WEB SERVICES VULNERABILITIES

A white paper outlining the application-level threats to web services

Date: February 15, 2007

Prepared By: Nishchal Bhalla
Sahba Kazerooni

Abstract

Security has become the limiting factor in the broad adoption of web services. As a result, much emphasis has been placed on the development of various high-level security standards and protocols, but in many cases the simplest application level attacks have been neglected.
This paper will explore, at a low-level, the vulnerabilities inherent to web services from an attacker's point of view. The paper covers the dependency of web services on XML, the various forms of XML-based attacks, including exploiting parsers and validators, and finally provides recommendations and countermeasures.
This paper is intended for developers and web application architects. It drills down to the details of web services implementation, while maintaining a focus on good versus bad architectural design.

Table of Contents

INTRODUCTION

The concept of service-oriented architecture (SOA) has revolutionized the distribution of applications by introducing web services that may be called both internally and publicly.

Web services have gained popularity for many reasons:

- **Interoperability:** Services can be built on any framework with any language, independent of any programming model
- **Maintenance:** SOA separates functional modules, thus centralizing maintenance efforts
- **Reuse**: SOA encourages code reuse among different applications by providing a 'service' to be used by all
- **Smart pipe**: Perhaps the most desired feature of SOA is the introduction of a 'smart pipe'; a chain of intermediary services which add value to a single call.

As a result of the 'smart pipe' concept, the latest research has been focused on the development of standards[1] to facilitate interaction of intermediaries through message-level security, federation of web services, and content and policy communication. The security posture of many web services, however, remains in a dismal state, mainly due to the various application-level vulnerabilities.

Web services are often the wrapped around backend systems that have traditionally been protected by multiple layers. Security is often handled by these layers, which once removed, leave the module exposed to the web and vulnerable to various attacks on the application logic. Furthermore, the introduction of XML as a communication protocol opens web services to a world of XML-based attacks. This paper will explore, at a low level, those attack vectors inherent to web services form an attacker's point of view.

FALSESECURE

For the purposes of demonstration, we will be using FalseSecure, a sample application with the architectural layout represented in Figure 1. The application consists of two logical layers, the FalseSecure Client and the FalseSecure Server. Previously, the two layers were part of a three-tier web-based architecture, but with the need to provide their services to business partners and other vendors, FalseSecure has extracted their business logic into a web service. Now, the sole purpose of the FalseSecure Client is to provide a user interface, consisting of Java Server Pages (JSPs), which are used to convert user input into calls to the FalseSecure web service.

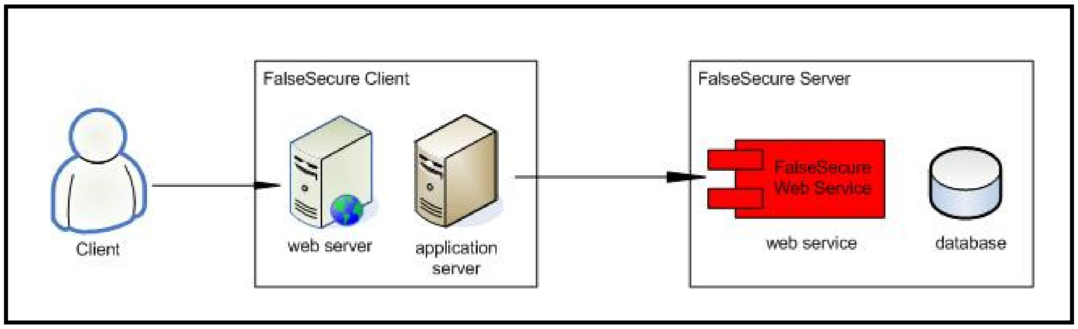

Figure 1: FalseSecure high-level architecture

FalseSecure is an e-commerce application. The user will be able to request a quote for an item, place an order for an item, or place a bulk order using an XML form. As such, the FalseSecure web service provides the following services:

- Authentication
- Checking the inventory of a product
- Requesting a list of products
- Requesting a price quote for an order
- Placing an order
- Placing an XML bulk order
- Processing a credit card transaction
- Logging off

The remainder of this paper will be focused on threats to the FalseSecure Web Service.

THE ATTACKS

Most web services communicate over HTTP and are essentially still web applications. They suffer from the same vulnerabilities as their presentation-oriented counterparts. The top web application security vulnerabilities, like those outlined in the OWASP top 10, still applies to web services. The goal of this paper, however, is not to discuss ALL these vulnerabilities but to outline the attack vectors unique to service oriented implementations.

In web services frameworks, XML documents are passed from client to server, in the form of a SOAP request. XML is then processed within the web service, opening it to an array of XML-based attacks. These attacks, among others, will be the focus of this section.

WSDL Enumeration

As with any other web application attack, the first step in formulating an attack on a web service is discovery and fingerprinting of the service deployment. The information gathered at this first stage is crucial to identifying possible entry points and carrying out an attack that is directed at the heart of the web service.

Luckily for today's hackers, web services are often deployed with what many refer to as a 'blueprint for attack'. The WSDL file is a web services deployment descriptor that outlines not only the functionality provided by the web service, but also the expected syntax, the input and output points, and the location to access the service. In other words, the web service is announcing to the world its location, the methods it provides and assumptions it is making regarding its input points – a goldmine for anyone with malicious intent.

Looking at FalseSecure's blueprint, a few items stand out to the trained eye:

1. *Login* method requires a username and password as input, so we can be quite certain that it is used for authentication into the web service. If we wish to bypass authentication, we could attempt to break this method, either by brute forcing, SQL injection or another method as discussed later.
2. SessionId is a parameter for all methods in this application, so perhaps we can bypass authentication by attacking the session management using brute force or session hijacking.
3. The *service* tag tells us the location of the web service. We will use this to access the web service directly, thus bypassing the client-side validation provided by the JSPs.

```
 <?XML version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="www.falsesecure.com"
XMLns:apachesoap="http://XML.apache.org/XML-soap"
XMLns:impl="www.falsesecure.com" XMLns:intf="www.falsesecure.com"
XMLns:soapenc="http://schemas.XMLsoap.org/soap/encoding/"
XMLns:wsdl="http://schemas.XMLsoap.org/wsdl/"
XMLns:wsdlsoap="http://schemas.XMLsoap.org/wsdl/soap/"
XMLns:xsd="http://www.w3.org/2001/XMLSchema">
+ <wsdl:types>
+ <wsdl:message name="placeXMLOrderRequest">
+ <wsdl:message name="placeOrderRequest">
+ <wsdl:message name="processCreditCardTransactionRequest">
+ <wsdl:message name="loginResponse">
+ <wsdl:message name="getQuoteRequest">
+ <wsdl:message name="getProductListRequest">
+ <wsdl:message name="loginRequest">
+ <wsdl:message name="getQuoteResponse">
+ <wsdl:message name="getProductListResponse">
+ <wsdl:message name="placeXMLOrderResponse">
+ <wsdl:message name="placeOrderResponse">
+ <wsdl:message name="processCreditCardTransactionResponse">
- <wsdl:portType name="FalseSecureIF">
+ <wsdl:operation name="getProductList" parameterOrder="session_id">
+ <wsdl:operation name="getQuote" parameterOrder="prod_id quantity
session_id">
+ <wsdl:operation name="placeOrder" parameterOrder="prod_id quantity
credit_card_num credit_card_date session_id">
+ <wsdl:operation name="login" parameterOrder="cust_id password">
+ <wsdl:operation name="placeXMLOrder" parameterOrder="XML">
+ <wsdl:operation name="processCreditCardTransaction" parameterOrder="XML
session_id">
  </wsdl:portType>
- <wsdl:binding name="FalseSecureIFPortSoapBinding"
type="impl:FalseSecureIF">
  <wsdlsoap:binding style="rpc"
transport="http://schemas.XMLsoap.org/soap/http" />
+ <wsdl:operation name="getProductList">
+ <wsdl:operation name="getQuote">
+ <wsdl:operation name="placeOrder">
+ <wsdl:operation name="login">
+ <wsdl:operation name="placeXMLOrder">
+ <wsdl:operation name="processCreditCardTransaction">
  </wsdl:binding>
- <wsdl:service name="FalseSecure">
- <wsdl:port binding="impl:FalseSecureIFPortSoapBinding"
name="FalseSecureIFPort">
  <wsdlsoap:address
location="http://192.168.0.100:8080/FalseSecure/services/FalseSecureIFPort"
/>
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

Figure 1: WSDL file of our FalseSecure web service

Most web services implementations are made publicly accessible to allow its use by partners and other front-end portals. As such, the WSDL file is also exposed to the public. An attacker can use one of the following methods to retrieve the file:

- Appending *?WSDL* or *.WSDL* (depending on the platform) to the end of the service URL more often than not will reveal the WSDL file. In our example, the we would use the following URL

  http://localhost:8080/FalseSecure/FalseSecureIFPort?WSDL

- Search engines will typically index WSDL files along with other files within a domain. Using cleverly crafted search engine queries, attackers can find these files, a method commonly referred to as Google Hacking [2]. Some possible queries include but are not limited to:

  - filetype:wsdl
  - indexof "/wsdl"
  - inurl:wsdl
  - inurl:asmx (note that asmx is the WSDL equivalent in ASP.Net [3])

- Universal Description, Discovery and Integration (UDDI) servers provide a centralized repository of web services and their WSDL files. Service providers often post their details on public UDDI's to be discovered by users at runtime. The key point to remember while using UDDI's is that they are publicly accessible and provide an easy way for a hacker to identify and locate your web service as well.

Defense

1. Avoid publishing the WSDL on UDDI or consider locking down access to the UDDI using the access control features built into UDDI version 3.0.2 or later [4]
2. Use robots.txt file to stop Google from indexing the directories that host WSDL's
3. Disable documentation of WSDL at container:
   .NET - Remove the following from web.config:
   ```
   <webServices>
       <protocols>
           <remove name="Documentation"/>
       </protocols>
   </webServices>
   ```
   J2EE - Remove the following from <APPLICATION>.properties file:
   ```
   wsdl.location=/WEB-INF/<APPLICATION>.wsdl
   ```

Exploiting XML Parsers

XML streams are parsed at some point within the application logic, and you can be certain that an attacker will attempt at breaking the web service or its assumptions within the parsing logic. Two types of parsers are typically used, SAX and DOM.

DOM based parsers load the entire XML stream into memory, creating a hierarchical object that is referenced within the application logic. A very obvious attack vector is inputting large XML files to consume server-side resources during parsing, resulting in a **Denial of Service** attack. The method *ProcessCreditCardTransaction* of our FalseSecure web service receives as input an XML stream of the transaction to be processed.

An attacker may attempt a DoS attack by inputting a large file, using repeated nodes:

```
<transaction>
       <total>1000.00<total>
       <credit_card_number>123456789</credit_card_number>
       <credit_card_number>123456789</credit_card_number>
       <credit_card_number>123456789</credit_card_number>
       <credit_card_number>123456789</credit_card_number>
       …
       …
       …
       <expiration>01012008</expiration>
</transaction>
```

Figure 2: Attacking DOM parser with large payload

An attacker may attempt a DoS attack by using recursion of XML elements, resulting in even more processing overhead:

```
<transaction>
       <total>1000.00<total>
       <credit_card_number>
              <credit_card_number>
                     <credit_card_number>
                            <credit_card_number>
                                   …
                                          …
                                                 …
       <expiration>01012008</expiration>
</transaction>
```

Figure 3: Attacking DOM parser with recursive payload

SAX-based parsers are not susceptible to the Denial of Service attacks discussed above. The difference is that SAX-based parsers are event-driven, meaning they parse the XML steam as needed, thus holding a maximum of two elements in memory at any given time. They are, however, subject to another form of XML-based attack, commonly referred to as **XML-injection**.

In an XML-injection attack, a user will attempt to 'spoil' the XML stream by inputting data that will overwrite the static portions of the stream. In the case of FalseSecure, a user may enter credit card number information in such a way that spoils the XML stream that is passed to the *processCreditCardTransaction* method.

Figure 4 below shows a sample payload to the transaction function, within which the user input (in bold) has overwritten the <total> element. The user has just fooled the parser into believing the total amount to be charged for this plasma TV is actually $6.66. The original $4,000 was calculated using the *getQuote* method, and the result was combined with un-validated user input into an XML stream.

```
<transaction>
       <total>4000.00<total>
       <credit_card_number>123456789</credit_card_number>
       <total>6.66</total><credit_card_number>123456789
       </credit_card_number>
       <expiration>01012008</expiration>
</transaction>
```

Figure 4: XML-Injection, spoiling XML stream (user input is bolded)

Contrary to popular belief, DOM-based parsers can also be susceptible to XML-injection. Consider the following code snippet that parses this same XML stream using a DOM-based parser. The final value of transaction.total will be 6.66 due to the nature of the parsing logic.

```
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(new StringBufferInputStream(XML));

for (int i=0; i < nl.getLength(); i++)
{
       Node node = nl.item(i);
       String node_name = node.getNodeName();
       if (node_name.equals(“total”){
             transaction.setOrderTotal(
                    Double.parseDouble(node.getTextContent()));
       } else if (node_name.equals(“credit_card_number”)){
             transaction.setCreditcardnum(
                    node.getTextContent());
       } else if (node_name.equals(“expiration”)){
             Transaction.setCreditcardexpiry(
                    node.getTextContent());
       }
}

TransactionProcessor.process(transaction);
```

Figure 5: Sample DOM-based parsing code that is also susceptible to XML-injection

<u>Defense</u>

1. Do not implement custom parsers, as you will most likely forget about certain attack vectors. Use available parsers that have been tried, tested and secured!
2. Use SAX-based parsing whenever possible to defend against Denial of Service attacks
3. If using DOM-based parser, validate the XML stream size against a maximum before parsing
4. XML-Injection is due to a lack of input validation, so Validate! Validate! Validate!

Exploiting XML Validators

XML steams are normally validated against certain rules before being used by the application. This is to ensure the data is complete and the assumptions the application makes about the data are met. Thus, an attacker may attempt to break or bypass the validation, resulting in unexpected input to the application logic.
Document Type Definition (DTD) is a common validation method that should be avoided at all costs. DTD’s may be defined:

a) Internally: Within the XML Document <!DOCTYPE TRANSACTION […]>
b) Externally: Reference an external file <!DOCTYPE TRANSACTION SYSTEM “file”>
c) Both: <!DOCTYPE TRANSACTION SYSTEM “file” […]>

In either case, the <!DOCTYPE > tag exists within the XML document. Web services that expect all XML input to be accompanied with a DTD will be hit with the reality that some clients have malicious intent and will modify the internal DTD to match their input or simply reference an external file.

A similar attack, referred to as a **DTD Entity Reference Attack** uses the concept of entities within a DTD to reveal sensitive data residing on the server hosting the web service. Consider the FalseSecure *placeXMLOrder* service which takes as input an XML order. The XML order in figure 6 would result in the following output:

> "Thank you Bob Smith. Your order has been placed successfully."

```
<!DOCTYPE order SYSTEM "c:\order.dtd">

<order>
       <cust_id>1</cust_id>
       <cust_name>Bob Smith</cust_name>
       <creditcardnum>999999999999</creditcardnum>
       <creditcardexpiry>0609</creditcardexpiry>
       <item>
              <prod_id>1</prod_id>
              <quantity>2</quantity>
       </item>
</order>
```

Figure 6: a valid XML order, complete with a valid dtd

A malicious user who notices the echo of his name in the response may manipulate the DTD to echo something more useful. The XML order in figure 7 would result in the following output:

> "Thank you [contents of c:\boot.ini on the remote server]. Your order has been placed successfully."

```
<!DOCTYPE order
[
       <!ELEMENT order ANY>
       <!ELEMENT cust_name ANY>
       <!ENTITY name SYSTEM "c:\boot.ini">
]
>

<order>
       <cust_name>&name;</cust_name>
</order>
```

Figure 7: DTD Entity Reference attack.

<u>Defense</u>

1. Use Xml Schema Definition (XSD)[5] to validate XML streams. Among other advantages such as higher granularity of definitions, XSD's are not defined within the XML document itself and thus cannot be changed by the end user.
2. If you must use DTD's, do not allow the client to supply the prologue portion of the XML document (e.g. the section before the root element). Also ensure you are validating all input for malicious characters before XML parsing or validation.

Error Handling

Error messages are a goldmine in the eyes of an attacker, as they often reveal sensitive information regarding the internal state of an application. This insight into the internal state can give the attacker enough information to execute an attack directed at specific application logic. A successful attack on a web service will also typically be preceded with error message gathering.

With web services, uncaught exceptions within application logic are caught at the soap engine and displayed as a SOAP fault element within the response to the client. For example, the following message is in response to a single quote (a common SQL injection vulnerability test) as the password to the FalseSecure *login* method:

```
<soapenv:Fault
   XMLns:soapenv=http://schemas.XMLsoap.org/soap/envelope/>
       <faultcode>soapenv:Server.userException</faultcode>
       <faultstring>java.sql.SQLException : you have an error in your
SQL syntax; check the manual that corresponds to your MySQL server
version for the right xyntax to use near "'" at line 1</faultstring>
       <detail>
              <ns1:hostname
XMLns:ns1= »http://XML.apache.org/axis/ »>web</ns1:hostname>
       </detail>
</soapenv:Fault>
```

Figure 8: SOAP fault revealing a SQL injection vulnerability

Web services generally tend to reveal more sensitive information than regular web applications. Many web services are wrapped around backend systems that have been opened to the public, and thus they lack the error handling that was at one point implemented at the surrounding layers. A common code review practice is to start from points of entry into an application, namely server pages, and to ensure any method that may throw an exception is surrounded by error handling logic so that only generic messages are revealed to the end user. In these instances when you remove the server page you remove the error handling along with it.

Defense

1. Ensure all exceptions are caught and generic error messages returned with the SOAP response
2. Some SOAP engines are configurable enough to suppress exception details from being included in the fault element. Within Axis, for example, the parameter **axis.development.system** in the **server-config.wsdd** file dictates whether or not stacktraces are stripped out of the fault string before being sent to the client. This value should be set to *false* for a production system.

XPath Injection

XML documents are often treated as data stores and queried by web services that consume and process them. As SQL is used to query traditional relational databases, XPath is used to query XML documents. XPath [6], like SQL, is susceptible to injection which can lead to arbitrary execution of queries on the server.

XPath queries that are dynamically combined with user input may be modified by the user in a way the developer did not intend. Consider the following XPath query that may be used in the *Login* method of our FalseSecure web service, where the bolded values are user input:

//users/custid[**123**]

A user may attempt to bypass authentication by modifying the XPath statement to return all users who are greater than one year of age:

//users/custid[**./age > 1**]

Defense

1. XPath injection is due to a lack of validation on user input. Validate, validate, validate!

CONCLUSION

Security standards such as WS-Security are crucial to the secure implementation of web services. However, a system is as secure as its weakest link, and in most web services implementations, that weakest link is the application logic.

Aside from the top application-level vulnerabilities published in the OWASP guide, web services are also susceptible to the various XML-based attacks discussed in this paper. Furthermore, the public disclosure of the WSDL file makes an attacker's job that much easier by providing a blueprint of the system. Reducing the amount of information disclosure and secure coding practices to defend against the common web application and XML-based attacks, combined with the latest security standards will improve the security posture of systems as we move towards a more service-oriented architecture.

ABOUT SECURITY COMPASS

Security Compass is a leader in application security consulting and training. Our practitioners have performed engagements and taught courses around the world in all security-sensitive industries, from financial institutions and defense to technology, healthcare, telecomm and others.

Our consultants are experts in the field of application security. We have also spoken at conferences around the world, including Reverse Engineering Conference 2005 in Montreal; HackInTheBox 2005 in Malaysia; Infosec Conference in Las Vegas, New York, Toronto and DC; CSI NetSec; and DallasCon. We have also literally written the book on the subject, with texts such as Buffer Overflow Attacks, Hack Notes - Network Security, Hacking Exposed - Web Applications, Windows XP Professional Security, and Writing Security Tools and Exploits. Our consultants are actively involved in the security community, we lead the Toronto OWASP (Open Web Application Security Project) Chapter, contributed to tools such as YASSP and also have released free tools to audit web application source code.

ADDITIONAL RESOURCES

1. Web Services WS-Standards - www.ws-standards.com
2. Johnny Long’s Google Hacking Database - http://johnny.ihackstuff.com
3. Web Services with ASP.NET - http://msdn2.microsoft.com/en-us/library/ms972326.aspx
4. Access Control with UDDI Version 3.0.2 - http://uddi.org/pubs/uddi_v3.htm#_Toc85908057
5. W3C XML Schema Reference - www.w3.org/XML/Schema
6. W3C XPath Specification - www.w3.org/TR/xpath