Getting started with OWASP WebGoat 4.0 and SOAPUI.

Hacking web services, an introduction.

Version 1.0 by Philippe Bogaerts

mailto:philippe.bogaerts@radarhack.com
http://www.radarhack.com

1. Introduction

SOA, web services, WS-security and lot of other related protocols and technology are becoming at fast pace business critical corner stones of today's IT infrastructures and business applications. Security efforts must undoubtedly focus more and more on the applications in use, simply because this is were companies are the most vulnerable today and can be impacted the most when applications, (read: the business processes) , are adversely used.

This paper should serve as a starting point for everyone that wants to learn, in a practical way, the basics of web services and how they can be exploited. This paper serves the one and only purpose of education and awareness creation, towards people wanting the world to become a better and saver world.

The tools used in this paper are freely available at http://www.owasp.org and http://www.soapui.org.

2. A word on WebGoat 4.0

From the OWASP website:
"WebGoat is a deliberately insecure J2EE web application maintained by OWASP designed to teach web application security lessons. In each lesson, users must demonstrate their understanding of a security issue by exploiting a real vulnerability in the WebGoat application. For example, in one of the lessons the user must use SQL injection to steal fake credit card numbers. The application is a realistic teaching environment, providing users with hints and code to further explain the lesson."

More info can be found at:
http://www.owasp.org/index.php/Category:OWASP_WebGoat _Project

3. A word on Soapui

From the SOAPUI website:
"Soapui is a desktop application for inspecting, invoking, developing and functional/load/compliance testing of web services over HTTP. It is mainly aimed at developers/testers providing and/or consuming web services (java, .net, etc). Functional and Load-Testing can be done both interactively in soapui and within an automated build/integration process using the soapui command-line tools. Soapui currently requires java 1.5 and is licensed under the LGPL license."

More info can be found at:
http://www.soapui.org/

4. Installing WebGoat 4.0

Installing WebGoat 4.0 is a straight forward process. Simply download the zipped binaries via the WebGoat project pages on http://www.owasp.org. This paper is based on the Windows_WebGoat -4.0_Release.zip of the tool.

1. Unzip Windows_WebGoat -4.0_Release.zip in a directory of your choice.

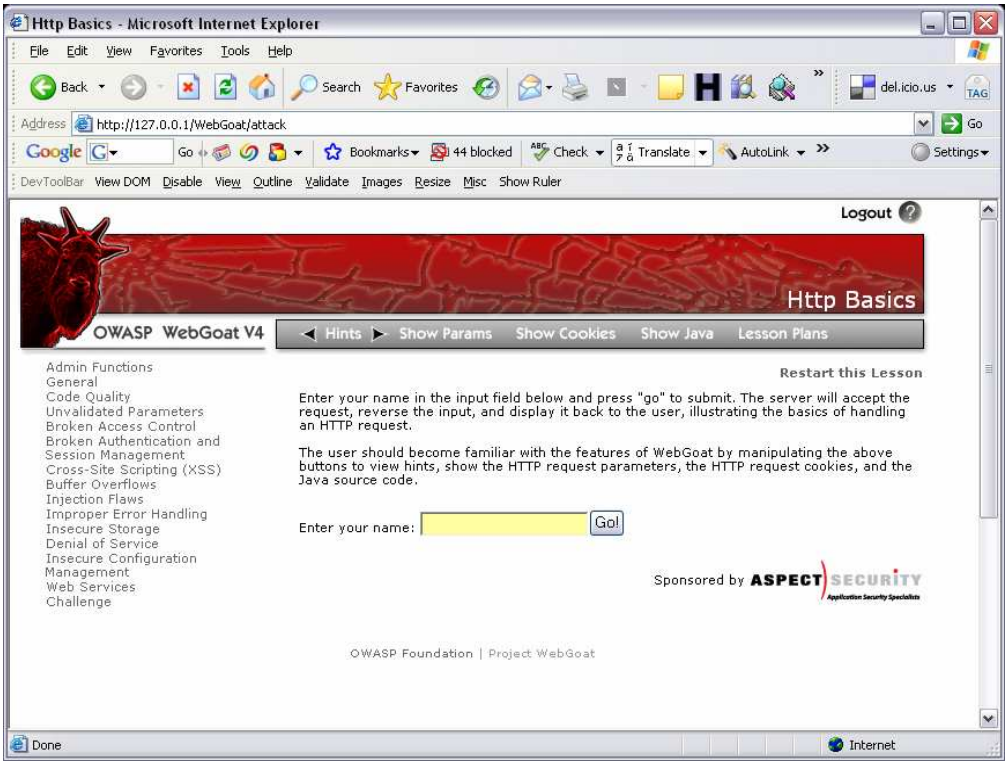
2. Make sure that all other web servers running on port 80 are stopped. Stop Microsoft IIS services and Apache services via the control panel if they were previously installed. Especially pay attention to Skype, it can/will use port 80 when available on startup and will inhibit WebGoat from booting correctly.

   Note: Use "netstat –an" on the command line to verify that port 80 is not in use.

3. Click *WebGoat.bat* in the installation directory and a command shell window will display the WebGoat starting process. If everything goes as planned, it will display a message like *"INFO: Server startup in 4719 ms".*

4. Connect to http://127.0.0.1/WebGoat /attack (mind the capital letters in the URL) and login with the username *guest* and password *guest*.

5. A nice welcome page is displayed in your browser inviting you to click start.



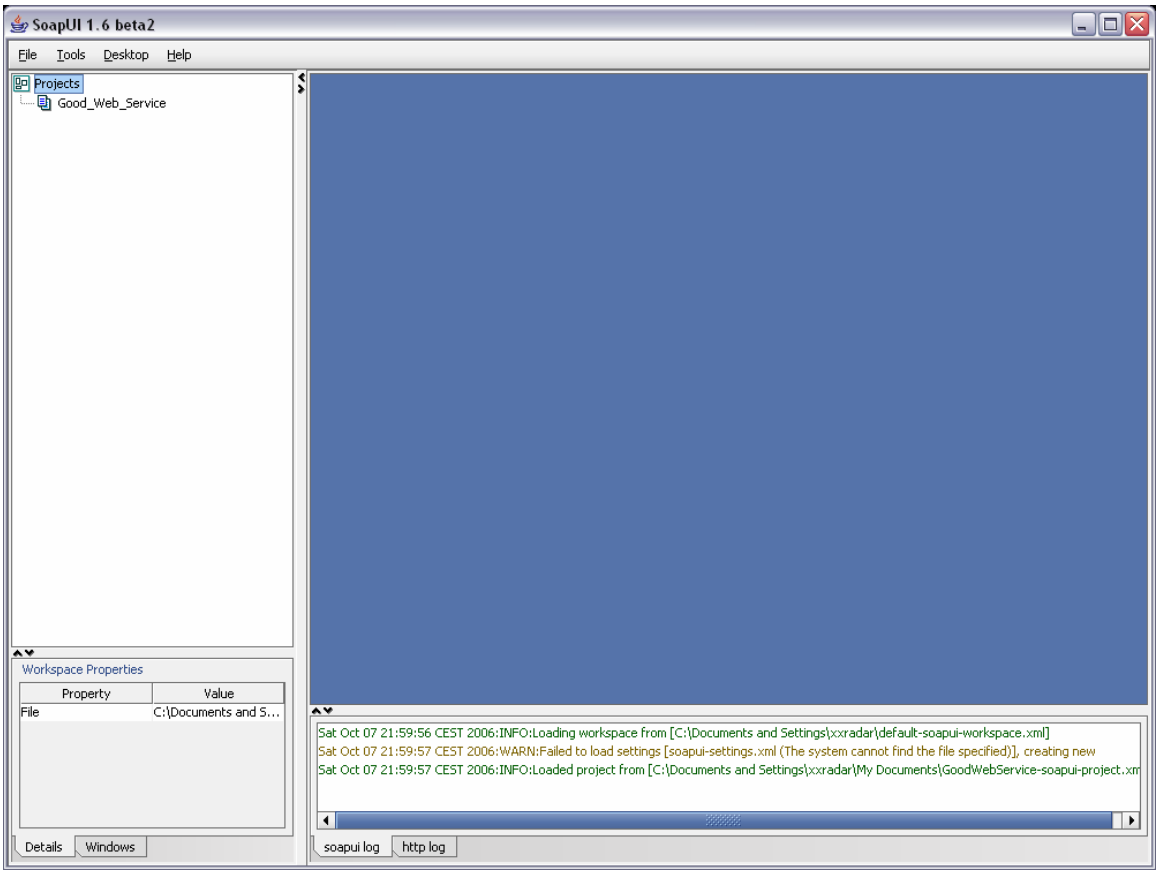
5. Installing soapui

Installing soapui is very easy. Just click the *"Webstart"* button on the http://www.soapui.org. Throughout the paper, soapui 1.5 is used, to guarantee the best results, although most functionality is working fine in the beta versions available on the website. Another option to install soapui is to download the binaries. This latter option is used in this paper.

1. Download the soapui 1.5 binaries (soapui-1.5-bin.zip)

2. Unzip the archive in a folder of your choice

3. Double click in the *soapui-1.5\bin* folder, the *"soapui.bat"* icon. Soapui should be started and present you a nice looking interface.



6. A hair rising explanation of web services

Web services are about applications communicating with other applications, opposed to a user to application communication model.

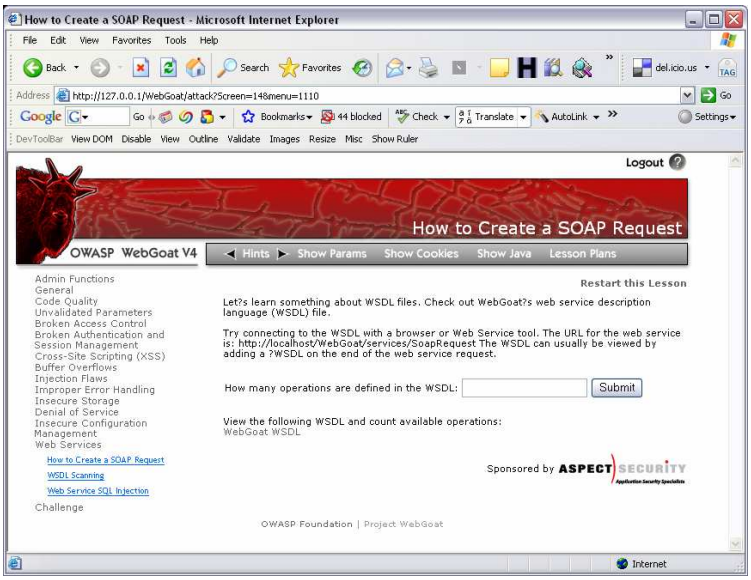
New applications might want to communicate to “reuse existing services” already offered by applications on the corporate network, partner network or simply somewhere available on the internet.

Imagine you want to develop a simple network monitoring application that sends an SMS when a certain host on your network goes down. The first part could be easily achieved by a simple ping command (or something more fancy if you have the time ☺), but sending the SMS when something goes wrong, might be much more difficult and a much more expensive undertaking. Wouldn’t it be handy if you could, with some few lines of code, reuse the existing SMS system already in place within the company’s mail-to-SMS service application?

Well, web services make this possible. A web services infrastructure provides you with a simple, documented and standardized way of invoking a remote service. One of the building blocks of web services is XML. XML is a way to represent the data being exchanged between systems in an unambiguously way, independently of the OS or development environment in use.

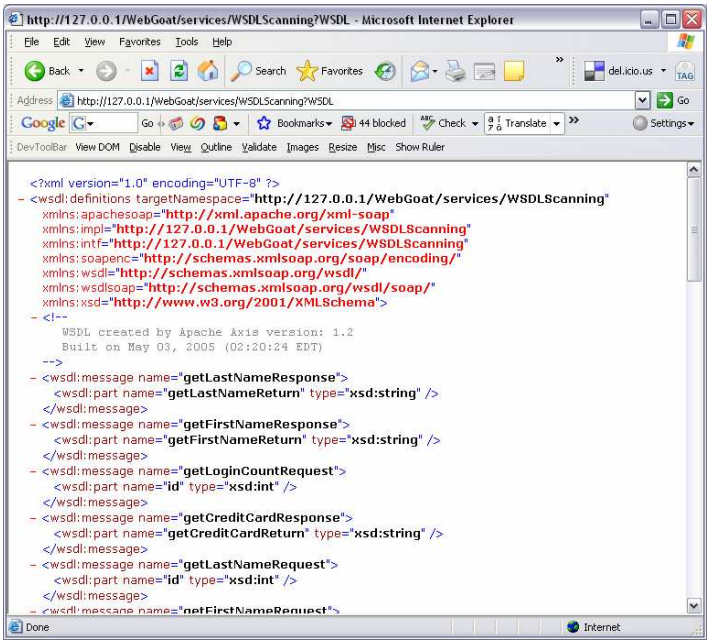
So, the first question is, where do I find the service I potentially want to reuse? Companies might use an UDDI repository in which developers (or applications) can look for available services. In our case (and often the case), the information about the web service is provided on a webpage.

Here is a starting link in WebGoat:
http://127.0.0.1/WebGoat/attack?Screen=14&menu=1110



Once you know what service is suitable for your application, it is of course mandatory to know how to invoke the remote service (what operations are available, what syntax needs to be used, what parameters need to be passed, what responses can be expected…). All this is described in the corresponding WSDL file (Web Services Description Language).

Take a look at the WSDL file of one of the web services available in WebGoat 4.0. It might be possible that you need to authenticate again, using *guest* as the username and password. http://127.0.0.1/WebGoat/services/WSDLScanning?WSDL

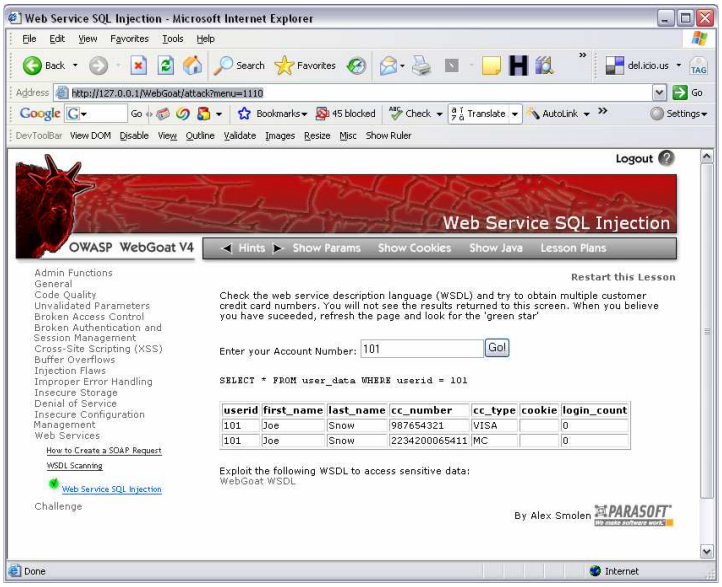


This WSDL file contains all the information your application needs to invoke the remote services. Typically, your application downloads the WSDL file and is able to craft the necessary requests and interpreting the responses. This can be achieved already with a few line of PERL code or soapui!

Where does SOAP come in the picture? Web services are designed to be totally independent of the underlying network protocols, whether you use TCP, UDP, SMTP, FTP or HTTP. An independent layer and standardized protocols on top all these protocols is necessary to exchange our service related messages between applications. In our example, SOAP is almost pure overhead, but in more complex environments SOAP is used to address web services more accurately, route messages and much more.

7. Enough theory, let's start playing

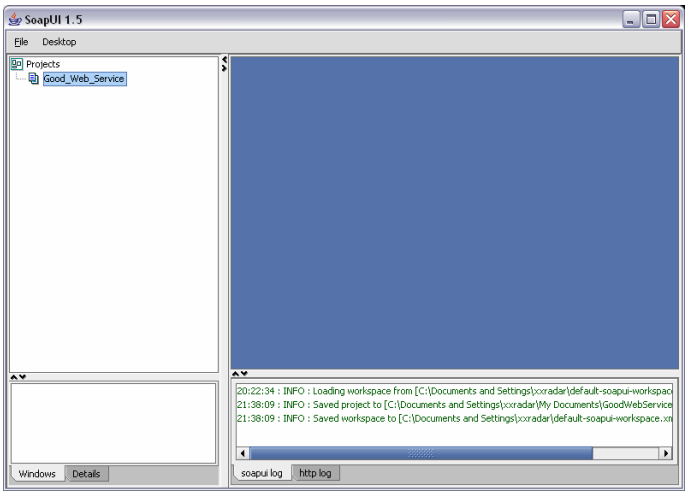
Visit http://127.0.0.1/WebGoat/attack?menu=1110. The information provided in the web interface, the Account Number, is received by an underlying java application when clicking the submit button. The java application builds the correct SOAP message (corresponding the WSDL file) and sends it to the web service that interrogates a database for the associated credit card numbers. A SOAP response message received by the java application is interpreted and the results are displayed in the browser.



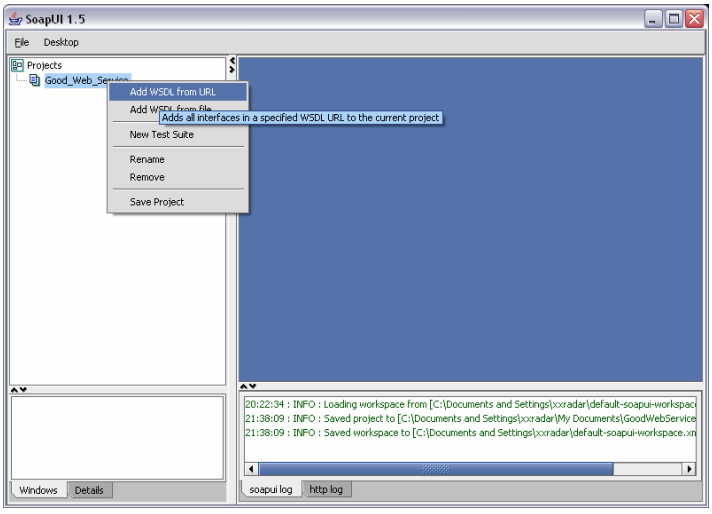
8. Invoking the web services directly

Instead of using the web interface, we can try to access the web service directly. The web page provides a link to the WSDL file describing a service to retrieve credit card numbers. (http://127.0.0.1/WebGoat/services/WsSqlInjection?WSDL)

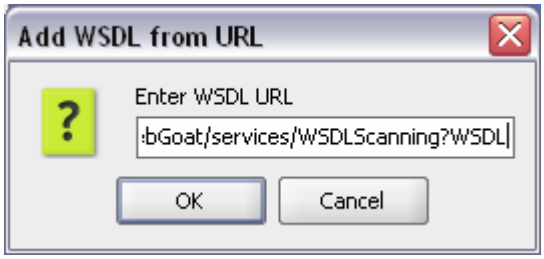
1. Open soapui, create a new “WSDL project” and name it “Good_Web_Service”, and save the project file to disk when prompted.



2. Next important step, import all information necessary to send and receive correct soap messages, because, as explained, this is how you interrogate web services!



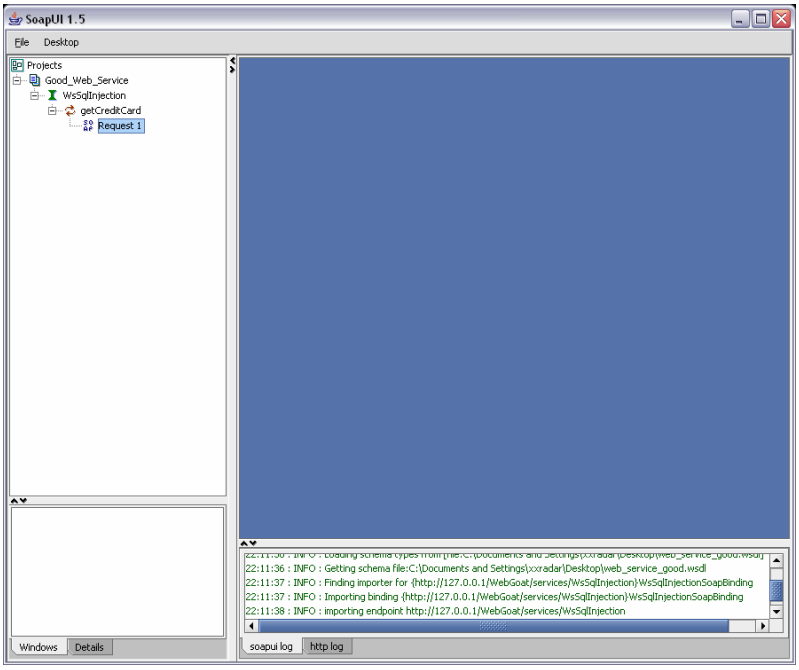
3. Click Add WSDL from URL, provide the URL for the WSDL file and click OK.



(You might be prompted for authentication. Always use the “guest” account)

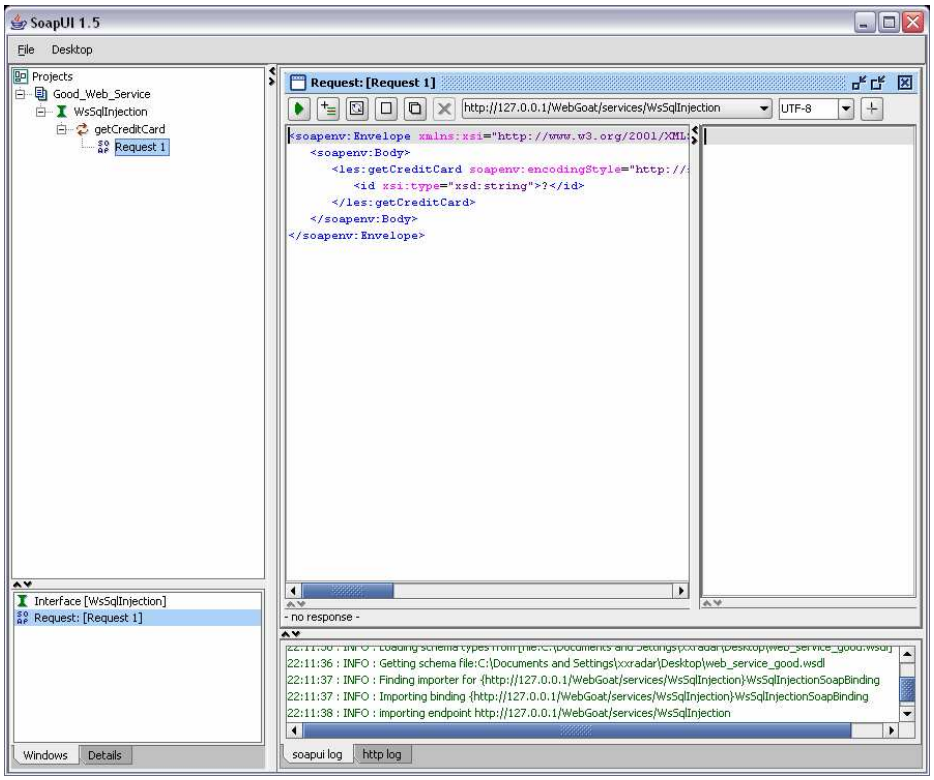
Note: soapui sometimes displays an error exception when downloading the WSDL file. If you experience any problems, just open the same link in a browser and save the file as “web_service_good.wsdl” and import it into soapui.

4. Soapui now prompts to create all default requests for all operations. This simply means that soapui can build the correct messages to interrogate the web service, based on the WSDL file just imported.



5. Double click “Request 1” in the interface and soapui shows a kind of template of the soap message to be

send (over http in this case) to the web service.

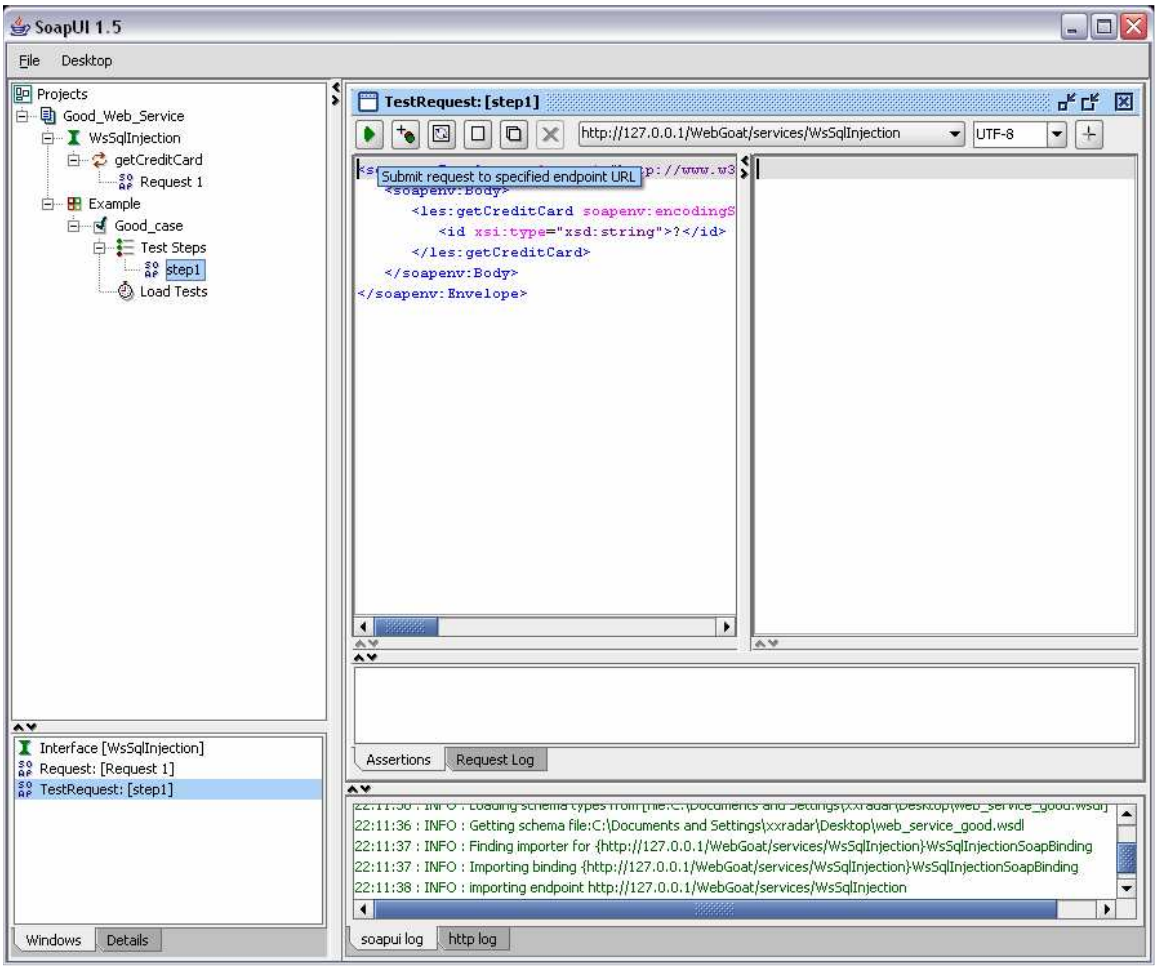


A closer look at the message reveals the structure of the SOAP message. The service request is encapsulated in a SOAP envelop. The SOAP envelop contains an optional SOAP header (not present in this case) and a mandatory SOAP body.

The interesting part to notice is the XML message in the SOAP body. This XML message is build according to the information in the WSDL file and is different for most web services available.

To illustrate the full picture, this soap message is send across the network within typically http(s), using the POST method to the service endpoint http://127.0.0.1/WebGoat/services/WsSqlInjection.

6. To easily use the service, create a test case by right-clicking *“request 1”.*

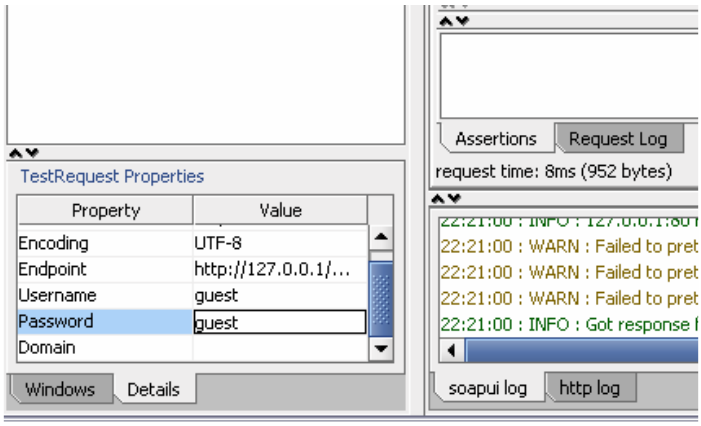


7. Now take a look at the SOAP message in the test case and change the id (Account Number) “?” in “101”. Other valid id values are 102 and 103.

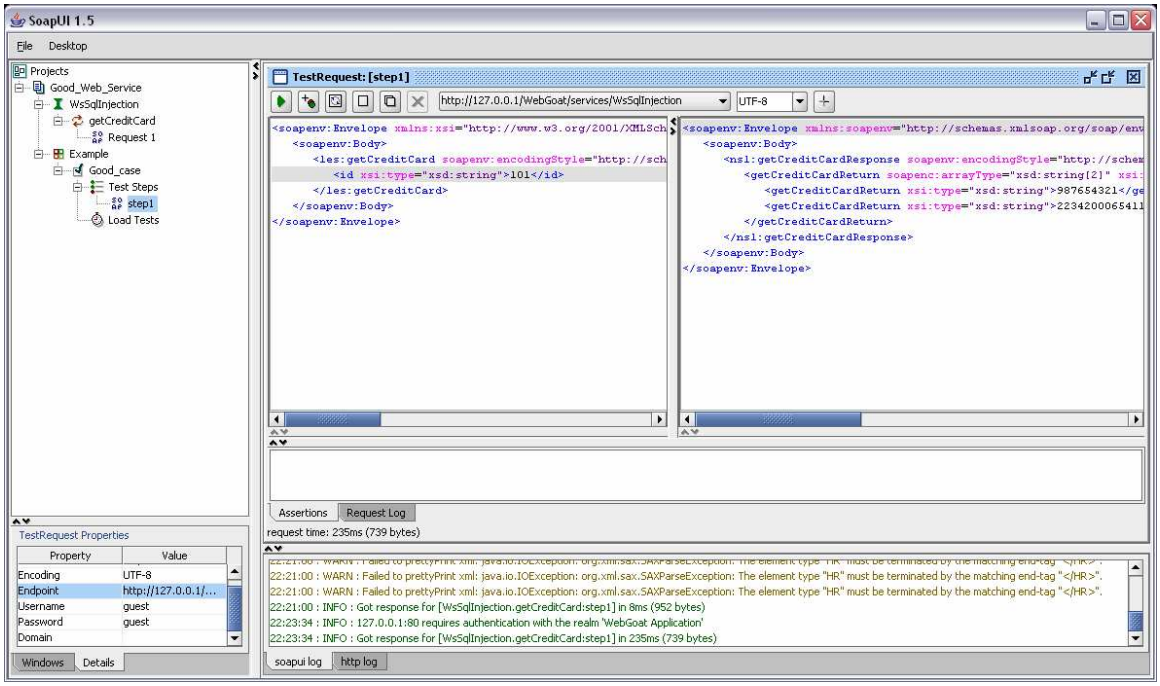
```
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:les="http://lessons.webgoat.owasp.org">
   <soapenv:Body>
      <les:getCreditCard
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <id xsi:type="xsd:string">101</id>
      </les:getCreditCard>
   </soapenv:Body>
</soapenv:Envelope>
```

8. Before you click the play button, make sure to tell soapui it needs to authenticate to use the service. The authentication mechanism is use is “basic HTTP authentication”

   Note: In most cases, the authentication step is not necessary. A lot of web services on the internet do not need authentication.



9. When done, click the play button in the interface.



10. A quick look at the SOAP response reveals the requested information.

```
<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:getCreditCardResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://lessons.webgoat.owasp.org">
         <getCreditCardReturn soapenc:arrayType="xsd:string[2]" xsi:type="soapenc:Array"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
            <getCreditCardReturn xsi:type="xsd:string">987654321</getCreditCardReturn>
            <getCreditCardReturn xsi:type="xsd:string">223420065411</getCreditCardReturn>
         </getCreditCardReturn>
      </ns1:getCreditCardResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Remark: The SOAP messages in these examples have no SOAP header. The SOAP header is optional.

11. Create new test cases and change the id value in some other valid and invalid values. Monitor the SOAP error messages, when the service fails to fulfill the request.

9. Time to hack

Revisit the web page to obtain associated credit card numbers an account number. As explained, to fulfill the request, the web services interrogate a database for the information linked to the account number. In plain words, the account number will be part eventually of a SQL statement. This is a snippet of code that builds the SQL statement:

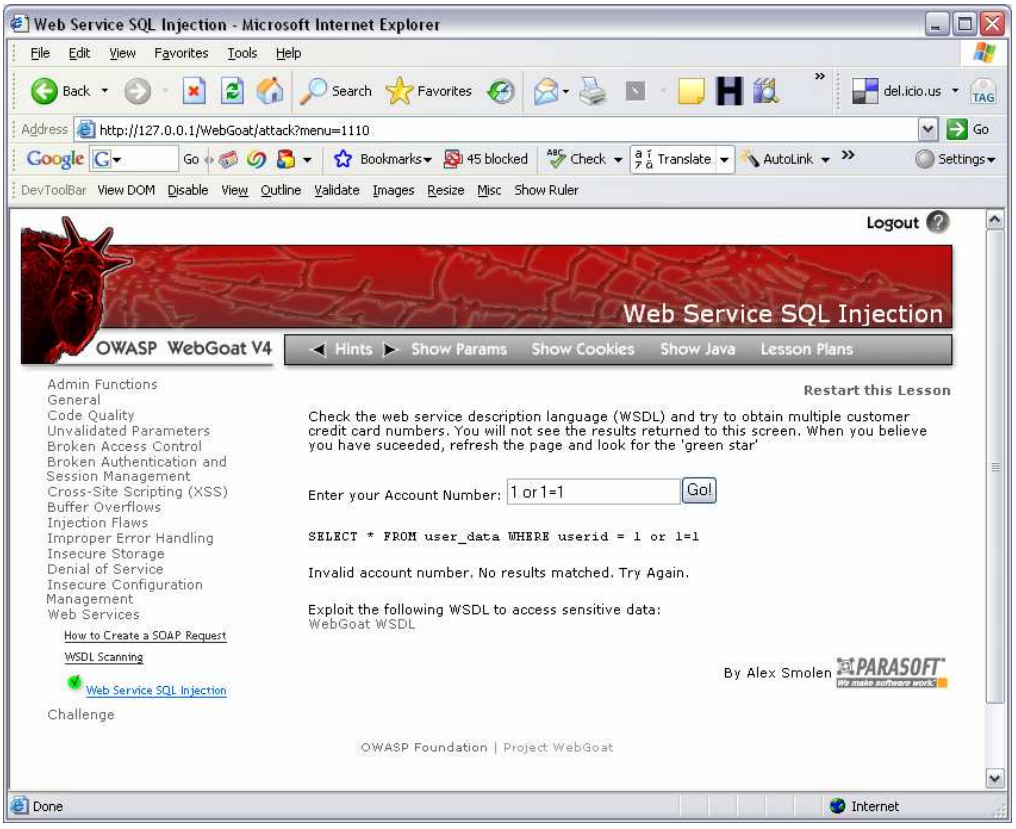
```
String query = "SELECT * FROM user_data WHERE userid = " + accountNumber ;
```

This is really bad! The accountNumber is simply appended to string that will be serve as the SQL statement. So in normal conditions, this would be:

```
SELECT * FROM user_data WHERE userid = 101;
```

Now use your imagination. What would happen if I could execute `SELECT * FROM user_data WHERE userid = 101 or 1=1;`?

Let's try this!

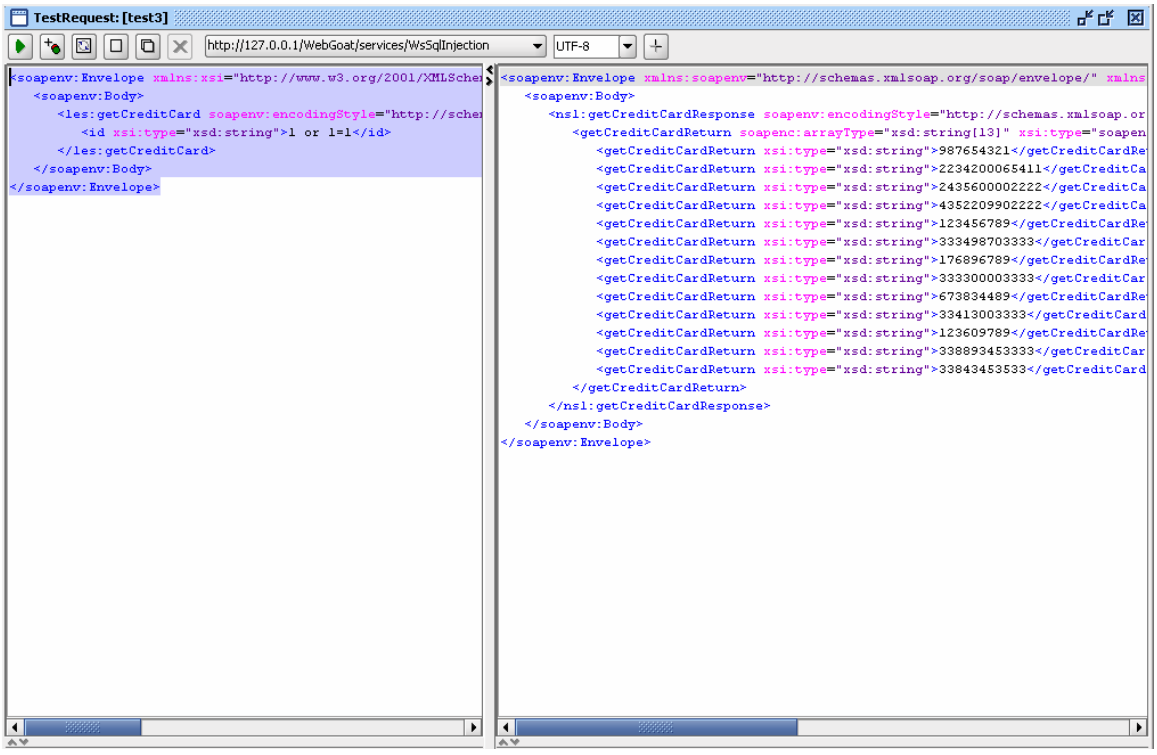


As you can see, the web application does not except this trick. But perhaps, the protection is build within the java code and not in the web service being used to interrogate the database.

Let’s create a SOAP message that interrogates the database with *1 or 1=1*.
Create a new test case (think about the authentication) and adjust to your needs.

```
<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:les="http://lessons.webgoat.owasp.org">
   <soapenv:Body>
      <les:getCreditCard
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <id xsi:type="xsd:string">1 or 1=1</id>
      </les:getCreditCard>
   </soapenv:Body>
</soapenv:Envelope>
```

Well, you’re supposed to be rich at this moment, if it wasn’t a training application ☺!



10. Conclusion

Web services might be vulnerable to the same type of attacks as web applications. It is important to notice that every component of the application needs to be secured and coded with security mind. Just imagine what would happen if this web service would be available and reused by other applications?

I hope to have guided any reader through an exciting and at first sight complicated world of web services and XML related technology and hope this paper might serve as a good starting point.

If you have any questions, comments or come across mistakes, feel free to drop me an email at xxradar@radarhack.com.