INTRODUCTION TO ASSESSING AND SECURING WEB SERVICES

Christoff Breytenbach

Security Analyst, SensePost

christoff@sensepost.com

+27 12 460 0880

PO Box 10692, Centurion, 0046

ABSTRACT

The primary purpose of the paper is to provide an introduction to security related problems in web services implementations, describe approaches used to identify these issues, and provide brief recommendations to resolve these problems.

Questions such as the following are important in this respect:

- How does the web service authenticate the service consumer or client?
- How does the client authenticate the web service?
- Is data protected between the web service provider and client?
- Does the web service provide an adequate authorisation framework to ensure user privileges are uniformly and consequently enforced?
- Does the application properly clean client or requester input?

Identification and exploitation of vulnerabilities in the above areas will be practically illustrated. Even though tools are important in this area, the analyst has to have a good understanding of the technology in question. The paper will focus on the high-level critical thinking that needs to be applied in assessing and securing web services.

KEY WORDS

Web services, WS-Security, UDDI, WSDL, SOAP

INTRODUCTION TO ASSESSING AND SECURING WEB SERVICES

1 INTRODUCTION

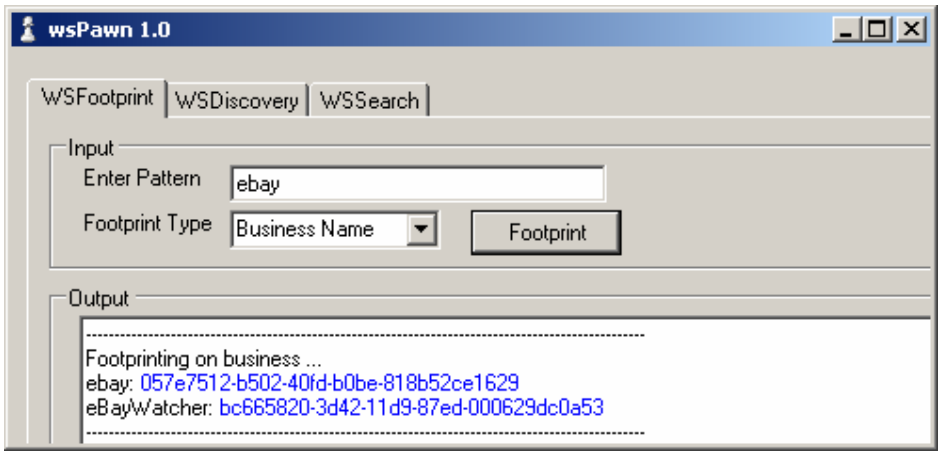
Web services provide a way to fulfil the intra- and inter-organisational need for sharing of information in a way that is platform independent and more standardised than traditional CGI scripts. Web services can be viewed as a collection of functions exposing business data or business processes. The core building blocks of web services consist of XML (eXtensible Markup Language) based SOAP (Simple Object Access Protocol) messages transmitted over HTTP or HTTP/S. Web services methods are exposed through WSDL (Web Services Description Language) and the web service and associated WSDL can be publicly listed through UDDI (Universal Description, Discovery and Integration). The use of widely accepted standards as building blocks for web services ensures one of its most important goals is achieved: a platform independent method for sharing information and functionality.

From an attacker's perspective, we find web services to be susceptible to many of the same attack vectors found in conventional web applications. Without developers and implementers of web services being aware of the pitfalls and security risks, organisations are opening gaping holes ripe for the picking by attackers. A typical high-level overview to assessing and securing web services will be provided in this document.

2 FOOTPRINTING WEB SERVICES

When assessing web services from an internet vantage point, we normally only have access to the name of the organisation we want to attack. If the assessment is going to be conducted in such a zero knowledge fashion, we need to first identify web services that the target organisation publishes. UDDI represents a registry of online businesses that wish to publicly expose web services to consumers. This is manifested through the concept of a central business registry. Similar to Trusted Certification Authorities in the PKI world; four companies (IBM, SAP, Microsoft, NTT-Com) currently provide a Universal Business Registry (UBR) service.

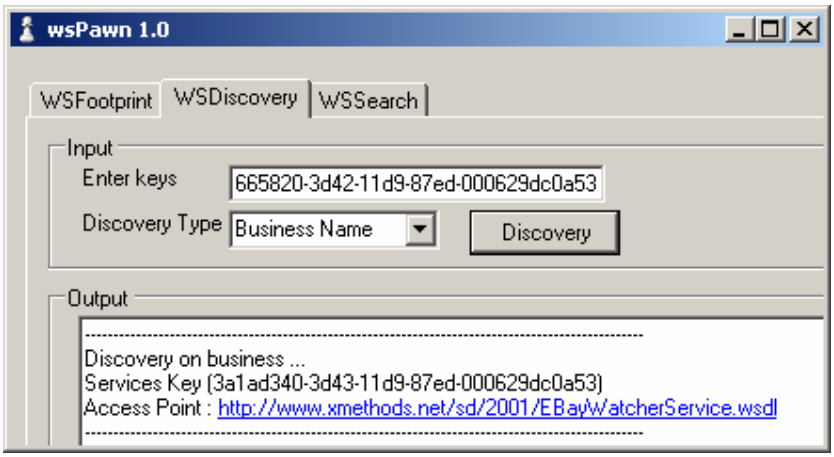
To query a Universal Business Registry, you can use wsPawn (part of the wsChess toolkit from Net-Square - http://net-square.com/wschess/index.html):



Using wsPawn to query UDDI for services by business name

As can be seen in the above figure, the UDDI response yields what is known as a *businessList* containing two entries and their associated business keys. A *find_service* query to UDDI, will

respond with a *serviceList* containing service keys for the particular business. Subsequently, a call to *get_serviceDetail* will reveal the service access point (WSDL):



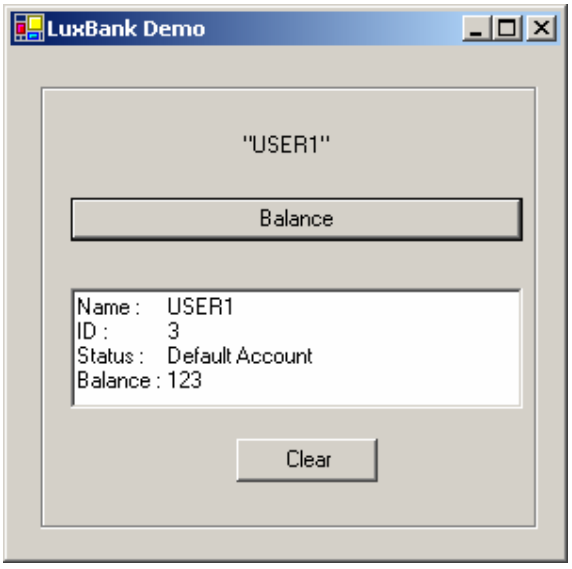
Using wsPawn to query UDDI on service key

What has to be taken note of is that many organisations may not wish to publish their web services in a public business directory, only wish to communicate with already trusted third parties or intends the web service for internal use only, in other words not exposing it to the internet. UDDI version 3 makes provision for multiple layers at which registries can be integrated, be that private through public. Once again, if intended for private use you will most likely not be able to query the UDDI service directly. When targeting most organisations, footprinting web services by means of UDDI may yield limited results.

3 WEB SERVICES PROFILING

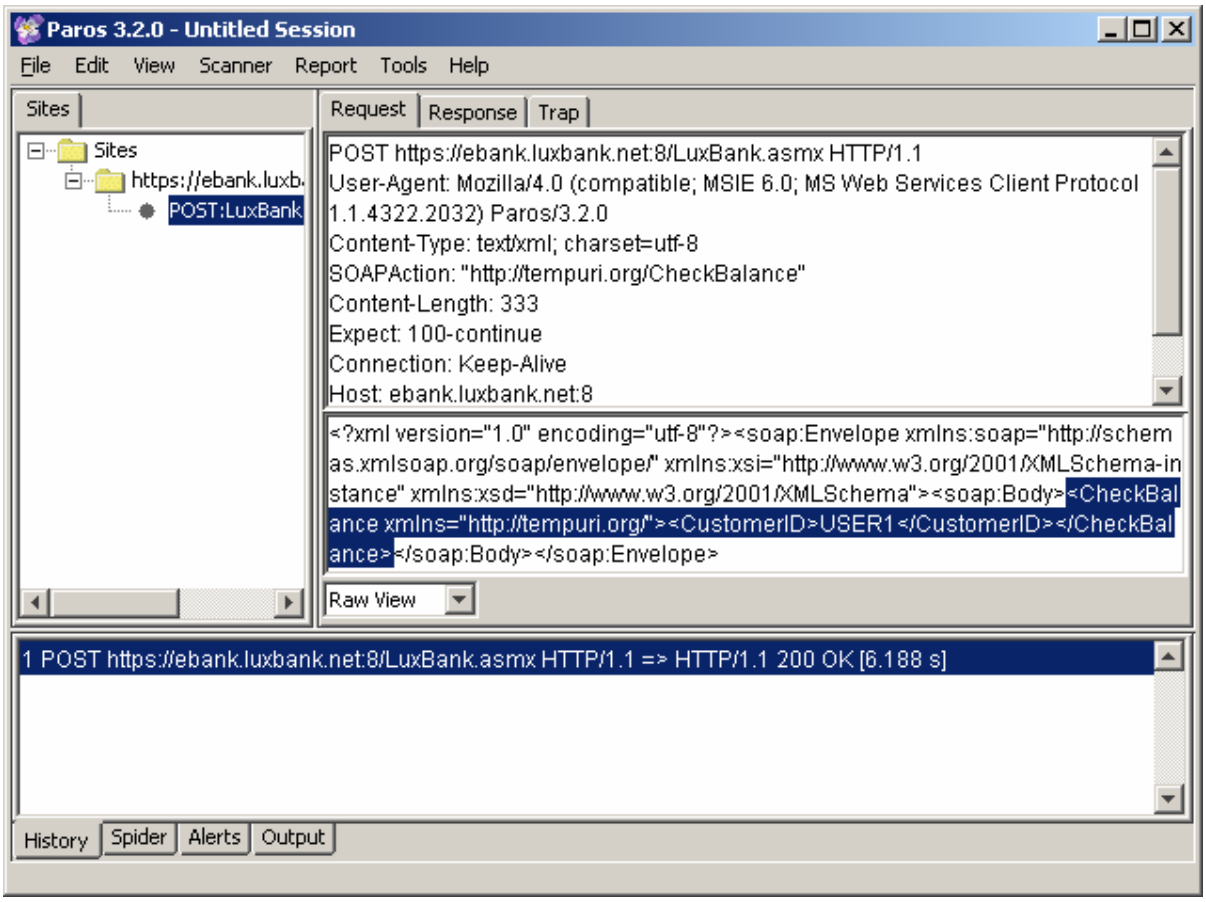
Once a web service has been identified we want to determine all the methods the web service offers. When dealing with client applications which utilise web services, the most obvious way would be to monitor communication between the web service consumer (client application) and the web service.

For demonstration purposes we'll look at a fictitious bank's (LuxBank) application, which queries a web service for a user's bank balance:



LuxBank demo application

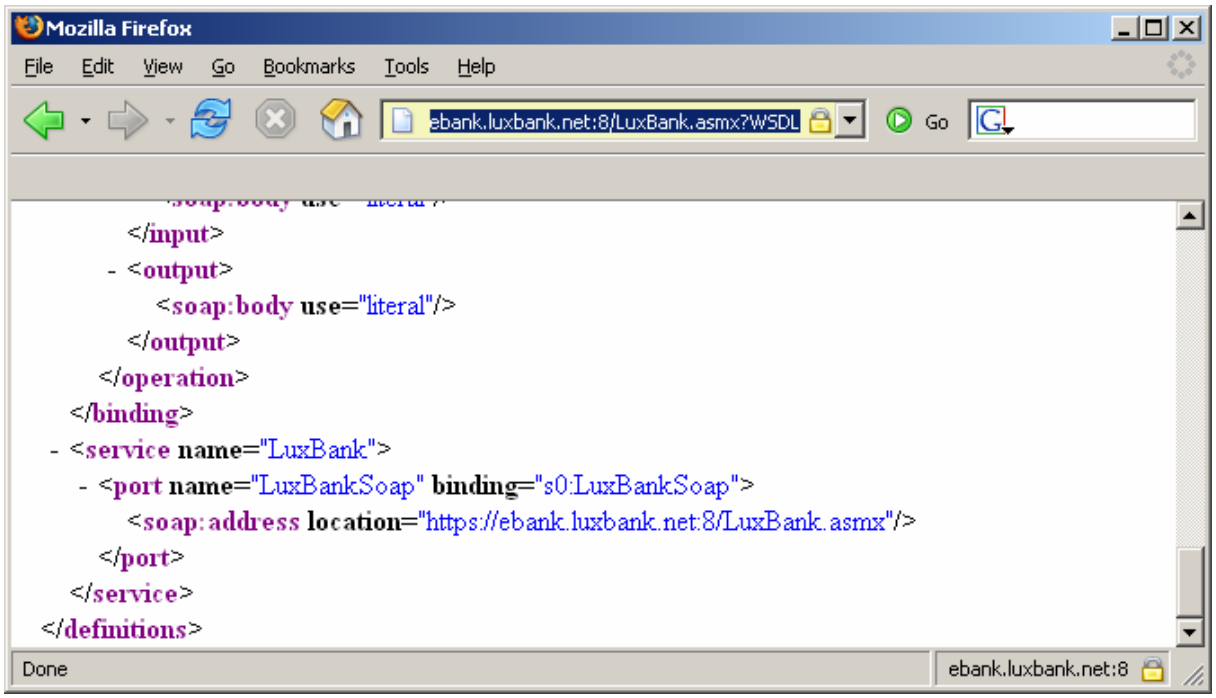
Except if system architects took very specific precautions in protecting the communications channel between the client and server, analysis of this communication is trivial. By evaluating our demo application, we see that the server is configured on an SSL (HTTPS) enabled and non-standard port (8/tcp). As stated earlier, the HTTP protocol is used to send requests and receive responses from web services. As our demo application was developed in Microsoft's .Net framework it uses an instance of Internet Explorer when sending requests to the web service. To capture and modify requests sent to the web service we use a web proxy (Paros in this case), and configure Internet Explorer to use the web proxy as a proxy server. Selecting *Balance* in the application user interface displays the user's account status and balance. The SOAP over HTTP request to the web service would appear as follows in the web proxy:



Evaluating web service requests through a web proxy

If we were to use this mechanism to determine all possible methods that can be invoked on the web service, we would have to "browse" the application, and record all requests. However, there are easier ways… the most obvious being WSDL.

WSDL (Web Services Description Language) is a definition of all the methods offered by a web service, as well as the associated input- and output parameters for these methods. As described earlier, we can identify the location of a WSDL for a web service through UDDI. Another method utilises the Google search engine. If crawled by Google, an advanced Google search query can help us identify the location of a web service's WSDL. Our search query, referred to as a Google hack, would be something similar to: *site:target_domain filetype:wsdl OR filetype:asmx OR filetype:jws*. Here we have to note that this resource will have to be linked to before Google will return it as a result. To retrieve the WSDL from an *.asmx (ASP.NET XML Web Service) or *.jws (Java Web Service) resource, append '*?WSDL'* to the end of the resource. This will display the WSDL for that web service, such as:

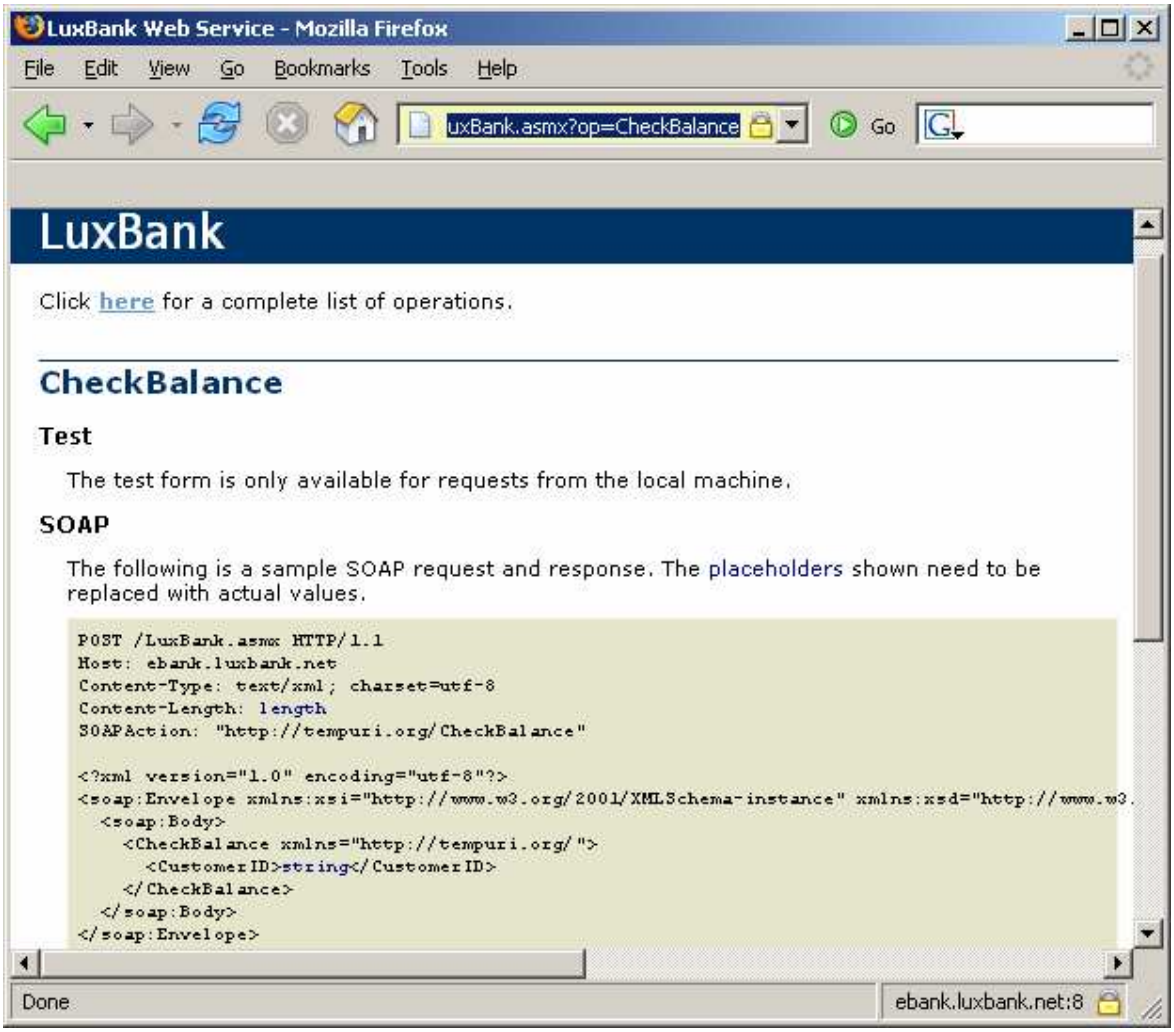


Viewing the WSDL through an ASP.Net XML Web Service

By taking into account that WSDL defines all the methods that can be invoked from a web service (in essence representing the attack surface of a web service), from a defensive position we have to ask ourselves if allowing attackers to gain access to a web service's WSDL is such a wise option. WSDL represents all the information an attacker requires to start attacking a web service, including the web service's location. It is highly recommended that organisations only provide its WSDL files to trusted users of its web service, such as for development purposes, or where the service is intended for public use and it is an acceptable risk to expose such information. In order to prevent the ASP.Net XML web service (*.asmx), for example, from leaking information, the following lines may be added to the web.config file:

```
<webServices>
    <protocols>
            <remove name=”Documentation”/>
    <protocols>
</webServices>
```

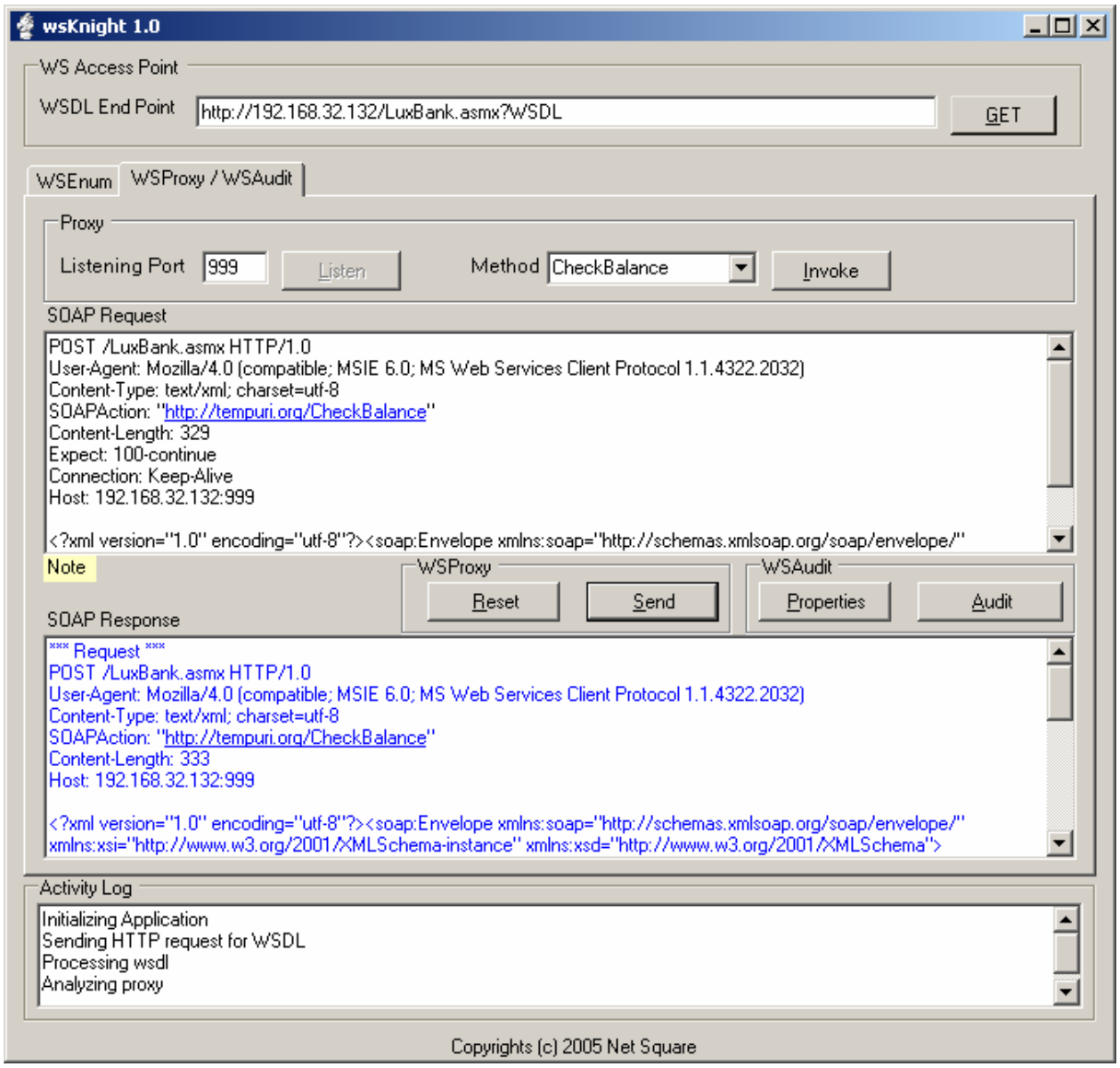
This will also prevent the attacker from viewing a listing of detailed information on the methods that can be invoked through a web service, such as by directly calling the *.asmx resource:



Viewing methods exposed by an ASP.Net XML Web Service

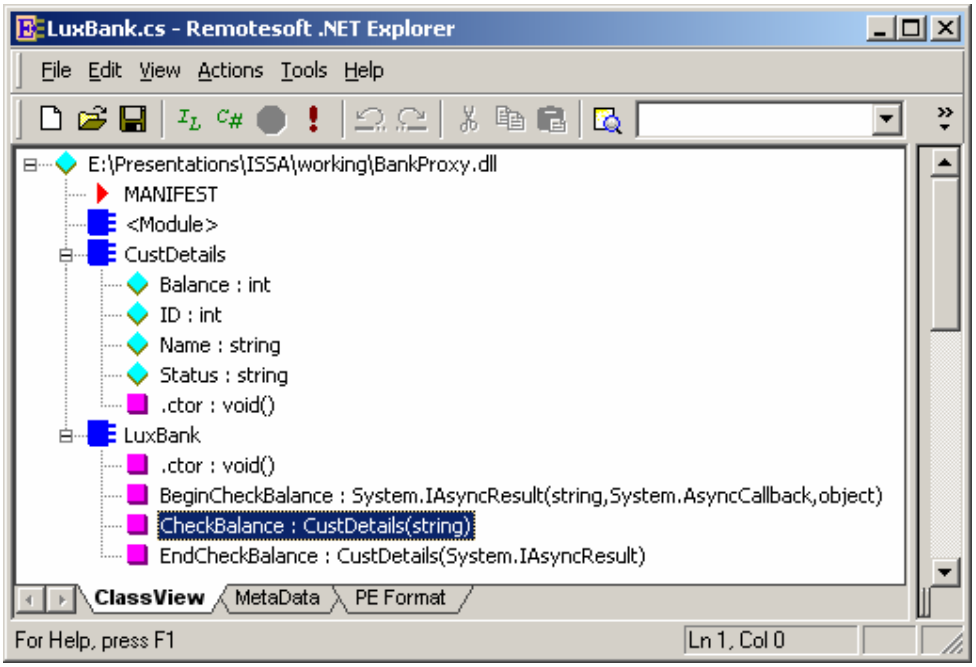
As could be seen in the above screen capture, we are provided with a sample of a SOAP request for the particular method selected (*CheckBalance* in this case). This is important from an assessment perspective as having easy access to the raw SOAP requests will make life a little easier in later stages of the assessment.

Apart from getting easy access to the SOAP requests as illustrated above, we can parse the WSDL file and construct our requests from there. There are various tools at our disposal to do this, including using Visual Studio .Net, adding a web reference pointing to the WSDL in our project, and writing the necessary code to call the available methods. There are tools that make this much easier though. An example is wsKnight, which is part of the wsChess toolset. wsKnight parses the WSDL file, enumerates the methods, and related inputs and outputs, and provides the functionality to generate valid SOAP requests.



Generating SOAP requests from WSDL with wsKnight

As was stated before, when deploying web services the methods exposed by the web service should not be freely made available except if deemed a functional requirement. However, even if the WSDL is not available, a client application can be inspected to determine which methods are on offer on the target web service. As our application was developed in Microsoft Visual Studio .Net, by examining the proxy dynamic link library (DLL) generated as part of our .Net web services project we can clearly identify the methods which may be invoked on the target web service:



Retrieving web services methods from proxy DLL

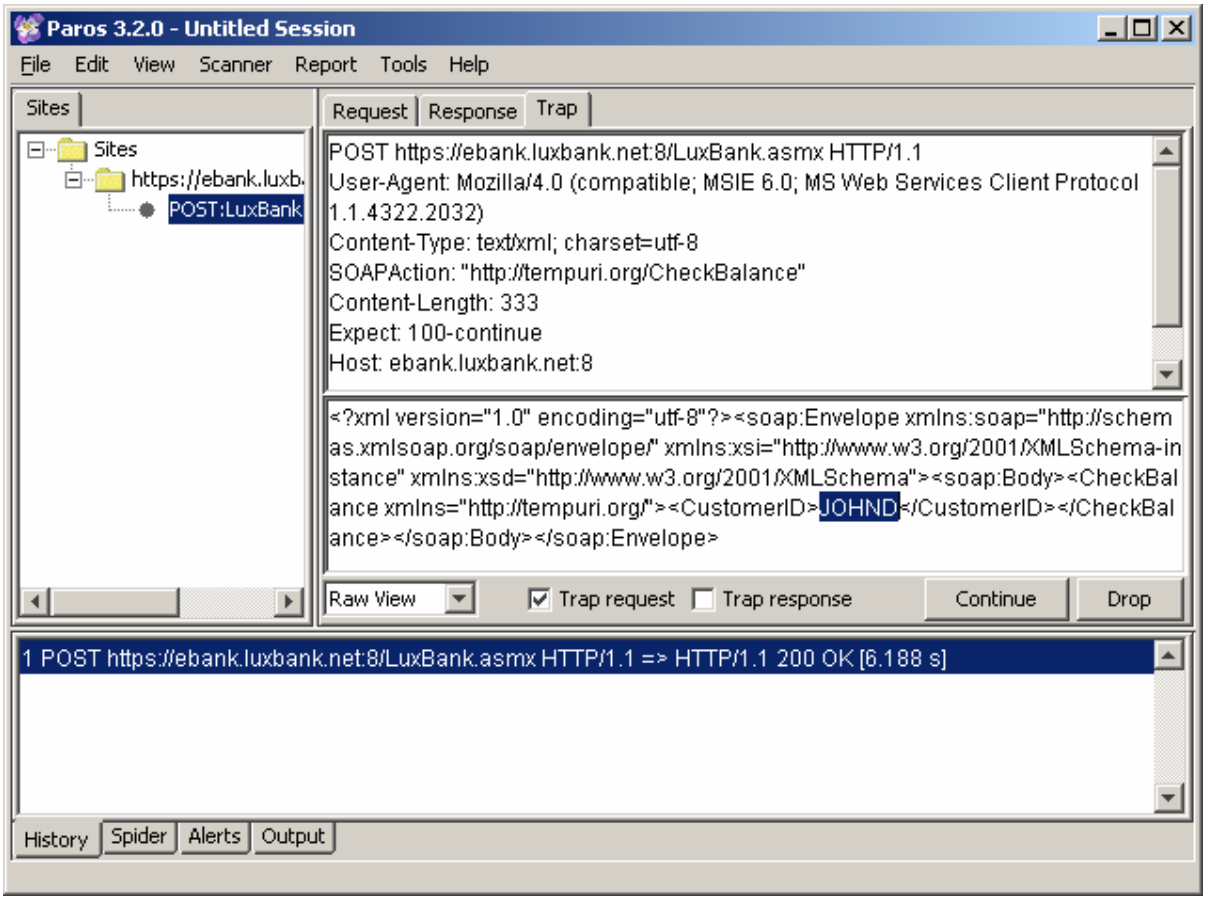
In order to protect against such information leakage, we can attempt to obfuscate the client executable. However, we need to keep in mind that obfuscating the client code will not be of much value, as the remote web service methods will still be named the same. These methods and parameters can therefore not be obscured without making the client unusable, as the method names and parameters are determined on the web service itself. Therefore, to make it more difficult for an attacker we should start by obfuscating the methods and parameters on the server-side first (*.asmx in our case), thereafter the WSDL is generated and the client built to use the obfuscated method and parameter names. It will however be extremely difficult for developers to deal with obfuscated method- and parameter names in code, and will most likely cause more of a diseconomy than do any good. Most likely the best way to achieve obfuscation will be to conduct the obfuscation process on both ends at the same time. In other words developers should run the obfuscator before releasing their code, which will have access to the source code of both the web service and the client application and obscure the method and parameter names on either side simultaneously.

Although the aforementioned precaution will raise the bar, by virtue of interacting with the application and inspecting the resulting web service request(s), as well as looking at the actual parameter values sent through, an attacker may still be able to figure out the necessary information to construct a valid web service request.

4 AUTHENTICATION AND AUTHORISATION

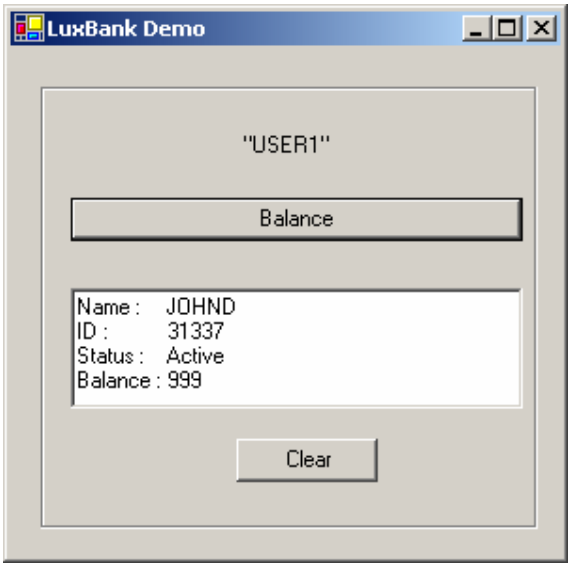
Once we have identified all the available methods, we can start looking at authentication and authorisation. Does the web service allow access to some or all of its methods without requiring authentication? Does it do so uniformly, and which authentication mechanisms does it employ? If a user is authenticated, are authorisation checks being performed before an action is allowed?

Let us look at a very simple example. In the earlier web service requests captured in our web proxy, the SOAP request called the *CheckBalance* method, with a *CustomerID* value passed as a parameter. The first attack on the web service would be to alter the *CustomerID* parameter sent to the web service:



Attacking the CustomerID parameter

So by merely changing the username to JOHND, we see the web service returning the account status and the bank balance of user John Doe:



Application displaying John Doe's bank balance

The developers have obviously made a grievous error by not properly authenticating the client before displaying the bank balance. Secondly, even if authentication is required it may be possible that the developers are not properly checking for authorisation. In other words, if the client is

authenticated as a user principal, does the web service restrict queries such as bank balance requests to that user principal only?

A process similar to the following should be followed:

1. Upon receiving a request, the web service should firstly authenticate the user. This may be done using various methods such as through session ID's being passed through in HTTP requests, using SSL client-side certificates, NTLM authentication or other methods. It should be clearly noted here that SSL will merely provide transmission security and will not address authentication, integrity and confidentiality of the actual message content. Mechanisms such as WS-Security will be discussed later.
2. Upon receiving authentication, the application should determine the user principal. Where passing session ID's, the web service will have to support a login method. Such a method will accept the user's credentials and pass a valid session ID, through a cookie for example, to the web services client. Subsequent attempts to access the web service will therefore involve first of all checking if the session ID is valid and secondly to which user principal the session ID is linked. With the other authentication methods mentioned, the user's identity will be passed through in some form or another in every request.
3. After identifying the user principal, it should be determined if the request is valid and authorised for that particular principal. So even if the user is authenticated, does the web service verify that the account actually belongs to the user principal before returning their bank balance? Developers often fail to uniformly and consistently implement authorisation frameworks, and we see both traditional web applications and web services becoming vulnerable to this attack vector.

If we take the above sequence into account, we see many areas that may lack security controls, possibly opening the door for an attacker. In assessing web services, as with assessment of web applications, we look at areas where authentication is not correctly or uniformly implemented. Developers should ensure an adequate mechanism is used to verify the identity of both the server- and the client-side. SSL configured on server-side, should be combined with the client application actually verifying the SSL server-side certificate to ensure it is talking to the correct web service. This will additionally make it difficult for someone to utilise a man-in-the-middle style traffic analysis, such as using a web proxy. Authentication of the client is particularly important and it should be ensured that an appropriate method is in use. In cases where session ID's are used to track a user's state, other aspects such as randomness and non-predictability of session ID's come into play. If developers do not make use of tried and tested mechanisms for session ID management it will be a particular point of interest for any attacker.

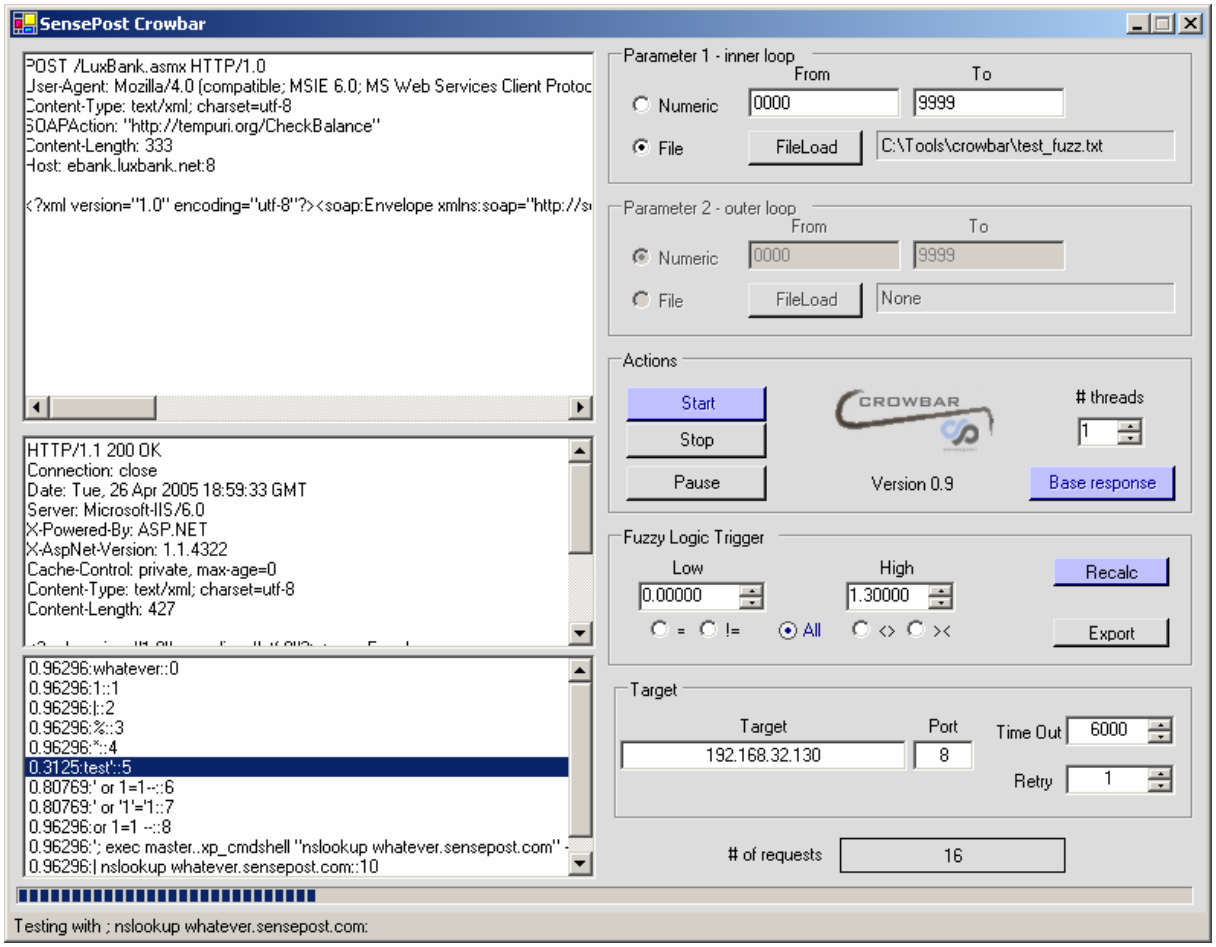
Web Services Security (WS-Security) can be employed, allowing SOAP messages to be digitally signed and/or encrypted. It will provide authentication, integrity and confidentiality both in message transmission and in storage, whereas SSL for example will only provide encryption between the transmission endpoints. It has to be carefully noted that problems such as in authorisation and input sanitisation (see next section) are not explicitly being addressed, and remains potential problem areas even in the light of using WS-Security.

Lastly, dependable and homogeneous authorisation checks should be performed. Many a time we find developers not implementing a robust solution or neglecting to include certain methods of the web service when performing authorisation checks, which may have dire consequences such as privilege escalation attacks succeeding.

5 INPUT SANITISATION

As with traditional web applications, web services may be vulnerable to input sanitisation problems. Input sanitisation refers to server-side processes such as web services not properly cleaning client supplied input. Depending on the functionality and underlying architecture of the web service, it may be vulnerable to attacks such as operating system command execution, SQL injection, LDAP injection or directory traversal. Explaining these attacks is entirely outside the scope of this paper, and it is assumed that the reader is familiar with the basic principals of these attacks.

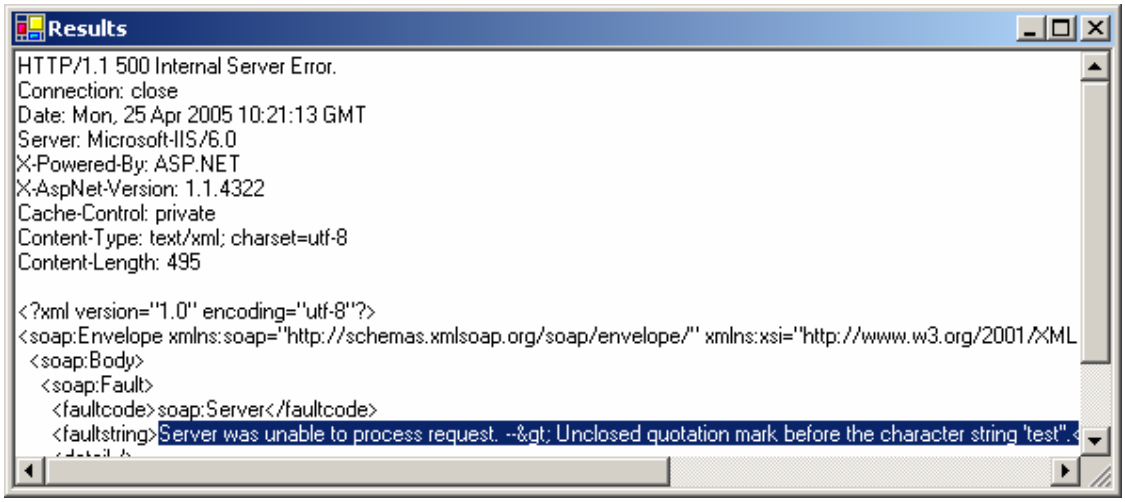
To test for input sanitisation problems in web services, we follow a similar approach than for traditional web applications. This involves passing potentially malicious characters to the web service and checking for telltale signs of the application responding in a way that is not expected, normally in the form of an error condition occurring. As we will most likely be passing various potentially malicious values, known as fuzz strings, to the web service, we can identify the need for automation. A SensePost developed tool called Crowbar (available from http://www.sensepost.com/research/crowbar/), even though not initially intended for this purpose, can be used to automate the fuzzing process, which refers to the process of sending a range of potentially malicious values to the web service. Configuring Crowbar is fairly simple. By copying a valid web service request from our web proxy (Paros) and pasting it into Crowbar, we can start the input sanitisation test:



Automated input sanitisation testing with Crowbar

As was stated before, we have to keep in mind that the web service is configured to use SSL. We therefore have to use an SSL proxy to convert normal HTTP traffic from Crowbar to SSL.

First off, the Crowbar utility sends a valid request and receives a response from the web service. The request is indicated in the top left hand pane, and the response in the pane just below it (middle pane). This is referred to as the base response. We now select the point within the request where we want to insert the fuzz strings, which in this case is the *CustomerID* parameter. Crowbar will now insert the strings found on each line of the fuzz file into the request at the specified insertion point. The request is sent to the web service and upon receiving the response Crowbar does a content comparison between the current response and that of the base response. This allows us to easily identify responses different to that of the base response, which may indicate potential problems. If we look at the content compare weights for fuzz string test' (weight 0.3125) we see that it is significantly different to that of fuzz string whatever (weight 0.96296). The response we received when sending the request with *CustomerID* parameter of test' appears as follows:



Error response for malicious input

We see the web service responding with an error message that reads: "Unclosed quotation mark before the character string 'test'"'. This is a typical ODBC error message which indicates the backend database query breaking successfully. Even if proper error handling was in place and the message cleaned before being sent to the client-side, the HTTP 500 error message would indicate the application breaking on being supplied the single quote character (‘). Anyone with a basic understanding of SQL insertion would be able to identify this problem even where verbose error messages lack (known as blindfolded SQL injection attacks). The process of exploiting this vulnerability is exactly the same as for web applications. We can now start to manipulate the backend database query to our heart's content, which depending on the backend database being used and application logic may include viewing, modifying or deleting information of other users or information in other tables, or making use of extended functionality provided by the database to for example execute operating system commands or perform other malicious actions.

Depending on system architecture, we may not only be restricted to SQL injection attacks and various other problems may be relevant such as directory traversal, canonicalisation issues, LDAP injection and more.

In order to properly protect against input sanitisation problems, web service developers should ensure all parameters accepted as input is properly sanitised. Whitelisting is the best method to achieve this. It means all data must be cleaned and all characters not expected for that specific data type must dropped. Data types in this instance refer to for example usernames, identity numbers and email addresses. On top of the aforementioned, it should be ensured that proper character escaping / encoding is being performed. For example single quotes (‘) in variables should be escaped before being passed to the backend database.

6 CONCLUSION

In order to adequately secure web services, we need to apply a few simple principles, and proper enforcement of these rules will largely depend on organisational policy and cultures. As a new frontier for attackers, system architects should realise the need for not only protecting infrastructure hosting web services, but also ensuring web services are architected and coded in a secure fashion. This document in no way represents a finite list of security controls to be implemented. It merely represents a typical approach a security analyst or attacker will follow when assessing web services, and the high-level areas we need to secure to protect against these attacks.

7 REFERENCES

W3C Recommendation. SOAP Version 1.2. 24 June 2003. Available from: http://www.w3.org/TR/soap12/.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. 15 March 2001. Available from: http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

UDDI Committee Specification. 19 July 2002. UDDI Version 2.04 API Specification. Available from: http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.

Anthony Nadalin, Chris Kaler, Phillip Hallam-Baker and Ronald Monzillo. Web Services Security: SOAP Message Security 1.0 (WS-Security 2004). March 2004. Available from: http://www.oasis-open.org/specs/index.php#wssv1.0.

Shreeraj Shah. Web Services – Attacks and Defense, Information Gathering Methods: Footprints, Discovery & Fingerprints. Available from: http://net-square.com/wschess/ WebServices_Info_Gathering.pdf.