*. Spring Security unterstützt out of the box die Verwendung von „bcrypt" und muss dazu lediglich um eine Bean in einer mit „@Configuration" annotierten Konfigurationsklasse erweitert werden *(siehe Listing 2)*.

Die „10" als Konstruktor-Argument steht dabei nicht für die Anzahl von Iterationen, sondern bestimmt die Komplexität der Berechnung. „10" ist der Default, Werte zwischen „4" und „31" sind möglich. Die Komplexität steigt dabei exponentiell an.

Eine weitere Alternative ist „scrypt", das im Gegensatz zu den beiden zuvor genannten Algorithmen nicht Iterationen, sondern einen Mindest-Speicherverbrauch als Schutz vor Brute-Force-Angriffen verwendet.

## Sichere Datenübertragung

Häufig verwenden Web-Anwendungen das ungesicherte HTTP zur Datenübertragung, solange keine Benutzerdaten (Benutzername und Passwort beziehungsweise Session ID) im Spiel sind. Selbst das Login-Formular wird noch per HTTP ausgeliefert, überträgt seine Daten aber sicher an eine HTTPS-URL. Die sichere Datenübertragung ist damit gewährleistet, allerdings verschenkt man damit die Integrität des Login-Formulars. Per „Man in the middle"-Angriff kann dann so das Formularziel manipuliert worden sein

und die Daten damit zu einem vom Angreifer kontrollierten Server übertragen werden.

Erschweren lässt sich dieser Angriff durch den konsequenten Einsatz von HTTPS. Um den Browser zu dessen Verwendung zu zwingen, sollten HTTPS-Seiten immer mit dem HTTP Strict Transport Security Header (HSTS) ausgeliefert werden. Mit Spring Security wird auch diese Anforderung automatisch für per HTTPS ausgelieferte Seiten umgesetzt *[6]*, ohne Spring Security fügt der folgende ServletFilter diesen Header zur Response hinzu *(siehe Listing 3)*.

Von nun an wird die gesamte Seite nach einer einzigen initialen Auslieferung per HTTPS nur noch über das sichere Protokoll ausgeliefert. Jeder moderne Browser sorgt dann dafür, dass selbst URL-Eingaben ohne Protokoll sofort auf HTTPS umgeleitet werden. Das Abfangen einer Session-ID oder anderer Informationen ist damit nicht mehr möglich.

## Die Session-ID nach dem Login ändern

Unter Entwicklern hält sich hartnäckig das Gerücht, dass ein Benutzer erst dann eine Session-ID erhält, nachdem er sich erfolgreich bei der Anwendung angemeldet hat. Richtig ist dagegen, dass ein Benutzer sie nahezu immer sofort beim Aufruf einer Seite bekommt. Bis zum erfolgreichen Login ist eine dem anonymen Benutzer zugewiesene Session-ID für einen Angreifer allerdings wertlos. Nach dem Login ist sie jedoch mit einem Benutzer verknüpft und damit wertvoll geworden. Ein Angreifer mit Kenntnis dieser ID kann sich gegenüber der Anwendung nun als dieser Benutzer ausgeben.

Mittels Session Fixation und Session Hijacking stehen einem Angreifer gleich zwei Angriffsmöglichkeiten zur Verfügung. Bei der Session Fixation schiebt ein Angreifer einem Benutzer eine ihm bekannte Session-ID unter und wartet, bis dieser sich angemeldet hat. Beim Session Hijacking stiehlt der Angreifer die Session-ID, beispielsweise bei einer ungesicherten Übertragung per HTTP.

Zur Vermeidung dieses Problems muss die Session-ID nach einem erfolgreichen Login immer geändert werden. Spring Security erledigt auch diese Aufgabe automatisch, ganz ohne notwendige weitere Konfiguration. Ohne Spring Security hilft ein Servlet weiter, das unmittelbar nach erfolgtem Login die Session-ID ändert *(siehe Listing 4)*.

## Wartung

In wohl kaum einer Programmiersprache ist der Einsatz von fremden Bibliotheken so verbreitet wie in Java. Mehr und mehr mitunter auch kritische Funktionalität wird in Bibliotheken ausgelagert und nicht mehr selbst entwickelt. Auf der einen Seite ist das natürlich überaus begrüßenswert, Standard-Entwicklungsaufgaben werden schließlich nur von den wenigsten Entwicklern gerne übernommen. Und sicherheitsrelevante Funktionalität überlässt man besser einer zig-fach erprobten Bibliothek, als diese selbst zu implementieren.

Auf der anderen Seite steigen die Anforderungen an die eingesetzten Bibliotheken, schließlich übernehmen diese kritische Aufgaben, die man sonst eigentlich selbst entwickelt hätte. Damit wird es wichtig, alle in einer Anwendung vorhandenen Bibliotheken aktuell zu halten und verwundbare Ver-

sionen zeitnah durch verbesserte Versionen auszutauschen.

Bei der üblich großen Menge an eingesetzten Bibliotheken ist das allerdings gar nicht so einfach. Mit OWASP Dependency Check *[7]* steht zur Unterstützung ein sehr empfehlenswertes Tool zur Verfügung, das Entwickler über verwundbare Bibliotheken in der Web-Anwendung informiert. Hierfür vergleicht Dependency Check die identifizierten Bibliotheken mit Einträgen in der National Vulnerability Database (NVD) und erstellt einen entsprechenden Report. Neben dem einfachen Aufruf im Terminal und der Integration in den Maven Build *(siehe Listing 5)* steht eine Integration in Jenkins zur Verfügung.

In der Standard-Konfiguration prüft Dependency Check nach jedem erfolgreichen Build. Damit das nicht zu lange dauert, empfiehlt es sich, das NVD-Update in diesen Jobs durch Aktivieren von „Disable NVD auto-update" auszuschalten *(siehe Abbildung 2)*.

Zusätzlich richtet man einen weiteren Job ein, der regelmäßig nur ein Update der lokalen NVD durchführt. Alle anderen Jobs verwenden diese gemeinsame Datenbank und werden dadurch spürbar schneller ausgeführt. Unabhängig von der gewählten Variante (Terminal, Maven, Jenkins) generiert Dependency Check einen HTML-Report mit den identifizierten möglichen Verwundbarkeiten *(siehe Abbildung 3)*.

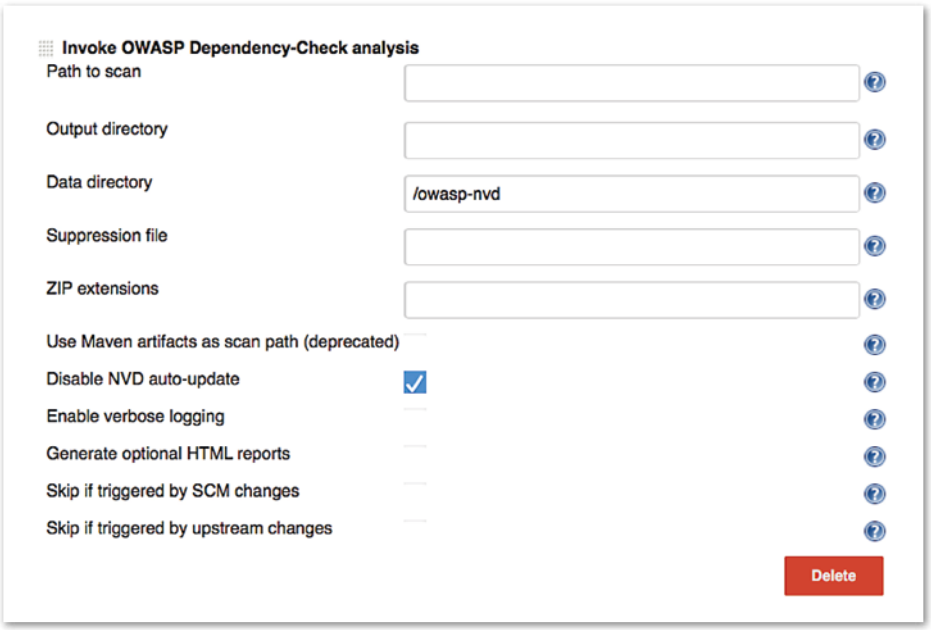


*Abbildung 2: OWASP Dependency Check in Jenkins*

Dieser Report enthält häufig „false positives", etwa wenn eine Bibliothek aufgrund ihres Namens falsch erkannt wurde. Beim Durcharbeiten der Liste gilt es daher, die Meldungen zu hinterfragen – bevor man sich an das Austauschen der Bibliothek macht. Die sich an den Austausch einer Bibliothek anschließenden Unit- und Integrationstests übernimmt Dependency Check zwar nicht,

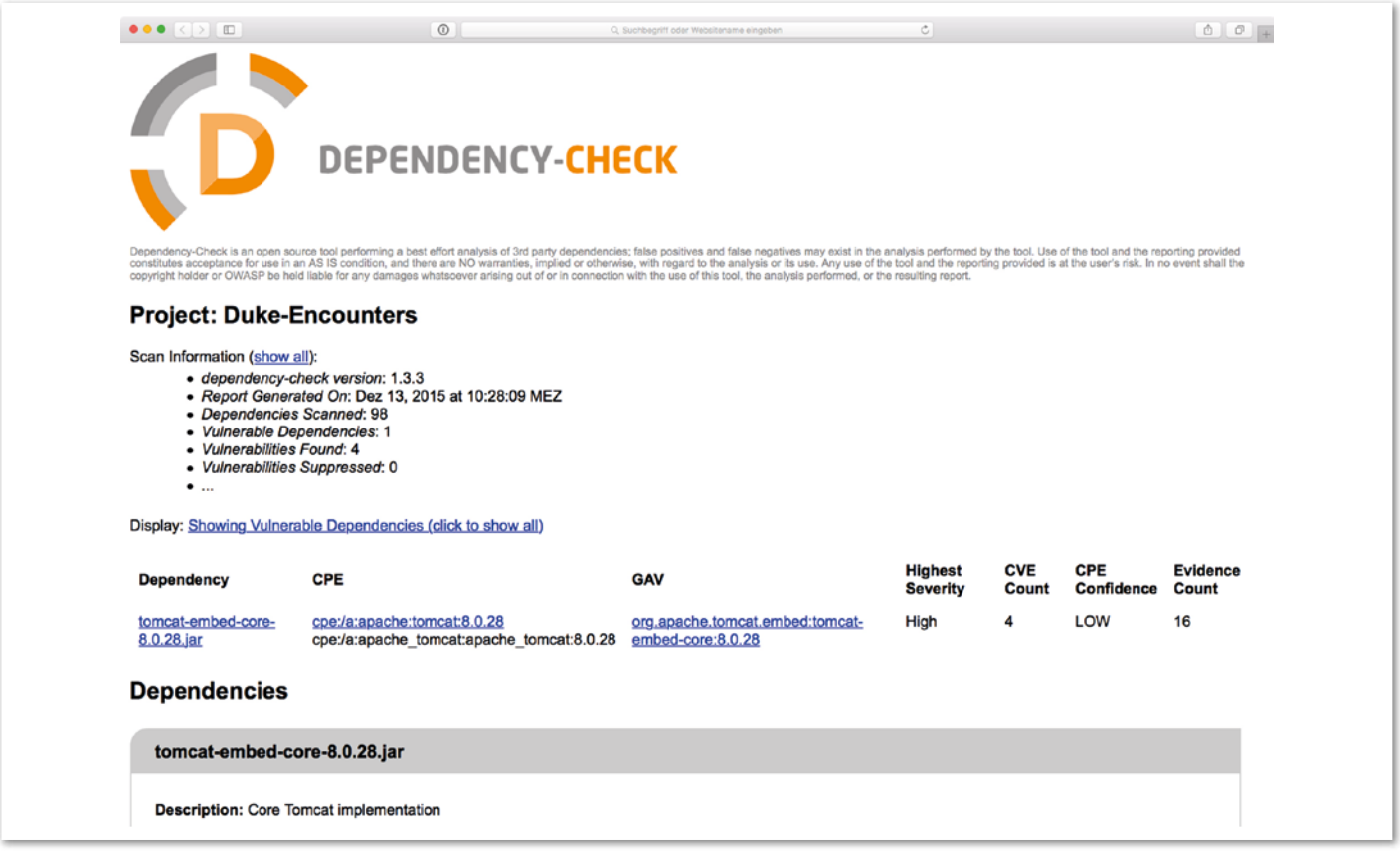


*Abbildung 3: OWASP Dependency Check Report*

unterstützt Entwickler aber immerhin bei der Identifikation von unsicheren Bibliotheken.

## Fazit

Trotz der immer weiteren Standardisierung bei der Entwicklung von sicheren Web-Anwendungen und immer mehr empfohlenen Vorgehensweisen ist und bleibt die Sicherheit einer Web-Anwendung ein komplexes Thema. Die Empfehlungen, vor Beginn der Implementierung ein Threat Model zu erstellen und währenddessen beziehungsweise danach die Aktualität der Bibliotheken zu überwachen und gegebenenfalls auszutauschen, lassen sich ohne große Anpassungen in jedem Projekt einsetzen.

Schwieriger wird es bei konkreten Implementierungsaufgaben. Hier sind die Anwendungen einfach zu unterschiedlich, empfohlene Vorgehensweisen (und damit auch typische Antipatterns) hängen stark von den angebundenen Systemen und eingesetzten Frameworks ab. Viele der typischen Fallstricke und Antipatterns lassen sich allerdings mit den richtigen Frameworks automatisch vermeiden. Wer diese nicht einsetzen kann oder will, muss bei der Entwicklung entsprechend stärker auf die gezeigten (und die anderen) Fallen achten.

## Ressourcen


[1] *https://www.owasp.org*
[2] Adam Shostack – Threat Modeling: Designing for Security – ISBN 978-1118809990
[3] *http://blogs.microsoft.com/cybertrust/2014/04/15/introducing-microsoft-threat-modeling-tool-2014/*
[4] *https://github.com/dschadow/JavaSecurity*
[5] *http://projects.spring.io/spring-security*
[6] *http://docs.spring.io/spring-security/site/docs/current/reference/html/headers.html*
[7] *https://www.owasp.org/index.php/OWASP_Dependency_Check*



*Dominik Schadow*
*dominik.schadow@bridging-it.de*


Dominik Schadow arbeitet als Senior Consultant beim IT-Beratungsunternehmen bridgingIT. Er hat mehr als zehn Jahre Erfahrung in der Java-Entwicklung und -Beratung. Sein Fokus liegt auf der Architektur und Entwicklung von Java-Enterprise-Applikationen und der sicheren Software-Entwicklung mit Java. Er ist regelmäßiger Speaker auf verschiedenen Konferenzen, Buchautor und Autor zahlreicher Fachartikel.

# *Java ist auch eine Insel – Einführung, Ausbildung, Praxis*

***gelesen von Daniel Grycman***


Mittlerweile ist die zwölfte aktualisierte Auflage des deutschsprachigen Standardwerks von Christian Ullenboom im Rheinwerk Verlag erschienen. Wie der Autor im Vorwort schreibt, sind hauptsächlich Fehlerkorrekturen vorgenommen worden. Als einzige Neuerung ist das bisherige zwölfte Kapitel aufgespalten. Im neuen dreizehnten Kapitel geht Ullenboom auf die kommende Modularisierung in Java 9 ein.

Das Buch besticht durch sein ausführliches Inhaltsverzeichnis, das die Inhalte in 23 Kapitel unterteilt. Im Vorwort bietet Christian Ullenboom eine mögliche Lernstrategie zum Lesen und Lernen an. Er bezieht sich hierbei auf die „PQ4R"-Methode. Diese soll einem Anfänger im Bereich der Java-Programmierung den Einstieg und den Wissenserwerb erleichtern.

In den ersten 13 Kapiteln wird dem Leser Java als Programmiersprache nähergebracht. Sie stellen zugleich auch den ersten Teil des Buches dar. Im zweiten Teil beschäftigt sich der Autor mit grundlegenden APIs. Allen Kapiteln ist gemein, dass an deren Ende ein Abschnitt mit dem Titel „Zum Weiterlesen" steht. Dieser bietet dem Leser die Möglichkeit, sich mit einem bestimmten Thema weiter auseinanderzusetzen. Den Abschluss des Werkes bilden eine ausführliche Übersicht über die in Java 8 enthaltenen Pakete sowie ein Index.

Die neuste Auflage der „Insel" stellt eine absolute Kaufempfehlung für Java-Neulinge dar. Das Buch präsentiert auf eine verständliche Art und Weise die notwendigen Techniken zur Java-Programmierung und gibt auch praxisorientierte Informationen an den Leser weiter.

| | |
|---|---|
| **Titel:** | Java ist auch eine Insel - Einführung, Ausbildung, Praxis |
| **Autor:** | Christian Ullenboom |
| **Auflage:** | 12. Auflage, 2016 |
| **Verlag:** | Rheinwerk Verlag |
| **Umfang:** | 1.312 Seiten |
| **Preis:** | 49,90 Euro |
| **ISBN:** | 978-3-8362-4119-9 |