Demystifying Webservices: How they really work

More than any recent technology, Web services are surrounded by hype, mystery, and a bit of mumbo jumbo thrown in for good measure. Does anyone really know what they are or how they work? To give you a leg up, in this column I'll demystify Web services for you and explain how Web services protocols do their work.

Let's start with the basics. Web services are modular software components wrapped inside a specific set of Internet communications protocols and that can be run over the Internet. These components can communicate with other components automatically without human intervention. They can be used on an Intranet inside a firewall, or out across the greater Internet. A Web service itself is a software module delivered over the Internet or an intranet via XML (eXtensible Markup Language) messaging. The software module can be built in a variety of ways, most notably, but not exclusively, using Java.

At the heart of the Web services architecture is the need for program-to-program communications. And in order for that communication to take place, the Web service itself first needs to be described in detail so that other programs can understand what it is and know how to connect to it. That is what XML does - it describes the service in a manner that can be understood and used. This XML depiction is called a service description, and includes all the details necessary for the Web service to be accessed, including its location, transport protocols and message formats it uses.

Understanding the key roles in the Web services architecture

In order for a computer or program to use a Web service, it needs be able to find the service description and then bind to it. To accomplish this, there are three key roles in Web services architecture: a service provider, a service registry and a service requestor. Together, they perform three operations on a Web service: publish, find and bind. The nearby figure (Fig. 1) shows how this all fits together.

![Fig. 1](image)

The publish operation makes information about the service available so that it can be found and used - in other words, it makes the service description publicly available. The
find operation discovers the Web service - it's the way in which the computer or program searches for, and understands, what the Web service is, where it's located, and how to link to it. The bind operation allows the service to be used by the person or program requesting the service.

Let's take a look at a typical scenario detailing how the service provider, service registry and service requestor work together to deliver a Web service. First the Web service is built as a software module, then a service description is created for it using XML. A service provider hosts the module. The provider also hosts the XML service description for the Web service, which includes details about the service, including its location, transport protocols and message formats it uses.

The service provider publishes this service description to a service registry, a public, searchable index of service descriptions through which people can find Web services. Included is information about the Web service, such as details about the service provider/host. The service registry's role is to make available service descriptions so that Web services can be found and run. A service registry isn't absolutely required for Web services to be run - service descriptions can be found in other ways, such as from an ftp site, a Web site, a local file, or from other sources.

The service requestor is the business looking to run a Web service, or an application looking to interact with a Web service. It can be a person using a Web browser, or can also be a program, or even another Web service. The service requestor searches the service registry and finds the service description for the Web service. Based on the information it finds in the service registry, it connects to the service provider hosting the Web service using a bind operation, and then runs the service.

A look at the underlying protocols
All of this is made possible by the basic building blocks of Web services, a group of three standards: Simple Object Access Protocol (SOAP); Web Services Description Language (WSDL); and Universal Description, Discovery and Integration (UDDI). Here's how they work together:

- **WSDL** is the language used to create service descriptions. It is able to create descriptions not only about the location of the service and how to run it, but also higher-level information, such as what business is hosting the service, the kind of service it is, keywords associated with the service and similar information.
- **SOAP** is the means through which the service provider, service registry and service requestor communicate. It's an XML-based technology used to exchange structured data between network applications. SOAP is used to publish the service description to a service registry. Similarly, all other interactions between service registry, service requestor and service provider are done via SOAP.
- **UDDI** is the directory technology used by registries that contain the description of Web services and that allows the directory to be searched for a particular Web service. UDDI is in essence a Yellow Pages that can be used to locate Web services. There can be both private and public UDDI directories.
That, in a nutshell, is how Web services work. In future columns, we'll examine the architecture and each of the protocols in more detail.

Web Services Threat Profile

Threats have evolved with distributed architectures from monolithic mainframes to two- and three-tier client server and on to n-tier Web environments. Web services introduce the concept of an n-peer architecture where components participate in a collective manner. Three basic characteristics of Web Services create both its functional power and also risk:

- **Standards** provide common methods and processes but also create an opportunity for an attacker to broaden his number of targets. As standards move ‘up the stack’ this reach increases drastically and the impact is felt more.

- **Loosely-coupled components** create a flexible, ‘plug-and-play’ architecture with replaceable pieces that foster scalability. The communications among these components provide new risks.

- **Federation** of sources for data can eliminate redundancy and add to the flexibility and scalability value proposition. But this federation also assumes much about the quality of the data and the inherent trust built into the environment.

Web Services Threats

A threat profile involves evaluating the components of an architecture and identifying likely avenues of attack. As mentioned earlier, the component architecture of Web Services increases the number of touch points that can be attacked. Figure 1 shows a diagram of many of these touch points.

Every threat needs an actor, input, and a target, with a focus on the latter two (actors are assumed). With Web Services, those three points are the attacker (consumer/source), XML document (inputs) and the target (vulnerable component).

Attacking and Defending Web Services

Vulnerability Classes

A specific review of the Web Services architecture provides some obvious attack points using traditional techniques. These vulnerabilities can affect both inputs and targets. What follows are descriptions of vulnerability classes based on weaknesses in inputs (in the case of XML/SOAP manipulation, protocol abuse, and untrusted configuration data) and targets (for legacy bolt-ons and untrusted entities).

XML/SOAP Manipulation

XML is the grammar and SOAP is the standard interface language of Web Services. New implementations, especially when pervasive across applications and entities, are prime targets for attackers.

XML documents are intelligent pieces of information. They may contain various types of data for input into a system. Some of the functional uses are described below:
SOAP Headers provides a pre-defined structure with an XML message for context-sensitive information including security tokens (e.g. SAML) as well as other volatile information intended for intermediary or end-point processing.

Protocol requests/responses provide the underlying communication mechanisms that programs understand.

Program instructions and variables can be passed as the content of XML elements.

Uniform Resource Indicators (URIs) are pointers to the source of other types of data or information.

Data input provides transactional data to a program.

Embedded code can insert data in other formats to support legacy systems or specialized formats.

It is clear that XML messages themselves can be the target of an attack or contain specific data elements that require targeted filtering for out of the norm signatures.

Protocol Abuse

Protocol abuse involves a subset of the overall XML/SOAP infrastructure. Web Services has more higher-level protocols than any previous technology. Each of these protocols provides a set of rules that can be bent, stretched, and outright broken in pursuit of weaknesses.

Untrusted Configuration Data

In a manner similar to entities, configuration data such as XML Schemas and Web Services Description Language (WSDL) files ‘live’ outside the application yet provide key information to the entities involved.

Operating as a dynamic component, the configuration information that supplies details to a web services consumer has a unique standing in the architecture. These are the sources that determine the specific operations of a service and, as such, are highly sensitive to any form of manipulation or access. Typical web services configuration information data includes:

- XML Schemas provide specific details about the grammar of a document and create the
template from which a parser interprets the documents themselves.

- WSDL files provide detailed information about the services ports and bindings available to consumers.
- XSLT files provide a mapping from one schema to another, in order to support desired transformations such as the conversion of documents from one grammar to another.
- WS-Policy provides handling rules and guidance about preferences for entities in a web services system.

This configuration information described can be maintained on the application server itself, housed separately in a UDDI directory or part of shipped with the transaction itself. The accuracy and integrity of configuration information highlights the importance of addressing any possibility of compromise.

**XML Processors**

XML processors may be standalone utilities or integrated into any of the components described above. Basically, they provide the intelligence to interpret XML documents as inputs to an application. More specifically, these processors perform the following functions:

- Parse the XML document into its component parts. SAX and DOM are the most popular parsing approaches. DOM is a tree-based parsing technique that builds up an entire parse tree in memory. Rather than building a tree representation of an entire document, a SAX parser fires off a series of events as it reads through the document. Streaming API for XML introduces a streaming model to parsing that resembles the SAX approach. Finally, deferred DOM parsing does not create the full tree structure of objects in memory.

- Aggregate and instantiate an XML document for processing using configuration information that is fetched typically by resolving URI’s or external pointers to repositories.

- Transform the document by using XSLT to map content from one schema to another or any other mapping required by XML manipulations such as XML Digital Signatures

- Canonicalize data to ensure that it is not only well-formed (which is a function of the parser) but also specifically formatted so that the document will be identical wherever it happens to be built, most notably on the producer and consumer sides.

- Compress the data to meet the performance needs of a particular enterprise function.

XML processors are being integrated into every facet of the enterprise computing environment. For example,

- Data repositories contain processors to recognize parse and “shred” XML documents to be stored in file systems, XML-aware relational databases, and new XML databases.

- Web service development environments, such as applications that support J2EE and .Net, require XML processors in order to understand the inputs into the environment.

- Intelligent networks are becoming XML-aware relying on XML tags to perform common services such as content based routing and quality of service as well as value added
Top ten Web Service Threats

1. Coercive Parsing
   XML is already recognized as a standard file format for many applications. As the obvious successor to legacy ASCII and presentation-oriented html, its position is unchallenged. This is easily seen by the number of grammars that claim XML as their parent.

   The basic premise of a coercive parsing attack is to exploit the legacy bolt-on - XML-enabled components in the existing infrastructure that are operational. Even without a specific Web Services application these systems are still susceptible to XML based attacks that whose main objective is either to overwhelm the processing capabilities of the system or install malicious mobile code.

2. Parameter Tampering
   Parameters are used to convey client-specific information to the Web service in order to execute a specific remote operation. Since instructions on how to use parameters are explicitly described within a WSDL document, malicious users can play around with different parameter options in order to retrieve unauthorized information. For example by submitting special characters or unexpected content to the Web service can cause a denial of service condition or illegal access to database records.

   An attacker can embed, for example, command line code into a document that is parsed by an application that can create a command shell to execute the command. One instance of this problem is described by Georgi Guninski’s attack against Excel that formats an XML document to pass a command line to (in his example, but not limited to) enumerate the file system.

3. Recursive Payloads
   One of the strengths of XML is its ability to nest elements within a document to address the need for complex relationships among elements. The value is easy to see with forms that have a form name or purpose that contains many different value elements, such as a purchase order that incorporates shipping and billing addresses as well as various items and quantities ordered.

   We can intuitively acknowledge the value of nesting elements three or four levels, perhaps more. An attacker can easily create a document that attempts to stress and break an XML parser by creating a document that is 10,000 or 100,000 elements deep.

4. Oversize Payloads
   XML is verbose by design in its markup of existing data and information, so file size must always be considered. While an enterprise’s programmers and analysts will work to limit the size of a document, there are a number of reasons to have XML documents that are hundreds of megabytes or gigabytes in size. Sometimes this is a function of converting a batch file transfer process into real-time. It may also be anticipated in the multimedia (e.g. digital video) world where gigabyte files are the norm. Or, it could be an attacker again exercising the parser to execute a denial-of-service attack. Parsers based on the DOM model are especially susceptible to this attack given its need to model the entire document in memory prior to parsing.

5. Schema Poisoning
   XML Schemas provide formatting instructions for parsers when interpreting XML documents. Schemas are used for all of the major XML standard grammars coming out of OASIS. Because these schemas describe necessary pre-processing instructions, they are
susceptible to poisoning. An attacker may attempt to compromise the schema in its stored location and replace it with a similar but modified one.

Denial-of-service attacks against the grammar are straightforward if the schema is compromised. In addition, the door is open to manipulate data if data types are compromised, like changing dates to numbers when the application is performing arithmetic operations, or modifying the encoding to allow for data obfuscation that eventually gets through to a parser and re-formed into an attack, in the same way a Unicode attack can traverse directories through web servers.

6. WSDL Scanning

Web Services Description Language (WSDL) is an advertising mechanism for web services to dynamically describe the parameters used when connecting with specific methods. These files are often built automatically using utilities. These utilities, however, are designed to expose and describe all of the information available in a method.

In addition, the information provided in a WSDL file may allow an attacker to guess at other methods. For example, a service that offers stock quoting and trading services may advertise query methods like requestStockQuote, however also includes an unpublished transactional method such as tradeStockQuote. It is simple for a persistent hacker to cycle thru method string combinations (similar to cryptographic cipher unlocking) in order to discover unintentionally related or unpublished application programming interfaces.

7. Routing Detours

The WS-Routing specification provides a way to direct XML traffic through a complex environment. It operates by allowing an interim way station in an XML path to assign routing instructions to an XML document. If one of these web services way stations is compromised, it may participate in a man-in-the-middle attack by inserting bogus routing instructions to point a confidential document to a malicious location. From that location, then, it may be possible to forward on the document, after stripping out the malicious instructions, to its original destination.

8. External Entity Attack

Another benefit of XML is its ability to build documents dynamically at the time of insertion by pointing to a URI where the actual data exists. These external entities may not be trustworthy. An attacker can then replace the data being collected with malicious data.

9. SQL Injection

Database parsers are aimed at native database languages in the same fashion as SQL injection. SQL injection could allow an attacker to execute multiple commands in an input field by using native command separators like ‘;’ or pipes. This capability may allow an attacker to execute native stored procedures or invalidated SQL commands.

10. Replay Attack

Similar to the “network ping of death” a hacker can issue repetitive SOAP message requests in a bid to overload a Web service. This type of network activity will not be detected as an intrusion because the source IP is valid, the network packet behavior is valid and the HTTP request is well formed. However, the business behavior is not legitimate and constitutes an XML-based intrusion. In this manner, a completely valid XML payloads can be used to issue a denial of service attack.
Routing Secured SOAP Messages Through Multiple SOAP Intermediaries

Introduction
XML Web services are increasingly being adopted in the enterprise environment, and Web Services Enhancements (WSE) for Microsoft .NET provides many useful features that are needed and desired by enterprises. While some of the WS-* specifications are still evolving, important core specifications, such as WS-Security, have been submitted to the Technical Committee of OASIS, and WS-Addressing specifications have been submitted to the Working Group of W3C.

The implementation of XML Web services has also brought closer to life the benefits and goals of the architectural philosophy of service-orientation. It is important to note that the objective of designing good business software is to deliver agile systems, not service-oriented architectures. Rather, look at service-orientation as an approach by which business and technological agility can be achieved. XML Web services offer a good implementation mechanism to achieve service-orientation. However, bear in mind that it takes more than adopting XML Web services protocols in systems deployment to achieve the desired agility. Basic, good design principles need to be adhered to, as well. While I won't go into discussion of what constitutes services and the design principles of service-orientation, it is important to think of services as some functional block that offers certain application, business, or other features. Services can be, amongst other things:

- Message destinations in terms of applications
- Message handlers in terms of infrastructure
- Message carriers in terms of transport

How to route secure SOAP messages through multiple HTTP SOAP intermediaries? Each node provides a specific service before the message reaches the final endpoint. Each of the SOAP intermediary nodes performs a critical role in providing a functional service before routing to the next logical node in the entire application system.

WSE 2.0 and Next-Hop Routing
WSE 2.0 supports the WS-Addressing specifications, which support the concept of the "Next-Hop" routing mechanisms. The basis of this concept is that SOAP message senders should only know of a single endpoint. This next logical endpoint takes care of and abstracts away the additional routing needs. This "Next-Hop" routing is essentially how TCP/IP works. A message traveling on a TCP/IP transport contains information about where a message came from and where it's going, but it doesn't contain any details on how the message gets to where it is. As the message travels through the TCP/IP network, each node inspects the header to see where it's headed and then it sends the message to the next node nearest to the destination. This continues until the destination is reached.
As services compose at coarse-grained message boundaries, this form of message routing becomes the primary extensibility model and it can be very powerful with the implementation of standard XML Web services protocols in an entire grid of message pipelines or message bus.

Examples of problems that can be solved are in scenarios where:

- The service consumer (SC) and the service provider (SP) have no trust for each other and thus need trusted intermediaries to route messages across.
- In some cases, the SP may reside on a different sub-domain from the SC and will therefore need intermediaries to route messages across.
- In good service-orientation principles, the SP should be built with no integration of the SC in mind, and would therefore need other trusted intermediaries to relay messages to complete certain business processes and functions.
- There is a need to perform authentication or other infrastructure services that are not specific to the business service itself. Sometimes, different parties own these services. Designing and implementing a routing pattern such as the one explained above promotes good practice in the separation of concerns. This aids tremendously in extensibility and scalability of the entire system application.
- There is a need to abstract the location endpoint of the ultimate receiver from the consumer.

**Multiple SOAP Intermediaries Routing Example**

Let us imagine and assume a scenario where routing of messages through multiple SOAP intermediaries may be required.
Figure 1. Message-routing scenario with multiple SOAP intermediaries

Figure 1 illustrates an architectural design view of the message dispatching and routing chain of the solution. This involves a cross-organizational boundary communication call where a service consumer from ABC Company needs to invoke a secured SOAP service hosted within the internal network of XYZ Company.

There are a few obstacles along the way that needs to be solved. Obviously, having to communicate with the service provider within the company's internal firewall and network from the Internet is one challenge. This is a very probable scenario in today's world where companies usually like to start or test their Web service implementations internally first before exposing it to external partners and customers over the Internet. In our case, we can easily implement a SOAP message router within the DMZ, but on the same sub-domain that forwards the requests to the listening host. Moreover, this message router can also apply certain message assertions so that it complies with the existing service policies: for example, a policy to assert that the message needs to be signed and encrypted with a certain X509 digital certificate private key which is only available within the confines of XYZ Company.

Requiring a policy assertion to use an asymmetric key algorithm (X509) is inherently more secure than a username/password token for signing and encryption. However, it will be pretty unrealistic to expect that the hundreds of service consumers will each have their own set of X509 digital certificates key pair. In this case, I propose to have another SOAP message router within the confines of ABC Company whose role is to validate and verify the user credentials and then sign those messages with a common set of X509 digital certificates key pair. We may utilize just one or a few of these digital certificates in this instance. This is significantly cheaper, and more importantly, more manageable and maintainable than expecting all service consumers to have their own set of Public-Private key pair. By implementing these sets of requirements described above, messages being dispatched out to the Internet from ABC Company are signed and encrypted separately by different sets of X509 digital certificate keys, which, in turn, is much more secure than signing and encrypting them with just the username tokens alone.

To improve availability of the SOAP message routers, you may choose to run multiple instances of them and load-balance them on the network to better distribute and share the connection and processing loads.

Note With the solution I have proposed above, I am assuming that XYZ Company trusts the delegation of the authentication of all service consumers to the SOAP intermediary of ABC Company. In other words, XYZ Company trusts all clients who are successfully authenticated by ABC Company. I am also assuming no differentiation of privileges or roles in accessing the SOAP service of XYZ Company. However, if XYZ Company chooses to take a more proactive and assertive role in the authentication of the service consumers of ABC Company, it is a perfectly acceptable solution to sign with the username token and then encrypt all data, including the username token in the message, with an X509 digital certificate. The only obvious cost to this alternative is the need to maintain another set of username information at another location. This is a huge tradeoff in this instance, and was not an option in the previous example.
Now, let us go more into details and assume that the service consumer (SC) and the service provider (SP) are separated by an arbitrary number of trusted SOAP intermediaries. The SC, in this instance, is the message sender and the SP is the message receiver. The exact structure of this routing chain is unknown to both the SC and SP. Certain business architectural requirements also explicitly specify that the SP only implement the functionality of the destination service and nothing else. The authentication service of the solution is delegated to and implemented by another node in the routing chain (SOAP Router 1, or SR1). Further extensibility is introduced into the mix when another node (SOAP Router 2, or SR2) is tasked with the responsibility to sign and then dispatch the SOAP message to the appropriate destination endpoint. Therefore, in this example, the first intermediate SOAP node (SR1) provides the authentication service while the second node (SR2) provides the URI Referral service. There you have it, each intermediary node, which may be owned by different stakeholders, plays a very important role in the entire routing chain and there is no other way a message can get from the SC to the SP without passing through SR1 and SR2. In other words, the starting SOAP message from the SC gets massaged a couple of times before it reaches a state whereby the SP can understand and process it. See Table 1 and the sequence diagram in Figure 2 for a better representation.

Table 1. SOAP intermediary message-routing process

<table>
<thead>
<tr>
<th>SOAP Node</th>
<th>Role</th>
</tr>
</thead>
</table>
| SC        | Encrypts a SOAP message meant for the SP with SP's Public Key  
|           | Signs the message meant for SR1 with username token  
|           | Sends the SOAP message to SR1  
|           | SC has no idea on the destination endpoint of SP. SC only knows of the existence and endpoint of SR1. |
| SR1       | Validates message signature with username credentials retrieved from the Username DB  
|           | Signs Message with its Private Key  
|           | Embeds signature and Public Key into a new Security Header whose intended target is SR2  
|           | SR1 still has no idea on the destination endpoint of SP. SR1 only knows of the existence and endpoint of SR2.  
|           | Encryption meant for SP remains intact. |
| SR2       | Verifies the signature of SR1  
|           | Verifies the incoming message signature comes from SR1. In other words, SR2 can only communicate with SR1 by accepting the digital certificate signatures of SR1.  
|           | Signs the message headers and body with its own Private key  
|           | Dispatches the signed message to SP via an URI referral service |
• Encryption meant for SP remains intact.

SP
• Policy file checks for conformance.
• Verifies the Signature of SR2
• Decrypts the message
• Processes and returns SOAP Message
• Message routes back to the SC via the same soap nodex in the reverse order.

Figure 2. SOAP intermediary message-routing sequence
As you can see in Figure 2, the message will need to be authenticated along the way through these SOAP intermediaries. New security headers targeted for different SOAP actors can be added into the message as well. The corresponding targeted SOAP actor will know to process. Once a security header targeted for a specific node is processed successfully, it is removed from the SOAP message.