Securing Web Services with ModSecurity

This document is largely based on Shreeraj Shah's article of the same name that appeared in the June, 9th 2005 Oreilly OnLamp Security Dev Center posting (http://www.onlamp.com/pub/a/onlamp/2005/06/09/wss_security.html) Ryan C. Barnett, Director of Application Security Training at Breach Security, has updated numerous sections to reflect the advanced ModSecurity 2.0 rules language.

Web services are increasingly becoming an integral part of next-generation web applications, however they're also vulnerable to attacks. The nature of these attacks is the same as for traditional web applications, but the modus operandi is different. These attacks can lead to information leakage; further, they 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. Without good defense at the source code level, your application is in danger of compromise and exploitation. ModSecurity operates as an Apache web server module, ideal for defending web services against attacks that also include malicious POST data containing SOAP envelopes.

The Problem Domain

Web services have four main attack vectors:

- Variable-length buffer injection
- Meta character injection
- SQL injection
- SOAP fault code disclosure

Common firewall configurations allow incoming HTTP/HTTPS traffic to pass through unobstructed. Each of these attacks is, simply put, malicious incoming traffic camouflaged to look like legitimate HTTP/HTTPS traffic and therefore able to penetrate firewalls quite easily. This article examines ways and means to first distinguish between legitimate and malicious HTTP/HTTPS traffic, and then block such traffic. Doing so can mitigate port 80/443 attacks to a very great extent.

What Is the Solution?

Solutions can take many different forms, ranging from secure coding practices to proper input validation. One approach is to perform content validation for each incoming request and compare it with predefined rules. This approach stops malicious SOAP requests from penetrating to the web services source code level. ModSecurity can help in defending against all of the above attacks, if you have deployed and configured it properly for web services. This article discusses in detail how ModSecurity 2.0 can be an effective web services defense tool.

After deploying web services, it is important to provide a sound defense against different sets of attack vectors. Each attack vector needs a different defense strategy. Consider a case study of a bank--a fictitious one and meant to serve only as an example.

Blue Bank (*www.bluebank.example.com*) has recently deployed web services securely using ModSecurity. It provides banking services over the internet for its financial partners and clients. Its web services provide online customer services such as account balances, money transfers, and modifying customer information. Figure 1 illustrates the web services deployment architecture at Blue Bank.

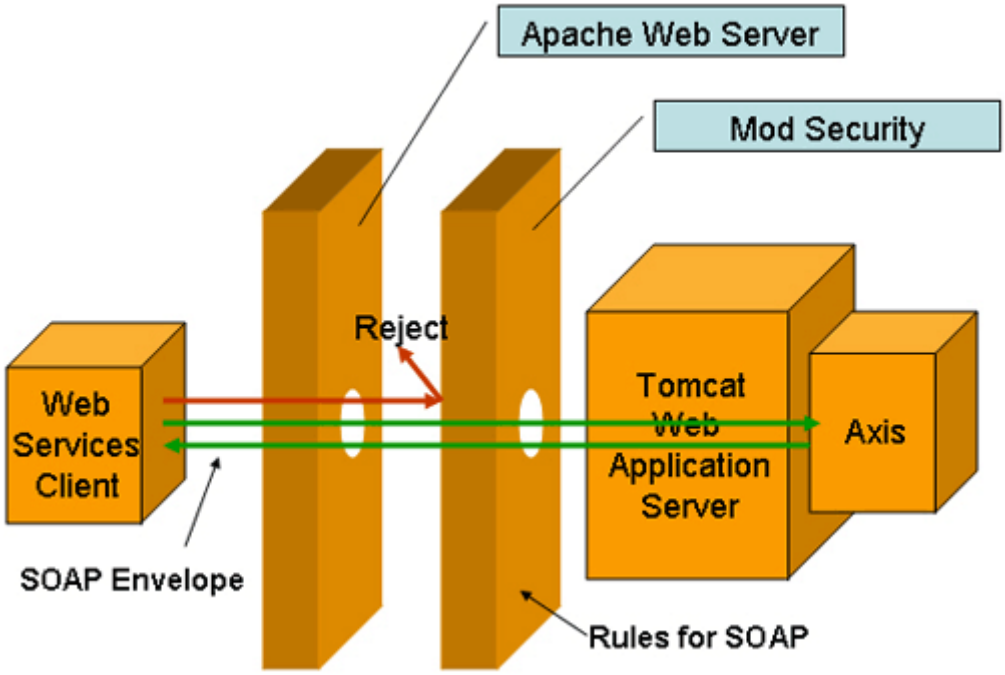

Figure 1. Deployment at Blue Bank

There are many different ways to deploy web services. In this case, the web service runs on Tomcat/Axis and plugs into the Apache web server. The *banking web services* application is Java code (with the .jws file extension).

Relevant sample snippets of Apache and Axis configuration files include the Apache *httpd.conf*, which loads Tomcat:

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

The Axis *web.xml* file, which supports AxisServlet for web services processing:

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

Once the above configuration is in place, you can deploy any web services on this server. If you observe the header information sent out with this server's responses, you can identify this line:

```
Server: Apache/2.2.0 (Unix) mod_ssl/2.0.50 OpenSSL/0.9.7d mod_jk2/2.0.4
```

The above header suggests that the web server is Apache/2.2.0 (Unix) running the Tomcat module mod_jk2/2.0.4. Axis runs as part of the Tomcat web application and is ready for the deployment of web services. To provide a defense at the web services level, Blue Bank has loaded the ModSecurity module that provides application-filtering capabilities. The following line in *httpd.conf* loads the security module:

```
LoadModule security2_module    modules/mod_security2.so
```

With ModSecurity in place, Blue Bank can add filtering rules from the Core Ruleset in *httpd.conf*:

```
<IfModule security2_module>
     Include conf/rules/*.conf
</IfModule>
```

Then, if you follow the steps outlined in the following Blog post – Handling False Positives and Create Custom Rules – you can then implement Web Service rules in a modsecurity_crs_15_customrules.conf file. With this, Blue Bank has everything in place for systems administrators and developers to build web services and use the ModSecurity's effective content filtering capability to defend against malicious incoming HTTP/HTTPS requests.

Consider a sample web service at *www.bluebank.example.com/axis/getBalance.jws*. The WSDL response for this specific web service comes from *www.bluebank.example.com/axis/getBalance.jws?wsdl*. Note: *www.bluebank.example.com* is a hypothetical domain used only as an example.

Method/Interface data derived from WSDL to show account balance

Scrutinize the WSDL response obtained as a result of the HTTP request passed. Of specific interest here is the invocation method, which takes the bank `id` and passes relevant account balance information back to the client over HTTP. The `operation` tag specifies methods that a web services client can invoke. The relevant snippets of the WSDL file include:

```
<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://www.bluebank.example.com/axis/getBalance.jws"
             use="encoded"/>
   </wsdl:output>

<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>
```

As shown above, passing an `id` to this particular method will cause it to return a `string` as output. When you pass a bank account `id` as input to the web service, you will receive the account `balance` available for that particular account `id`.

Invoking the web service over HTTP

Blue Bank's clients or partners seeking account balance information over the internet can fetch the requisite information by sending a properly formatted envelope to the banking web services. Figure 2 shows an HTTP request for account balance information for account `id 12123` being sent to the web service.

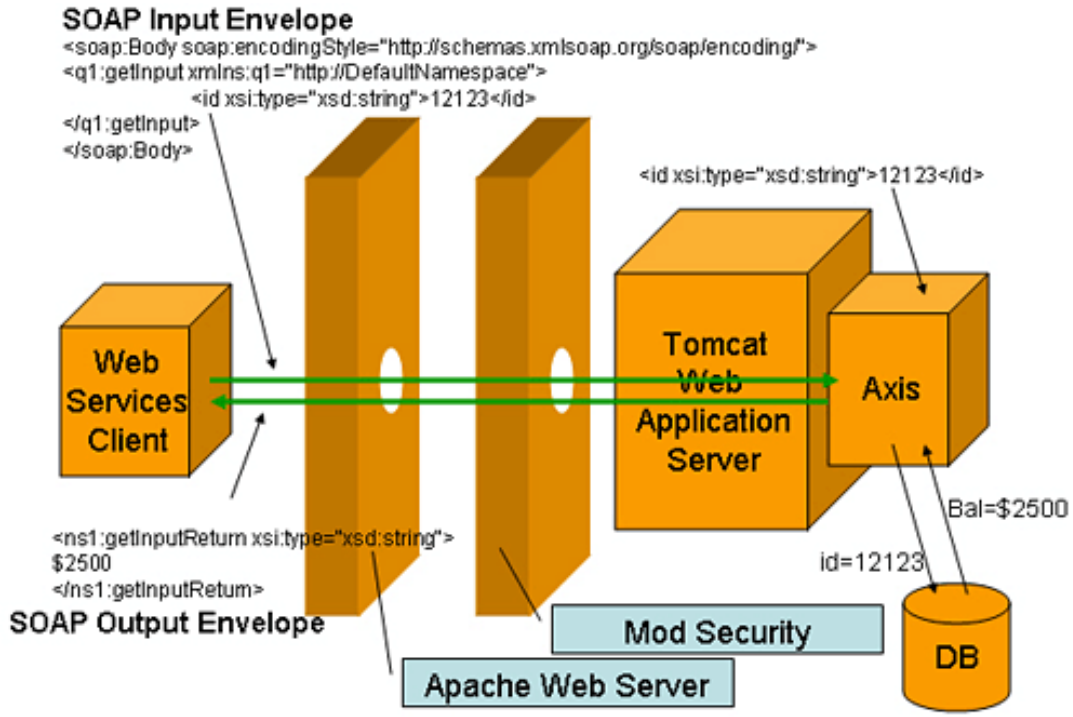

Figure 2. Invoking the web service over HTTP

There are several different ways to create SOAP clients that generate properly formatted SOAP requests. Here is an example of a SOAP client using SOAP::Lite in Perl. The following simple code generates a SOAP request on the wire.

```
#!perl -w
use SOAP::Lite;

print SOAP::Lite
  ->
service('http://www.bluebank.example.com/axis/getBalance.jws?wsdl')
  -> getInput('12123');
```

Another option for sending SOAP/XML requests is to use the Curl application. Let's say you have the following example SOAP/XML data in a file called "xml.post" -

```
<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">12123</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>
```

With curl, you can use command line options to quickly simulate sending XML POST data by using the "-d" flag to send the data in the xml.post file such as this –

```
# curl -d "`cat xml.post`" -H 'Content-Type: text/xml; charset=utf-8' -
H 'SOAPAction: ""' http://www.bluebank.example.com/axis/getBalance.jws
```

The advantage of using a tool such as curl for this task is that if you decide to alter the data in the xml.post file (to insert malicious attack payloads), curl will automatically correct the Content-Length header size. An HTTP/SOAP request sent to *www.bluebank.example.com* with the `id` of `12123` will generate something like:

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

```
Host: www.bluebank.example.com

<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">12123</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>

...

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">$2500</ns1:getInputReturn>
                </ns1:getInputResponse>
        </soapenv:Body>
</soapenv:Envelope>
```

Gluing a Web Services Resource into ModSecurity

Blue Bank's web service uses the URL *www.bluebank.example.com/axis/getBalance.jws*. It's generally a good idea to create a rule set for this resource. To do so, Blue Bank has glued its resource into *modsecurity_crs_15_customerules.conf* file with:

```
SecRuleEngine On
SecRequestBodyAccess On
SecResponseBodyAccess On
SecResponseBodyMimeType text/xml text/plain text/html
SecDebugLog logs/modsec_debug_log
SecDebugLogLevel 3
```

```
SecAuditLogType Serial
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4\d[^4])"
SecAuditLog logs/mod_audit_log
SecDefaultAction “phase:2,deny,log,status:500
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase”

# ------- Rules for web services -------------------------
      SecRule REQUEST_HEADERS:Content-Type "text/xml" \
      "phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML"

      <Location /axis/getBalance.jws>
        SecRule REQBODY_PROCESSOR "!^XML$" skip:1,t:none
        SecRule XML "@validateSchema /path/to/soapenvelope.xml"

        SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
        "^(RegEx)$" "phase:2,capture,log,deny,status:500, \
        xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
        xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
        Violation: %{TX.0}'"
      </Location>
#-------------------------------------------------------------
```

The following directive block applies filtering criteria for `/axis/getBalance.jws`. This adds the required placeholder for the defense of the web service. Placeholders go in the `<Location>` block.

```
SecRuleEngine On
SecRequestBodyAccess On
SecResponseBodyAccess On
SecResponseBodyMimeType text/xml text/plain text/html
SecDebugLog logs/modsec_debug_log
SecDebugLogLevel 3
SecAuditLogType Serial
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4\d[^4])"
SecAuditLog logs/mod_audit_log
SecDefaultAction “phase:2,deny,log,status:500
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase”

# ------- Rules for web services -------------------------
      SecRule REQUEST_HEADERS:Content-Type "text/xml" \
      "phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML"

      <Location /axis/getBalance.jws>
        SecRule REQBODY_PROCESSOR "!^XML$" skip:1,t:none
        SecRule XML "@validateSchema /path/to/soapenvelope.xml"

        SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
        "^(RegEx)$" "phase:2,capture,log,deny,status:500, \
        xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
        xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
        Violation: %{TX.0}'"
      </Location>
#-------------------------------------------------------------
```

There are five important actions taking place here:

1. Enabling Request Body Access

   Web services invocation methods go over `POST`. Hence, the next directive to enable is `SecRequestBodyAccess` to enable inspection of `POST` XML payload data –

   ```
   SecRequestBodyAccess On
   ```

2. Enabling the XML request body processor

   By default, ModSecurity does not use the an XML processor to parse the request body content. In order to enable this functionality, you must enable it in phase:1 so that it can then properly parse the data when it reads the request body content in phase:2.

   So, how do we know that the request body is in XML format? We inspect the Content-Type request header to identify the body type. The following rule handles this action –

   ```
   REQUEST_HEADERS:Content-Type "text/xml" \
   "phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML"
   ```

3. Setting up an XML location container

   In this scenario, there is only one location that utilizes web services functionality - /axis/getBalance.jws. Therefore, we want to contain our XML processing and inspection to only this location. We can do this by placing the remaining rules in an Apache <Location> scope directive container like this –

   ```
   <Location /axis/getBalance.jws>
     ModSecurity Rules…
   </Location>
   ```

4. Using the @validateSchema operator
   The "@validateSchema" operator is used to inspect the format and syntax of the XML body. Oftentimes, malicious requests will alter the various location fields and this rule will catch them quite easily.

5. Using the XPath variable location
   The main advantage that ModSecurity 2 has over its previous version is that it is able to utilize XPath variables to define specific input vectors in the XML body. In previous ModSecurity versions, you would have to inspect the entire POST_PAYLOAD data and use regular expressions to attempt to trap the desired XML field location. The ability to use XPath expressions increases the rules accuracy and significantly decreases false positives.

   ```
   SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
   ```

```
"^(RegEx)$" "phase:2,capture,log,deny,status:500, \
xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
Violation: %{TX.0}'"
```

It is important to also note that you will most likely need to use the "xmlns" action to specify XML Name Spaces that are required for XPath processing.

With the above details in place, Blue Bank has deployed a shield in the form of ModSecurity. It also knows its defense target--the content of `id`, which clients pass to the web service over a SOAP envelope.

Defending Attack Vectors

As a first step to defend all incoming malicious requests, Blue Bank needs to trap the value of `id` to prevent a client from passing an invalid value. The SOAP request uses an XML tag to pass this `id` information to the internal web services code, looking something like:

```
<q1:getInput xmlns:q1="http://DefaultNamespace">
     <id xsi:type="xsd:string">12123</id>
</q1:getInput>
```

To filter the request, ModSecurity must have some way to read the value associated with the tag; in this case it is 12123. ModSecurity provides the XML variable as a way to trap an XML field location value passed through a `POST` request. Here is an example of a customized filter:

```
SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
  "^(RegEx)$" "phase:2,capture,log,deny,status:500, \
  xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
  xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
  Violation: %{TX.0}'"
```

The highlighted line traps the request made to id by using an XPath expression which traverses the XML path to the targeted location. If the sent `XML` request has an `id` field contained in it, the server can process the information. However, a malicious client can modify this particular value to inject malicious content into it. There are four main popular attack vectors.

Attack vector 1: variable-length buffer injection

One of the major security concerns when passing a large buffer to a variable is that a large buffer may cause the application to misbehave and/or crash somewhere down the line during execution. The following rule defends the variable `id` against precisely this type of attack:

```
SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
  "^(.{6,})$" "phase:2,capture,log,deny,status:500, \
```

```
        xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
        xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
        Violation: %{TX.0}'"
```

In the above directive, the regular expression pattern "^(.{6,})$" restricts the buffer to allow only a buffer of five characters. In order to ascertain the effect of the above directive block, Blue Bank can send across two requests, one that matches the buffer length set and the other that exceeds the buffer length set in the directive block.

```
POST /axis/getBalance.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 576
Expect: 100-continue
Host: www.bluebank.example.com

<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">12123</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>

...

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">$2500</ns1:getInputReturn>
                </ns1:getInputResponse>
        </soapenv:Body>
```

```
</soapenv:Envelope>
```

In the above case, a buffer of five characters is passed, and the server sent a response with the value of `$2500`. Here's the response and request with a modified `id` of `121234` (six characters):

```
POST /axis/getBlalance.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 577
Expect: 100-continue
Host: www.bluebank.example.com

<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">121234</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>

...

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 />
 </body></html>
```

The ModSecurity module has rejected this request. Status 500 is the response received. This indicates that the request never hit the web services level. Blue Bank has succeeded, therefore, in providing a sound defense against any kind of buffer overflow, the most common and often ignored vulnerability.

Attack vector 2: meta-character injection

Another major threat to input variables is the use of metacharacters such as `%`, single quote (`'`), and double quotes (`"`). These characters can cause SQL injection attacks to occur and may also cause unnecessary information leakage. Adopting the following strategy provides a sound defense against such attacks.

```
SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
  "^(.{6,}|[^\d])$" \
"phase:2,capture,log,deny,status:500, \
  xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
  xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
  Violation: %{TX.0}'"
```

Instead of taking a negative policy enforcement methodology and listing every possible meta-character that could potentially cause problems, it is much wiser to use a positive policy rule to enforce specific data types. The regular expression pattern `"[^\d]"` denies an HTTP request if the id XPath variable's data consists of any non-digit characters. The ability to create positive security rulesets by either forcing regular expression character classes or by using ModSecurity's rule inversion (with the "!" character) is an important technique in ModSecurity.

Here is the request and response sent using an `id` of `12'12`:

```
POST /axis/getBalance.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 576
Expect: 100-continue
Host: www.bluebank.example.com

<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">12'12</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>
```

```
...

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
```

This attack also failed, and ModSecurity caught it.

Attack vector 3: SQL injection

It is also possible to inject SQL statements in variables. Concatenating together many SQL statements is possible. If your application does not sanitize its input, malicious clients can append additional SQL statements to existing SQL queries, often with disastrous consequences. Fortunately, Blue Bank already has protections to block SQL injection attacks by using the previous positive policy filter! When you combine both the size restrictions and only allowing digit characters, it is not possible to successfully execute an SQL infection attack.

Attack vector 4: SOAP fault code disclosure

One of the major sources of information in web services is the fault code. A forced error on the server can create a fault code. This type of detailed error message may aid an attacker by divulging technical information that could help them to fine tune their next attack. Let's assume that either an attacker somehow found a method to evade our inbound protection rules or if we didn't implement any inbound protections and was able to cause an error with our web service. Here's the request and response from a malicious user probing Blue Bank by sending a request with the character `a` in the `id` variable instead of an integer:

```
POST /axis/getBalance.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 569
Expect: 100-continue
Host: www.bluebank.example.com

<?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://www.bluebank.example.com/axis/getBalance.jws"
xmlns:types="
http://www.bluebank.example.com/axis/getBalance.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">a</id>
                </q1:getInput>
        </soap:Body>
</soap:Envelope>

...

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:"a"</faultstring>
   <detail/>
   </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

As shown in the response, the fault code may disclose critical internal information. That's reason enough to define and apply filters. Blue Bank can fix this by applying the following rules:

```
SecRuleEngine On
SecRequestBodyAccess On
SecResponseBodyAccess On
SecResponseBodyMimeType text/xml text/plain text/html
SecDebugLog logs/modsec_debug_log
SecDebugLogLevel 3
SecAuditLogType Serial
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus "^(?:5|4\d[^4])"
SecAuditLog logs/mod_audit_log
SecDefaultAction “phase:2,deny,log,status:500
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase”

# ------- Rules for web services -------------------------
      SecRule REQUEST_HEADERS:Content-Type "text/xml" \
      "phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML"

      <Location /axis/getBalance.jws>
```

```
        SecRule REQBODY_PROCESSOR "!^XML$" skip:1,t:none
        SecRule XML "@validateSchema /path/to/soapenvelope.xml"

        SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id/text() \
        "^(.{6,}|[^\d])$" "phase:2,capture,log,deny,status:500, \
        xmlns:soap=http://schemas.xmlsoap.org/soap/envelope/, \
        xmlns:q1=http://DefaultNamespace,msg:'Client Input Data \
        Violation: %{TX.0}'"

        SecRule RESPONSE_BODY
        "\<fault(code|string)\>.*\\</fault(code|string)\>" \
        "phase:4,deny,status:500,capture,msg:'SOAP fault code disclosure
        detected: %{TX.0}'"
      </Location>
#-------------------------------------------------------------
```

In order to scan the response body data, we need to set the SecResponseBodyAccess directive to On. Additionally, we need to make sure to add the “text/xml” mime-type to the SecResponseBodyMimeType directive.

```
SecResponseBodyAccess On
SecResponseBodyMimeType text/xml text/plain text/html
```

We can then apply this rule that will inspect the RESPONSE_BODY variable and look for the fault-code html tags.

```
SecRule RESPONSE_BODY
"\<fault(code|string)\>.*\\</fault(code|string)\>" \
"phase:4,deny,status:500,capture,msg:'SOAP fault code disclosure
detected: %{TX.0}'"
```

This directive blocks outgoing traffic that contains fault codes. If the attacker sends the ill-formed request again with a in the integer field, he will receive a response more like:

```
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 />
</body></html>
```

ModSecurity, properly configured, has also defended against this information leakage attack.

Conclusion

ModSecurity may seem like just another tool in the security firmament, but it has a subtle advantage over other security tools already available. While providing intrusion detection and defense capabilities at the HTTP layer, ModSecurity also allows both Request and Response body filtering capabilities.

However, the advantage ModSecurity offers is in allowing developers and web administrators to defend web services without actually modifying their source code. This doesn't make shabby code tolerable; it simply means that an organization can mount an additional effective defense of its web services without having to run through numerous lines of code.

This article focuses on just one of the techniques for securing web services, and an effective one too. Feel free to choose your own techniques of defending web services.

About Ryan C. Barnett
Ryan C. Barnett is the Director of Application Security Training at Breach Security. He is also a Faculty Member for the SANS Institute, where his duties include Instructor/Courseware Developer for Apache Security/Building a Web Application Firewall Workshop, Top 20 Vulnerabilities Team Member and Local Mentor for the SANS Track 4, "Hacker Techniques, Exploits and Incident Handling" course. He holds six SANS Global Information Assurance Certifications (GIAC): Intrusion Analyst (GCIA), Systems and Network Auditor (GSNA), Forensic Analyst (GCFA), Incident Handler (GCIH), Unix Security Administrator (GCUX) and Security Essentials (GSEC). In addition to the SANS Institute, he is also the Team Lead for the Center for Internet Security Apache Benchmark Project and a Member of the Web Application Security Consortium. Mr. Barnett has also authored a web security book for Addison/Wesley Publishing entitled "Preventing Web Attacks with Apache."

About Shreeraj Shah
Shreeraj Shah, BE, MSCS, MBA, is the founder of Net-Square and leads Net-Square's consulting, training and R&D activities. He previously worked with Foundstone, Chase Manhattan Bank and IBM. He is also the author of Hacking Web Services (Thomson) and co-author of Web Hacking: Attacks and Defense (Addison-Wesley). In addition, he has published several advisories, tools, and whitepapers, and has presented at numerous conferences including RSA, AusCERT, InfosecWorld (Misti), HackInTheBox, Blackhat, OSCON, Bellua, Syscan, etc. His

articles are published on Securityfocus, O’Reilly, InformIT and HNS. You can read his blog at shreeraj.blogspot.com.