Defending Web Services using Mod Security (Apache)

Methodology and Filtering Techniques

Abstract

Web services are vulnerable to several attacks. These attacks can lead to information leakage and further aid in remote command execution. By using WSDL an attacker can determine an access point and available interfaces for web services. These interfaces or methods take inputs using SOAP over HTTP/HTTPS. If these inputs are not defended well at the source code level, they can be compromised and exploited. ModSecurity operates as an Apache Web server module, ideal for defending web services against attacks that also include malicious *POST variable* content. This paper describes techniques to defend your web services layer using mod_security.

Pre-requisites

Familiarity with the Apache Web server, the basics of web services (WSDL, SOAP and HTTP) and working knowledge of *mod_security* is required in order to understand the content included in this paper.

Shreeraj Shah
Co-Author: "Web Hacking: Attacks and Defense" (Addison Wesley, 2002) and published several advisories on security flaws.
shreeraj@net-square.com
http://www.net-square.com

TABLE OF CONTENTS

Acknowledgement

Lyra Fernandes for her help on documentation.

Introduction

Deployment Scenario:

Web services are deployed in many different ways. For example, a web service may be running on Tomcat/Axis and plugged into Apache Web server. The simplest of web services is created using java code (.jws extension)

Tomcat integrated with Apache:

The Apache configuration file httpd.conf must be modified to allow support for Tomcat requests.

Apache httpd.conf

```
LoadModule jk2_module modules/mod_jk2.so
JkSet config.file /usr/local/apache2/conf/workers2.properties
…
```

Axis web.xml:

All jws files will be processed by AxisServlet and provide a web services deployment framework.

```
<servlet-mapping>
   <servlet-name>AxisServlet</servlet-name>
   <url-pattern>*.jws</url-pattern>
 </servlet-mapping>
```

Sending *HEAD* request to server:

```
C:\>nc 192.168.7.50 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 03 Jan 2005 18:58:32 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Content-Location: index.html.en
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Fri, 04 May 2001 00:01:18 GMT
ETag: "1c4d0-5b0-40446f80;1c4e6-961-8562af00"
Accept-Ranges: bytes
Content-Length: 1456
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-Language: en
Expires: Mon, 03 Jan 2005 18:58:32 GMT
```

The above request suggests that the web server is Apache/2.0.50 (Unix) and supported by the Tomcat module mod_jk2/2.0.4.

Loading mod security

To provide a defense at the web services level, we have loaded the mod_security module that provides application filtering capabilities. Installation of mod_security and other relevant information can be found on http://www.modsecurity.org.

The following line in httpd.conf loads the security module.

Apache httpd.conf

```
LoadModule security_module    modules/mod_security.so
…
```

Next, we need to modify httpd.conf to apply filtering rules. The screenshot below shows the section where the directives need to be applied.

Applying directives: An example

A section of the configuration file httpd.conf where directives for mod_security are applied.

```
<IfModule mod_security.c>

    # Turn the filtering engine On or Off
    SecFilterEngine On
    SecFilterDefaultAction "deny,log,status:500"
    SecFilterScanPOST On

    # Make sure that URL encoding is valid
    SecFilterCheckURLEncoding On
    SecFilterCheckCookieFormat On

    # Unicode encoding check
    SecFilterCheckUnicodeEncoding Off

    # Only allow bytes from this range
    SecFilterForceByteRange 1 255

    # Only log suspicious requests
    SecAuditEngine RelevantOnly

    # The name of the audit log file
    SecAuditLog logs/audit_log

    SecFilterSelective REQUEST_METHOD "!^GET$" chain
    SecFilterSelective HTTP_Content-Type "!(^$|^application/x-www-form-urlencoded$|^multipart/form-data)"
```

```
    SecFilterSelective REQUEST_METHOD "^POST$" chain
    SecFilterSelective HTTP_Content-Length "^$"

    SecFilterSelective HTTP_Transfer-Encoding "!^$"

</IfModule>
```

We now have the entire defense plan in place. We have Apache running with Tomcat/Axis on which any web services can be deployed. At the same time we have ModSecurity loaded on the server. Restart Apache Web server and make sure everything is in place.

Sample web service

For our example, we shall consider a sample web service at the following URL:
http://192.168.7.50/axis/modsec.jws

We can obtain a WSDL response for this specific web service from following URL
http://192.168.7.50/axis/modsec.jws?wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://192.168.7.50/axis/modsec.jws"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://192.168.7.50/axis/modsec.jws" xmlns:intf="http://192.168.7.50/axis/modsec.jws"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:message name="getInputResponse">
  <wsdl:part name="getInputReturn" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="getInputRequest">
  <wsdl:part name="id" type="xsd:string"/>
 </wsdl:message>
 <wsdl:portType name="modsec">
  <wsdl:operation name="getInput" parameterOrder="id">
   <wsdl:input message="impl:getInputRequest" name="getInputRequest"/>
   <wsdl:output message="impl:getInputResponse" name="getInputResponse"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="modsecSoapBinding" type="impl:modsec">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getInput">
   <wsdlsoap:operation soapAction=""/>
   <wsdl:input name="getInputRequest">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://DefaultNamespace" use="encoded"/>
   </wsdl:input>
   <wsdl:output name="getInputResponse">
    <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://192.168.7.50/axis/modsec.jws" use="encoded"/>
   </wsdl:output>
```

```
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="modsecService">
  <wsdl:port binding="impl:modsecSoapBinding" name="modsec">
   <wsdlsoap:address location="http://192.168.7.50/axis/modsec.jws"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Method/Interface information derived from WSDL

We scrutinize the WSDL response obtained as a result of the http request passed in the previous step. Of specific interest here is the invocation method for this web service.

```
<wsdl:operation name="getInput">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getInputRequest">
     <wsdlsoap:body encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
            namespace="http://DefaultNamespace" use="encoded"/>
    </wsdl:input>
    <wsdl:output name="getInputResponse">
     <wsdlsoap:body encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
            namespace="http://192.168.7.50/axis/modsec.jws" use="encoded"/>
</wsdl:output>
```

getInput is the invocation method. The parameter is also disclosed – "id" which accepts and sends a string as output. This is a very simple echo web service.

```
 <wsdl:message name="getInputResponse">
  <wsdl:part name="getInputReturn" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="getInputRequest">
  <wsdl:part name="id" type="xsd:string"/>
 </wsdl:message>
```

Invoking the web service over HTTP

Request sent to 192.168.7.50 over port 80,

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 573
Expect: 100-continue
```

```
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://192.168.7.50/axis/modsec.jws"
xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:string">hi</id></q1:getInput>
</soap:Body></soap:Envelope>
```

Response from server,

```
HTTP/1.1 200 OK
Date: Mon, 03 Jan 2005 19:24:10 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Set-Cookie: JSESSIONID=69C6540CC427A8B064C0795ADDFC20EA; Path=/axis
Content-Type: text/xml;charset=utf-8
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
   <ns1:getInputResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:ns1="http://DefaultNamespace">
       <ns1:getInputReturn xsi:type="xsd:string">hi</ns1:getInputReturn>
   </ns1:getInputResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Gluing web services resource into mod_security

We have our web service resource loaded at the following URL.
http://192.168.7.50/axis/modsec.jws
It's generally a good idea to create a rule set for this resource. To do so we have to glue our resource into httpd.conf in following way.

```
<IfModule mod_security.c>
     SecFilterEngine On
     SecFilterDefaultAction "deny,log,status:500"
```

```
    SecFilterScanPOST On
    SecFilterCheckURLEncoding On
    SecFilterCheckCookieFormat On

    SecFilterCheckUnicodeEncoding Off
    SecFilterForceByteRange 1 255

    SecAuditEngine RelevantOnly
    SecAuditLog logs/audit_log

    SecFilterSelective REQUEST_METHOD "!^GET$" chain
    SecFilterSelective HTTP_Content-Type "!(^$|^application/x-www-form-urlencoded$|^multipart/form-data)"
    SecFilterSelective REQUEST_METHOD "^POST$" chain
    SecFilterSelective HTTP_Content-Length "^$"
    SecFilterSelective HTTP_Transfer-Encoding "!^$"
# ------- Rules for web services --------------------------
    <Location /axis/modsec.jws>
        SecFilterInheritance Off
        SecFilterDefaultAction "deny,log,status:500"
        SecFilterScanPOST On
        SecFilterCheckURLEncoding On
        SecFilterCheckUnicodeEncoding On
    </Location>
#-------------------------------------------------------------
</IfModule>
```

The following directive block will apply filtering criteria for /axis/modsec.jws. These will add the required placeholders for the defense of the web service and are specified in the <Location> block.

```
# ------- Rules for web services --------------------------
    <Location /axis/modsec.jws>
        SecFilterInheritance Off
        SecFilterDefaultAction "deny,log,status:500"
        SecFilterScanPOST On
        SecFilterCheckURLEncoding On
        SecFilterCheckUnicodeEncoding On
    </Location>
#-------------------------------------------------------------
</IfModule>
```

Explained below are two of the directives.

SecFilterInheritance Off

This line will turn off all other rules and provide a clean space for a new rule set, specific to this location only.

Web services invocation methods go over POST. Hence, we are enabling POST filtering using the line below:
SecFilterScanPOST On

Defending web services inputs

The following input will be passed to the web service.

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 573
Expect: 100-continue
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://192.168.7.50/axis/modsec.jws"
xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:string">hi</id></q1:getInput>
</soap:Body></soap:Envelope>
```

In the above block **<id xsi:type="xsd:string">hi</id>** is the key XML block in which value can be passed. Objective is to define value passed in parameter called “id” and reject suspicious values in it.

Trapping *id* using regex:

ModSsecurity provides ways to trap a value passed through a POST request. One of the ways to trap the request made for “id” using filters, is shown below:

```
<Location /axis/modsec.jws>
        SecFilterInheritance Off

        SecFilterDefaultAction "deny,log,status:500"
        SecFilterScanPOST On
        SecFilterCheckURLEncoding On
         SecFilterCheckUnicodeEncoding On

      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain
```

```
</Location>
```

In above directive block, the following line will trap the request made to “id”
SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain

“*POST_PAYLOAD*” will intercept the POST data block and look for regular expression pattern ("<\s*id[^>]*>"). This regex pattern will capture the data block with “*id*” call and proceed with the rest of the checks since a “chain” directive is also applied.

Locking variable length:

One of the major security issues is passing a large buffer to a variable. This is so because this large buffer may cause the application to misbehave and/or crash, somewhere down the line during execution. The following rule will defend variable “id” against precisely this type of attack

```
<Location /axis/modsec.jws>
      SecFilterInheritance Off

      SecFilterDefaultAction "deny,log,status:500"
      SecFilterScanPOST On
      SecFilterCheckURLEncoding On
       SecFilterCheckUnicodeEncoding On

      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.{6,}</\s*id\s*>" "deny,status:500"
</Location>
```

In above directive, the regular expression pattern **"<\s*id[^>]*>.{6,}</\s*id\s*>"** will restrict the buffer and only allow a buffer of 5 characters.
In order to ascertain the effect of the above directive block, let us send across two responses, one that matches the buffer length set and the other exceeding the buffer length set in the directive block shown above.

Request: hello

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 576
Expect: 100-continue
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://192.168.7.50/axis/modsec.jws"
  xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:string">hello</id></q1:getInput>
   </soap:Body>
</soap:Envelope>
```

Response

```
HTTP/1.1 200 OK
Date: Mon, 03 Jan 2005 22:00:21 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Set-Cookie: JSESSIONID=DD892724132D7525362A2543F2C29B4C; Path=/axis
Content-Type: text/xml;charset=utf-8
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:getInputResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:ns1="http://DefaultNamespace">
    <ns1:getInputReturn xsi:type="xsd:string">hello</ns1:getInputReturn>
  </ns1:getInputResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

In the above case, a buffer of 5 characters was able to pass and we got a response ***hello***, sent in our request block, echoed back.

Now let's try sending across a request *hello1* (six characters), instead of hello and assess the response:

Request: hello1

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 577
Expect: 100-continue
```

```
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns="http://192.168.7.50/axis/modsec.jws"
    xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
       <q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:string">hello1</id></q1:getInput>
</soap:Body></soap:Envelope>
```

Response

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 03 Jan 2005 22:00:33 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Content-Length: 657
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error or misconfiguration and was unable to complete your request.</p>
<p>Please contact the server administrator,  you@example.com and inform them of the time the error occurred,
and anything you might have done that may have caused the error.</p>
<p>More information about this error may be available in the server error log.</p>
<hr />
<address>Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4 Server  at 192.168.7.50 Port
80</address>
</body></html>
```

This request is rejected by mod_security module and status 500 was sent back. This indicates that the request never hit the web services level, thereby providing a sound defense against any kind of buffer overflow, the most common and often ignored vulnerability.

Filtering meta characters:

Another major threat to variables are meta characters like **%**, single quote (**‘**) or double quotes (**“**) etc. These characters can cause SQL injection attacks to occur and may also cause unnecessary information leakage.

Adopting the following strategy will provide a sound defense against such attacks.

```
<Location /axis/modsec.jws>
        SecFilterInheritance Off

        SecFilterDefaultAction "deny,log,status:500"
        SecFilterScanPOST On
        SecFilterCheckURLEncoding On
      SecFilterCheckUnicodeEncoding On

      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.{6,}</\s*id\s*>" "deny,status:500"
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.+[\"%][^<]*</\s*id\s*>" "deny,status:500"
  </Location>
```

The regular expression pattern "<\s*id[^>]*>.+[\"%][^<]*</\s*id\s*>" denies serving an HTTP response if a variable's POST PAYLOAD consists of any of the following characters: **‘** or **“** or **%**.

Request: hi'hi

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 576
Expect: 100-continue
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://192.168.7.50/axis/modsec.jws"
 xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:string">hi'hi</id></q1:getInput>
 </soap:Body>
</soap:Envelope>
```

Response: 500 Internal Server Error

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 03 Jan 2005 22:00:33 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
```

```
Content-Length: 657
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error or misconfiguration and was unable to complete your request.</p>
<p>Please contact the server administrator,  you@example.com and inform them of the time the error occurred,
and anything you might have done that may have caused the error.</p>
<p>More information about this error may be available in the server error log.</p>
<hr />
<address>Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4 Server  at 192.168.7.50 Port
80</address>
</body></html>
```

Similarly, it is also possible to lock down variable types using appropriate regex patterns containing \d or \w.

For example SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>\d{6,}</\s*id\s*>" "deny,status:500" can be used to restrict inputs to integer values only.

Blocking SQL injection attacks:

It is also possible to inject SQL statements in variables. Concatenating together many SQL statements is possible. If input is not sanitized, additional SQL statements can be appended to existing SQL queries, often with disastrous consequences.

Block SQL injection attacks by adding the following directives:

```
<Location /axis/modsec.jws>
      SecFilterInheritance Off
      SecFilterDefaultAction "deny,log,status:500"
      SecFilterScanPOST On
      SecFilterCheckURLEncoding On
      SecFilterCheckUnicodeEncoding On

      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain
      #SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.{6,}</\s*id\s*>" "deny,status:500"
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.+['\"%][^<]*</\s*id\s*>" "deny,status:500"
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.*select.+from[^<]*</\s*id\s*>" "deny,status:500"
   </Location>


    SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.*select.+from[^<]*</\s*id\s*>" "deny,status:500"
```

The above line will look for the "*select * from . . .*" clause and if detected, will send back *500*. Similarly, SQL statements for delete, update etc can also be added.

Blocking fault codes from leaking out:

One of the major sources of information in web services is faultcode. A forced error on the server can create faultcode.

Request: a

```
POST /axis/modsec.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 569
Expect: 100-continue
Host: 192.168.7.50

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tns="http://192.168.7.50/axis/modsec.jws"
   xmlns:types="http://192.168.7.50/axis/modsec.jws/encodedTypes"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <q1:getInput xmlns:q1="http://DefaultNamespace"><id xsi:type="xsd:int">a</id></q1:getInput>
 </soap:Body>
</soap:Envelope>
```

In the above case we are requesting for character "a" in variable id which expects an integer as input.

Response: 500 Internal Server Error

```
HTTP/1.1 500 Internal Server Error
Date: Tue, 04 Jan 2005 16:22:14 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Set-Cookie: JSESSIONID=1CAF4CD0ED0F38FB40ECBC7BDAB56C75; Path=/axis
Content-Type: text/xml;charset=utf-8
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <soapenv:Fault>
      <faultcode>soapenv:Server.userException</faultcode>
```

```
      <faultstring>java.lang.NumberFormatException: For input string: &amp;quot;a&amp;quot;</faultstring>
   <detail/>
   </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

As we can see from the response screenshot above, faultcode may disclose critical internal information – reason enough to define an apply filters. In our example, we apply the following rules:

```
   <Location /axis/modsec.jws>
        SecFilterInheritance Off

        SecFilterDefaultAction "deny,log,status:500"
        SecFilterScanPOST On
        SecFilterCheckURLEncoding On
      SecFilterCheckUnicodeEncoding On


      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>" chain
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.{6,}</\s*id\s*>" "deny,status:500"
      SecFilterSelective POST_PAYLOAD "<\s*id[^>]*>.+['\"%][^<]*</\s*id\s*>" "deny,status:500"

      SecFilterScanOutput On
      SecFilterSelective OUTPUT "faultcode" "deny,status:500"
   </Location>
```

SecFilterScanOutput On

This line will scan an output data block and apply defined filters.

SecFilterSelective OUTPUT "faultcode" "deny,status:500"

This will block out going traffic which has “faultcode” in it.

Now if we send the above request again with “a” in the integer field we will get a response that resembles the output shown below:

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 03 Jan 2005 22:00:33 GMT
Server: Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
Content-Length: 657
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
```

```
<h1>Internal Server Error</h1>
<p>The server encountered an internal error or misconfiguration and was unable to complete your request.</p>
<p>Please contact the server administrator,  you@example.com and inform them of the time the error occurred, and anything you might have done that may have caused the error.</p>
<p>More information about this error may be available in the server error log.</p>
<hr />
<address>Apache/2.0.50 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4 Server  at 192.168.7.50 Port 80</address>
</body></html>
```

Conclusion

ModSecurity may seem like just another tool in the security firmament, but there is a subtle advantage over other security tools already available. While providing intrusion detection and defense capabilities at the HTTP layer, mod_security also allows POST PAYLOAD filtering capabilities.

However, the advantage mod_security offers is allowing developers and web administrators to defend web services without actually modifying the source code. This does not mean that shabby code is tolerable; it simply means that an effective defense of web services can be mounted without having to run through numerous lines of code.

This paper focused on just one of the techniques securing web services and an effective one too. Users are free to choose their own techniques of defending web services.