Einbruchssichere **Webanwendungen**

Anwendungssicherheit wird häufig vor allem mit den Themen Authentisierung, Autorisierung und Verschlüsselung in Verbindung gebracht. Dabei lassen sich jedoch heutige Angriffe auf Webanwendungen zumeist eher auf Fehler in deren Datenvalidierung oder Session Management zurückzuführen.

von Matthias Rohr

Mit HDIV und Stinger sind nun zwei kostenlose Security Plugins verfügbar, mit denen sich sowohl neue als auch bereits vorhandene Webanwendungen nachrüsten lassen. Und dies, ohne dabei eine einzige Zeile Code zu schreiben.

Die zunehmende Bedeutung des Internets hat in den letzten Jahren zu einer rasanten technologischen Entwicklung geführt. Gleichzeitig bieten neue Technologien aber potenziell immer auch neue Angriffsmöglichkeiten. Das Open Web Application Project (OWASP) listet auf seiner Webseite [1] mittlerweile über 500 bekannte Schwachstellen in Webanwendungen auf.

Wie aber lassen sich die bestehenden und natürlich am besten gleich auch zukünftige Schwachstellen nun von Softwareentwicklern praktisch noch beherrschen? Die Lösung ist erstaunlich einfach, denn die Ursache für die meisten Schwachstellen lässt sich auf Fehler bei der Datenvalidierung oder im Session Management zurückführen. Cross Site Scripting (XSS), Session Riding bzw. Cross Site Request Forgery (CSRF), SQL Injection und Co. stellen für eine Anwendung mit entsprechend restriktiver Datenvalidierung und gehärtetem Session Management dann kaum mehr eine Gefahr da. Sucht man hingegen nach Informationen zur Implementierung von Sicherheit in Java-EE-Webanwendungen, so wird man meist nur auf solche stoßen, die sich mit Authentisierung, Autorisierung und Verschlüsselung befassen. Das liegt wohl auch daran, dass es für diese Themen mit JCE, JAAS, Acegi und durch den Servlet Container selbst sehr ausgereifte Lösungen gibt. Für Datenvalidierung und Session Management war man hier hingegen meist auf die Implementierung von Best Practices [2] angewiesen, was Fehler an dieser sensiblen Stelle sicherlich begünstigt. Seit Kurzem hat sich dies allerdings geändert, denn mit Stinger und HDIV gibt es nun zwei kostenfrei erhältliche Lösungen, mit denen sich genau diese Schwachstellen deklarativ im Servlet Container adressieren lassen. Neben Neuentwicklungen lassen sich dadurch auch bereits vorhandene Webanwendungen nachrüsten.

OWASP Stinger

Eingabevalidierung stellt die wahrscheinlich universelle Maßnahme zum Schutz einer Webanwendung dar. Da letztlich alle Angriffe über die eine oder andere Form von Eingabe, sei es über den URL, Formularparameter oder Werte aus dem HTTP Header (z.B. Cookies), erfolgen, lässt sich an dieser Stelle zumindest theoretisch auch jede Form von Angriff abwehren. Mit Stinger ist es nun möglich, genau diese Funktionalität direkt im Webcontainer abzubilden. Stinger wird hierzu einfach als Servlet Filter direkt vor die zu schützende Anwendung (bzw. Ressource) gehängt und kann so auch jeden eingehenden HTTP Request überprüfen. Eine Webapplikation rudimentär mit Stinger zu schützen, ist relativ schnell getan. Nach Herunterladen des aktuellen Pakets [3] wird das Stinger-JAR (*Stinger-2-5.jar*) im Verzeichnis *WEB-INF/lib* der Anwendung abgelegt und anschließend der Filter im entsprechende Deployment Descriptor eingetragen (Listing 1).

Wichtig ist hierbei zu beachten, dass der Container die vorhandenen Filter der Reihe nach aufruft, das Filter-Mapping von Stinger also in den meisten Fällen vor allen übrigen stehen sollte. Zusätzlich lässt sich Stinger auch auf eine bestimmte Anwendung eingrenzen. Was jetzt noch fehlt, sind die Regeln, die in einer zentralen Konfigurationsdatei (hier: *WEB-INF/stinger.xml*) definiert werden, die beim Start des Application Servers geladen wird. Setzt man die Option *reload* auf *true*, ist das Laden auch nach jeder Änderung möglich, was aber nur für Testumgebungen zu empfehlen ist.

Quellcode auf CD

Listing 1

```
</web-app>
 ...
 <filter>
  <filter-name>StingerFilter</filter-name>
  <filter-class>org.owasp.stinger.StingerFilter</filter-class>
  <init-param>
   <param-name>config</param-name>
   <param-value>stinger.xml</param-value>
  </init-param>
  <init-param><!-- Abfangen von Exceptions -->
   <param-name>error-page</param-name>
   <param-value>/error.jsp</param-value>
  </init-param>
  <init-param><!-- autom. Nachladen der Konfiguraton -->
   <param-nme>reload</param-name>
   <param-value>false</param-value>
  </init-param>
 </filter>
 <filter-mapping>
  <filter-name>StingerFilter</filter-name>
  <url-pattern>.*</url-pattern>
 </filter-mapping>
</web-app>
}
```

Listing 2

```
<regex>
 <name>emailpattern</name>
 <pattern>
   ^[\w-]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}$
 </pattern>
 <description></description>
</regex>
<ruleset>
 <path>/MyApp/edit.jsp</path>
 <name>editform</name>
 <rule>
  <name>emailcheck</name>
  <regex>emailpattern</regex>
  <missing>
   <severity>ignore</severity>
  </missing>
  <malformed>
   <severity>fatal</severity>
   <action class="org.owasp.stinger.actions.Forward">
    <parameter>
     <name>page</name>
     <value>/edit_error.jsp</value>
    </parameter>
   </action>
  </malformed>
 </rule>
  <!-- weitere Regeln für edit.jsp -->
</ruleset>
```

Ein nahe liegendes Einsatzszenario für Stinger ist die Whitelist-Validierung von Eingabewerten. Listing 2 zeigt, wie dies zur Prüfung von optionalen E-Mail-Adressen in Formulareingaben eingesetzt wurde. Für jeden zu testenden Parameter würde hierzu eine Regel (*rule*) definiert, die als Regelsatz (*ruleset*) auf einen spezifischen URL angewendet wird.

Jede Regel ermöglicht die Unterscheidung von zwei Fällen: Ob ein bestimmter Parameter überhaupt im HTTP Request enthalten ist, wird über einen <*missing*>-Block geprüft. Da der Parameter *email* hierbei optional sein soll, wird die entsprechende Prüfung mit *severity=ignore* ausgesetzt. Entscheidend ist jedoch die Prüfung, ob ein Parameter im korrekten Format vorliegt. Dies wird innerhalb des <*malformed*>-Blocks und auf Basis eines RegEx-Patterns [4] ausgedrückt, das in diesem Beispiel alle nicht alphanumerischen Zeichen erfasst. Hierzu wurde ein Pattern für die Validierung von E-Mail-Adressen definiert. Unter [5] finden Interessierte weitere Ausdrücke, die sich auch zum Parsen von Kreditkartennummern und sogar Passwortstärken nutzen lassen. Trifft dieses Pattern zu, wird die definierte Action ausgeführt, über die hier eine Weiterleitung auf eine dialogspezifische Fehlerseite erfolgt.

Eine Anwendung, die alle ihre Parameter entsprechend restriktiv prüft, wird kaum mehr Probleme mit XSS oder Ähnlichem haben. Ob eine solche Form der Eingabevalidierung aber überhaupt noch beherrschbar ist, wenn sie auf eine gesamte Anwendung angewendet wird, bleibt dagegen zu bezweifeln. Anwendungen, die auf Basis eines MVC-Frameworks wie Struts oder Spring MVC realisiert wurden, sind daher besser beraten, Whitelisting direkt über die integrierten Validierungsfunktionen abzubilden. Natürlich besitzen aber nun einmal nicht alle Anwendungen eine entsprechende Architektur. Und selbst wenn, lassen sich entsprechende Lücken in der Konfiguration kaum ausschließen. Im Folgenden soll daher ein anders Anwendungsszenario vorgestellt werden: Sicherlich am meisten Probleme bereiten Sonderzeichen in den Eingabedaten. Im Fall von XSS sind dies vor allem die spitzen Klammern (< und >) bei SQL-Injection dagegen neben anderem die Anführungsstriche (" und '). Ein zentralisierter Ansatz könnte daher lauten, sicherzustellen, dass sämtliche Parameter (POST und GET) ausschließlich aus alphanumerischen Zeichen bestehen dürfen. Die entsprechende Regel sähe dann so aus:

```
<ruleset>
 <name>DEFAULT</name>
 <path>STINGER_DEFAULT</path>

 <rule>
  <name>STINGER_ALL</name>
  <regex>^[a-zA-Z0-9\.]*$</regex>
  <missing>
   <severity>ignore</severity>
  </missing>
  <malformed>
   <severity>continue</severity>
    <action class="org.owasp.stinger.actions.
                                        Encode" />
  </malformed>
 </rule>
</ruleset>
```

Was hierbei zunächst auffällt, ist die Verwendung der äußerst nützlichen Schlüsselwörter STINGER_ALL und STINGER_DEFAULT. Mit dem ersten lässt sich dabei eine Regel gleich auf alle Parameter anwenden, mit dem zweiten wird der Regelsatz als Default definiert und dadurch auf alle Anfragen angewendet. Ein weiterer Unterschied ist die Action, mit der in diesem Fall sämtliche nicht-numerischen Zeichen als HTML Entity kodiert werden. Eine spitze Klammer (<) wird dadurch automatisch in die entsprechende HTML-Darstellung umgesetzt (&*lt;*) und damit unschädlich gemacht. Alternativ ließe sich an dieser Stelle auch das entsprechende Sessionobjekt gleich invalidieren. Derart „harte" Actions sind aber nur dann zu empfehlen, wenn wirklich sichergestellt werden kann, dass tatsächlich ein Angriff auf die Anwendung durchgeführt wurde.

Bei einer Validierung auf Zeichenebene ist dies aber sicherlich nicht der Fall. Nur weil ein Benutzer ein Sonderzeichen eintippt, soll dieses wahrscheinlich nicht automatisch aus der Anwendung ausgeloggt werden. Außerdem können Sonderzeichen mitunter auch in zahlreichen Stellen (Passwörter, Kom-

mentare, Suchanfragen) erlaubte Eingaben sein. Ein Ansatz besteht nun darin, den oben verwendeten Ausdruck einfach um die betreffenden Zeichen zu erweitern. Eine solche Aufweichung würde dann jedoch schnell dazu führen, dass auch immer mehr Angriffe nicht mehr abgefangen werden und der gewünschte Schutz gegen Null geht. Sinnvoller ist es, Ausnahmen für die betreffenden Dialoge zu spezifizieren, wodurch der Gebrauch von Sonderzeichen nur an diesen Stellen möglich ist. Da dies bei großen Anwendungen aber schnell zu dem beschriebenen Effekt der schwierigen Beherrschbarkeit führt, lässt sich die Sache natürlich auch umdrehen und die nicht erlaubten Zeichen explizit angeben.

Dieses dritte Anwendungsszenario von Stinger ist zwar weniger restriktiv, dafür aber auch deutlich praktikabler. Eine solche Blacklist, die sehr schlicht nur die Zeichen (< > ' ") beinhaltet, würde bereits vor den meisten XSS- und SQL-Injection-Angriffen Schutz bieten. Um diese als Pattern zu formulieren, müssen die betreffenden Zeichen Hex-kodiert werden (\x3C entspricht <), da die XML-Konfiguration sonst nicht mehr valide wäre:

```
^[^\x3C\x3E\x22\x27].*$
```

Doch auch bereits mit einer solch einfachen Blacklist lassen sich Fehlfunktionen nicht ausschließen. Angenommen eine Webanwendung verfügt über ein Formular, über das Kommentare mit beliebigen Sonderzeichen eingegeben werden können. Dann sollen diese Zeichen wahrscheinlich nicht unbedingt HTML-kodiert in der Datenbank abgelegt werden. Kann aber sichergestellt werden, dass ein bestimmter Parameter vor der Ausgabe sauber enkodiert wird, ist dies auch gar nicht unbedingt erforderlich. Bei der Nutzung von bestimmten Taglibs (z.B. JSTL oder Struts) für die Ausgabe erfolgt dies automatisch. In diesem Fall kann der betreffende Parameter von der Prüfung ausgenommen werden:

```
<ruleset>
 <path>/MyApp/edit.jsp</path>
 <name>COMMENT</name>
 <rule>
  <name>comment</name>
  <missing>
   <severity>ignore</severity>
  </missing>
  <malformed>
   <severity>ignore</severity>
  </malformed>
 </rule>
</ruleset>
```

Listing 3

```
<regex>
 <name>malformed</name>
 <!-- Erkennen der Zeichen <>'"()\/ -->
 <pattern>^[^\x3C\x3E\x26\x22\x28\x29\x27\x3F\x3B\x2B].*$</ pattern>
</regex>
<ruleset>
 <path>STINGER__ALL</path><!-- Default-Regel -->
 <name>STINGER_DEFAULT</name>
 <rule>
  <name>STINGER_ALL</name>
  <regex>malformed</regex>
  <missing>
   <severity>ignore</severity>
  </missing>
  <malformed>
   <severity>continue</severity>
   <action class="org.owasp.stinger.actions.Encode" />
    <action class="org.owasp.stinger.actions.Log">
   <parameter>
    <name>log</name>
    <value>stinger.log</value>
   </parameter>
   <parameter>
   <name>level</name>
    <value>info</value>
   </parameter>
   <parameter>
    <name>message</name>
    <value>
  parameter %name with value %encoded_value from %ip has been encoded
    </value>
   </parameter>
  </action>
  </malformed>
 </rule>
</ruleset>
}
```

Die Möglichkeit, Zeichen mittels Kodierung anders darzustellen, lässt sich allerdings auch von einem Angreifer dazu nutzen, eine solche Blacklist zu umgehen. Um dies zu verhindern, lässt sich natürlich auch die Blacklist um entsprechende Kodierungszeichen erweitern (z.B. \ oder &). Listing 3 enthält hierzu ein vollständiges Beispiel, in dem auch weitere problematische Zeichen berücksichtigt wurden.

Anders als beim Whitelisting, sollten sich auch bei größeren Anwendungen die erforderlichen Ausnahmen immer noch in Grenzen halten. Besonders hilfreich ist hierbei eine Logging Action, mit der es möglich ist, zunächst die Auswirkung einer neuen Regel auf die Anwendung zu testen und darauf entsprechende Ausnahmen zu definieren. Aber auch eine Form von Intrusion Detection System lässt sich hiermit einrichten.

Da Actions schlicht von *AbstractAction* abgeleitete Java-Klassen darstellen, lassen sich diese auch sehr einfach selbst schreiben. Auf diese Weise ließen sich etwa spezifische Kodierungen durchzuführen oder bestimmte Werte im Sessionobjekt ablegen.

Bleibt noch zu erwähnen, dass Stinger auch die Prüfung von Cookie-Werten unterstützt, denn auch diese können von einem Angreifer manipuliert werden. Grundsätzlich macht eine solche Prüfung für Session-Cookies allerdings wenig Sinn, da diese üblicherweise bereits vom Application Server übernommen wird.

Listing 4

```
<web-app>
 ...
 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
   /WEB-INF/applicationContext.xml,/WEB-INF/hdiv-config.xml
  </param-value>
 </context-param>

 <filter>
  <filter-name>ValidatorFilter</filter-name>
  <filter-class>org.hdiv.filter.ValidatorFilter</filter-class>
 </filter>

 <filter-mapping>
  <filter-name>ValidatorFilter</filter-name>
  <url-pattern>*.do</url-pattern>
 </filter-mapping>

 <listener>
  <listener-class>
   org.springframework.web.context.ContextLoaderListener
  </listener-class>
 </listener>

 <listener>
  <listener-class>org.hdiv.listener.InitListener</listener-class>
 </listener>
  <!-- Taglib-Mapping: hier für Struts -->
 <taglib>
  <!--<taglib-uri>http://struts.apache.org/tags-html</taglib-uri>-->
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/hdiv-html.tld</taglib-location>
 </taglib>
 <taglib>
  <taglib-uri>/WEB-INF/struts-nested.tld</taglib-uri>
  <taglib-location>/WEB-INF/hdiv-nested.tld</taglib-location>
 </taglib>
 <taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/hdiv-logic.tld</taglib-location>
 </taglib>
</web-app>
```

HDIV

Arbeitete Stinger noch als reiner Eingabevalidator, so lässt sich HDIV schon als vollwertiges Security Framework bezeichnen, über das sich gleich zahlreiche Sicherheitsfunktionen implementieren lassen.

Der Grundgedanke bei HDIV besteht darin, jeden URL-Zugriff auf eine Anwendung von einem spezifischen State abhängig zu machen. Anders als Stinger unterscheidet HDIV dafür zwischen editierbaren und nicht editierbaren Parametern. Zu letzteren zählen etwa Hidden Fields oder Select Boxes, die von einem Benutzer in der Regel nicht verändert werden sollten. HDIV merkt sich nun eben diese Parameter mitsamt Werten innerhalb eines State-Objekts, welches es wiederum im Sessionobjekt des Benutzers ablegt. Schließlich wird der betreffende URL um einen zusätzlichen Parameter erweitert, der auf das jeweilige State-Objekt referenziert.

Abbildung 1 zeigt das Beispiel einer entsprechend angepassten Seite, wie sie schließlich der Client erhält. Der Wert *_HDIV_TOKEN_* repräsentiert hier die Referenz auf den im Speicher abgelegten State des jeweiligen URL. Wem der Name für diesen Token nicht zusagt, der kann ihn in der Konfiguration auch beliebig anders wählen. Durch diesen Mechanismus sind Manipulationen von nicht editierbaren Parametern wie Hidden Fields nun nicht mehr möglich. Selbst wenn eine Anwendung bereits zahlreiche Eingaben validiert, wird die Behandlung genau solcher Parameter jedoch zur Freude von Hackern oft vergessen. Darüber hinaus ist es nun für einen Benutzer nicht mehr möglich, einen bestimmten URL (z.B. /admin) direkt aufzurufen, der nicht zuvor als Entry Point definiert wur-

Installation von HDIV

Die Installation von HDIV ist nicht besonders kompliziert, obwohl es den einen oder anderen Fallstrick zu beachten gibt. Das HDIV-Paket, das sich auf der OWASP-Seite [3], herunterladen lässt, enthält hierzu alle benötigten Dateien. Unabhängig von dem verwendeten Framework, sollten die darin im Pfad *bin/commons* befindlichen Bibliotheken in das Verzeichnis *WEB-INF/lib* der jeweiligen Installation kopiert werden. Abhängig davon, ob Struts, Spring MVC oder JSTL eingesetzt wird, müssen diese dann um die entsprechende Bibliothek (z.B. *hdiv-spring-mvc-2.0.jar*) ergänzt werden. Ebenfalls installationsabhängig sind die Konfigurationsdateien. Es empfiehlt sich daher, einfach den Inhalt des entsprechenden Unterordners (z.B. *spring-mvc/web*) direkt im betreffenden *WEB-INF*-Verzeichnis abzulegen.

Schließlich müssen nur noch die entsprechenden Einstellungen im Deployment Diskriptor vorgenommen werden (siehe Listing 4). Wichtig ist hierbei eigentlich nur das Taglib-Mapping. Hier sollten exakt die URIs stehen, die auch in den entsprechenden JSP-Dateien verwendet werden. Sowohl relative Pfade (z.B. */WEB-INF/struts-html.tld*) als auch absolute Pfade (z.B. *http://struts.apache.org/tags-html*) lassen sich hier abbilden.

de. Hierdurch lässt sich zum einen die Durchführung von Angriffen wie Session Riding (CSRF) verhindern, zum anderen aber gleich auch Fehler in der Autorisierung beheben. Durch das beschriebene State-Prinzip lassen sich natürlich auch keine URL-Parameter mehr von Angreifer manipulieren, um auf andere Datenbankeinträge zuzugreifen. Häufig referenziert die Anwendung nämlich über einen ID-Parameter direkt auf eine entsprechende Objekt-ID in der Datenbank. Um eine Anwendung entsprechend zu schützen, ist noch nicht einmal eine besondere Konfiguration erforderlich (siehe Kasten). HDIV setzt dabei eine Ebene tiefer als Stinger an und erweitert die vorhandenen Taglibs um die benötigte Funktion zum Setzen des States. Unterstützt werden derzeit Struts (Version 1 und 2), Spring MVC (ab Version 2) sowie Jakarta Taglibs.

HDIV wurde auf Basis des Spring Frameworks entwickelt, was sich in der Praxis jedoch nur insofern auswirkt, als dass die Konfiguration eben auch Spring-spezifisch erfolgt. Einige wichtige Einstellungsoptionen sind in Listing 5 dargestellt. Als Erstes sollte hier mittels der Option *errorPage* der URL die Fehlerseite eingetragen werden, die bei Aufrufen mit einem ungültigen oder fehlenden State angezeigt werden soll.

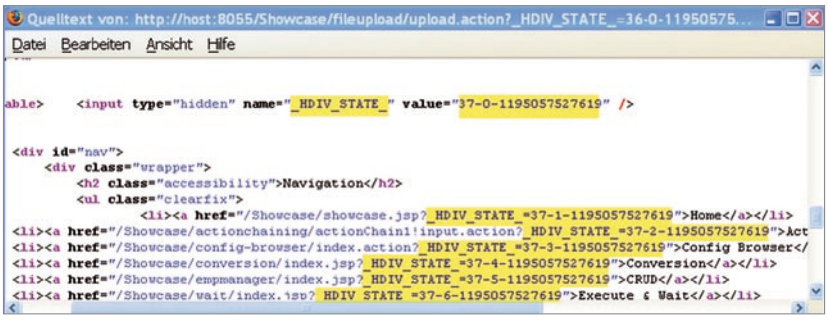

Abb. 1: Mit HDIV geschützte Seite

Da ein State natürlich nicht bei dem Aufruf der Startseite mit gesendet werden soll, muss diese hier unter *userStartPages* als Entry Point eingetragen werden. Auch URLs, die direkt oder über JavaScript aufgerufen werden, sind hierüber explizit oder mithilfe einer RegEx auszunehmen. Die Manipulation von Cookies wird von HDIV ebenfalls nicht durch den Anwender erlaubt, was in den meisten Fällen auch überaus sinnvoll ist. Im Bedarfsfall lässt sich dieses Verhalten aber über die Option *avoidCookiesIntegrity* abschalten.

Ähnlich Stinger bietet auch HDIV die Möglichkeit zur Validierung von Eingabedaten. Aufgrund des State-Prinzips brauchen dabei jedoch sinnvoller Weise nur editierbare Felder berücksichtigt zu werden. Die Regeln werden auch hier wieder etwas umständlich über Spring-Bean-Konfigurationen vorgenommen. Diese Funktion lässt sich jedoch auch problemlos von Stinger übernehmen, das in dieser Hinsicht auch mehr Möglichkeiten bietet. Um dabei nur die editierbaren Felder zu validieren, muss das Filter-Mapping von HDIV lediglich vor dem von Stinger gesetzt werden. HDIV besitzt darüber hinaus noch weitere, interessante Einstellungsmöglichkeiten. Ein Blick in die ausführliche Dokumentation [2] oder die beiliegenden Beispieldateien lohnt sich daher.

Gegenüberstellung mit anderen Ansätzen

Auch bisher gab es natürlich mit Web Application Firewalls (WAFs) durchaus adäquate Lösungen, mit denen sich die Datenvalidierung und das Session Management einer Webanwendung peripher schützen ließen. Neben dem Preis haben diese Systeme jedoch auch verschiedene andere Nachteile: Was ist etwa, wenn gar kein Zugriff auf die Infrastruk-

tur besteht oder eine Anwendung unerwartet auf ein anderes System umgezogen wird? Daneben besteht natürlich immer auch die Gefahr, dass solche Systeme den Durchsatz des teuer angeschafften Application Servers ausbremsen. Genau diese Probleme bestehen nicht mehr, wenn die WAF-Funktionalität gleich in die Applikation (bzw. WAR-Datei) implementiert wird. Entwickler haben so auch die Möglichkeit, die Anwendung bereits in der Entwicklungsphase zusätzlich abzusichern, egal auf welchem System diese später deployed wird.

Soll das nun aber heißen, das WAFs hier ihre Daseinsberechtigung verloren haben? Nein, natürlich nicht. Gerade kommerzielle Produkte bieten zahlreiche Funktionalitäten wie URL Encryption, Heuristiken oder Anti-Automatisierungs-Techniken, mit denen derzeit weder Stinger noch HDIV dienen können [6]. Zukünftig ist allerdings sicherlich davon auszugehen, dass WAF-Anbieter zunehmend die deutlichen Vorteile von Container-integrierten Lösungen erkennen und auch hier verstärkt entsprechende Lösungen anbieten werden.

Fazit

Mit dem hier vorgestellten Ansatz, WAF-Funktionalität innerhalb der Anwendung zu konfigurieren, lassen sich neue wie vorhandene Webanwendungen wirkungsvoll gegen eine Vielzahl heutiger Hackerangriffe schützen. Das Ganze ist dabei völlig kostenlos und lässt sich noch dazu sauber im Application Server Cluster skalieren. Je restriktiver man jedoch die Regeln anzieht, desto größer ist auch die Wahrscheinlichkeit, dass die ein oder andere gewünschte Funktion der Anwendung nicht mehr richtig funktioniert. Ein sinnvoller Ansatz ist daher, Stinger zunächst im reinen Logging-Modus zu betreiben, die Anwendung intensiv zu testen und für eventuelle Treffer Ausnahmen zu definieren oder einzelne Regeln entsprechend anzupassen. Da sowohl Stinger als auch HDIV vollständig durch den Deployment Descriptor eingebunden werden, braucht man diesen lediglich auszutauschen, um deren Funktionalität vollständig ein- bzw. auszuschalten. Dies lässt sich auch bequem über ein entsprechendes Target in der Build-Datei steuern. Dazu sei noch bemerkt, dass beide hier vorgestellten Lösungen noch relativ neu sind, ein ausgiebiger Anwendungstest hinsichtlich etwaiger Nebeneffekte daher in jedem Fall anzuraten ist.

So interessant beide Ansätze auch sind, soll dies nicht bedeuten, dass zusätzliche Sicherheitsmaßnahmen damit überflüssig wären. So führt die Validierung von Benutzereingaben in den seltensten Fällen zu einer wirklichen Korrektur einer Schwachstelle, sondern verhindert stattdessen lediglich deren Ausnutzbarkeit. Da Regeln immer auch Löcher enthalten können, sollten neue Anwendungen niemals ausschließlich auf diesen Schutz vertrauen, sondern stattdessen zusätzlich eine konsequente Validierung aller Ausgabedaten durchführen. Im Falle von HTML-Ausgaben besteht dies vor allem in der HTML-Kodierung, im Falle von SQL-Abfragen der Einsatz von Prepared Statements.

Der konsistente Einsatz von MVC-Frameworks wie Struts oder Spring MVC stellt eine sinnvolle Ergänzung zu den hier vorgestellten Verfahren dar, da beide eine konsistente Kodierung von Ausgabedaten durchführen und darüber hinaus äußerst mächtige Funktionen zur Whitelist-Validierung bieten. Letztere lassen sich aufgrund des Umfangs dagegen eher umständlich mit einem zentralen Filter wie Stinger implementieren und pflegen. ■

Listing 5

```
<beans>
 ...
 <bean id="config" class="org.hdiv.config.HDIVConfig">
  <!-- you can use regular expressions -->
  <property name="userStartPages">
   <list>
    <value>/</value>
    <value>/index.jsp</value>
    <value>/Welcome.*</value>
   </list>
    </property>

    <property name="errorPage">
     <value>/error.jsp</value>
    </property>
              </bean>

<!-- Parameterwerte umbenenen -->
<bean id="confidentiality" class="java.lang.Boolean">
 <constructor-arg>
  <value>true</value>
 </constructor-arg>
</bean>

<!-- Optionales Validieren von Eingabedaten -->
<bean id="editableParemetersValidations"
 class="org.hdiv.config.HDIVValidations">
 <property name="urls">
  <map><!-- zu validierende URLS --></map>
 </property>
</bean>

<!-- Cookies schützen -->
<bean id="avoidCookiesIntegrity" class="java.lang.Boolean">
 <constructor-arg>
  <value>false</value>
 </constructor-arg>
</bean>

<!-- Cookie-Werte umbenennen -->
<bean id="avoidCookiesConfidentiality" class="java.lang.Boolean">
 <constructor-arg>
  <value>false</value>
 </constructor-arg>
</bean>
</beans>
```

Matthias Rohr ist Mitarbeiter bei der SecureNet GmbH in München, wo er sich als Experte für Web Application Security insbesondere mit Sicherheitsaspekten innerhalb des Java Enterprise Bereichs beschäftigt.

Links & Literatur

[1] www.owasp.org/index.php/Category:Vulnerability
[2] www.bsi.de/literat/studien/websec/index.htm
[3] www.owasp.org/index.php/Category:OWASP_Stinger_Project
[4] java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html
[5] www.owasp.org/index.php/OWASP_Validation_Regex_Repository
[6] www.securenet.de/papers
[7] www.hdiv.org