Chapter 5

Hardening Web Services

by Mark O'Neill

- Harden Your Web Services Environment
- Understand Web Services
- Understand and Use Standards, Profiles, and Specifications Such as W3C, OASIS, and WS-I
- Implement Security Requirements for Your Web Services
- Block "Malicious XML" Attacks
- Implement a Policy to Secure Your Organization's Services-Oriented Architecture
- Review and Implement Products that Protect Web Services

Imagine the scene: a chief security officer (CSO) has found out that her colleagues in the IT department are going to be using web services for internal integration projects. In fact, there are *already* some pilot web services deployments and soon the company will be sending eXtensible Markup Language (XML) across the firewall to business partners. The CSO must bring these web services under the blanket of corporate security, the first step of which is to research exactly what are the specific security issues facing web services. After reading a few introductory web services security articles, the CSO starts to get a sinking feeling because she has learned that XML is sent over web ports, firewalls are oblivious to XML traffic, and business applications talk to each other by sending messages across the network, using text-based protocols. To the CSO, this sounds like a grand plot to bypass the company's carefully constructed security policies.

This scene has been played out in countless organizations since web services emerged. The security picture actually seemed darker in the early days of web services, back in 2001 and 2002, when many of the press descriptions of web services described a "giant directory in the sky" (using Universal Description, Discovery, and Integration [UDDI]) for businesses to connect to other businesses on an ad hoc basis to order goods and services. This early fixation on ambitious e-commerce scenarios came about because the early days of web services happened to overlap with the last days of the business-to-business (B2B) hype wave, when the promise of "e-marketplace hubs" was beginning to sour. The legacy of that period in some quarters is a deep suspicion of web services.

This suspicion has been shown to be misdirected. Instead of being an immediate silver-bullet solution for B2B, web services gradually found a niche in solving internal application-integration problems. Although less glamorous than B2B, application integration is an acknowledged problem familiar to anyone who has ever tried to roll out a new Customer Relationship Management (CRM) or Enterprise Resource Planning (ERP) system, or tried to access the information held inside such systems. This position inside the organization is where web services have found their niche, with the ambitious cross-firewall B2B scenarios being set aside until recently.

At the same time that web services were finding a niche on the internal network behind the firewall and progressing through the early proof-of-concept stage, a great deal of work was being done to develop standards for XML and web services (WS) security. These standards were needed to fulfill basic security requirements, such as confidentiality and integrity, and to deal with situations in which XML travels between systems that use differing security technologies. This chapter examines these standards, which include WS-Security, Security Assertion Markup Language (SAML), XML Signature, and XML Encryption. This chapter also examines new specifications that, although not yet standards, address important security requirements for web services, such as WS-Trust, WS-Policy, and WS-SecureConversation.

While vendors and standards bodies were developing new standards and specifications for web services security, parallel work by security researchers on both sides was underway to discover security vulnerabilities in XML and web services. It turns out that, like a Shakespearean tragic hero, some of the most powerful characteristics of a web service are also its chief vulnerabilities and can be used against it by an attacker.

However, applying sound security practices can protect your web services from known attacks and potentially protect them from attacks as yet unknown. In this chapter, you encounter some of the chief vulnerabilities of web services, including their susceptibility to XML denial of service (DoS) attacks, and learn how to defend your own web services against them.

A recent trend has been to link together isolated internal web services projects into a services-oriented architecture (SOA). An SOA allows you to apply management and co-ordination to web services traffic in a top-down view. An SOA also allows you to add and remove web services, or swap them for different implementations, without disruption to the system as a whole. The use of SOAs introduces security challenges, so this chapter also examines how you can map security to an SOA.

Finally, this chapter looks at practical examples of vendor products for web services security, one from each of the following established categories: a web application security product that also addresses web services security, an XML router, and an XML gateway/firewall product.

Harden Your Web Services Environment

Although web services are new technologies, they depend on an underlying stack of preexisting technologies, some of which have been around for ten years or more. A security professional who secures only their web services interface, and whose system is then brought down by an attack at the operating system level, hasn't done their job properly. An attacker always goes after the point of least resistance. Do not make the attacker's job easy by failing to apply patches or to follow other security-hardening best practices for the OS, the network environment, and the development process. If your web services are being deployed using a web server, then ensure that your web server patches are up to date and that the web server is fully secured according to best practices and according to your specific requirements. If you are reading that last sentence and wondering what "*If* your web services are…using a web server" means, then the next section is for you. It explains both what web services are and, along the way, why the name "web service" is so misleading.

Understand Web Services

You cannot secure an information system, application, or service that you know little to nothing about. Before you can harden web services, you need to understand what they are, where they are currently used, and how they are integrated into the rest of your information systems. The term *web services* implies a definition of "services on the World Wide Web." If this were the definition, then an airline-booking web site would count as a web service. However, the term *web service* refers to something specific. There are many

definitions of web services available, mostly from vendors who add a spin in order to say "this is not all that different from what we've been doing all along." So, to get a more accurate, objective definition, here is the definition of web service from the World Wide Web Consortium (W3C):

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed with HTTP using an XML serialization in conjunction with other XML-related standards.
>
> [Source: *W3C Web Services Glossary*, Sec. 3, "General Terms," http://www.w3.org/TR/ws-gloss/#defs]

This definition includes several key points, which are examined in the following sections.

Web Services Processing Is Between Machines, Not People and Machines

A web service is for *machine-to-machine* interaction. A human end user does not directly invoke a web service. The web service is accessed by a client, which the W3C defines as "A system entity making use of a Web service." Either no human end user is involved at all or the human end user accesses the web service indirectly. An example of the latter scenario is the National Basketball Association's web site, http://www.nba.com, which uses Amazon Web Services (AWS). An NBA.com customer does not access AWS directly. Instead, the customer browses to NBA.com, fills their shopping cart with basketball gear, and then pays for the purchase, all the time staying at NBA.com. In the background, XML passes between NBA.com and AWS, unknown to the end user. Prior to the existence of the Web, this system may have been implemented by giving the customer a specific application that orders basketball gear by communicating over the Internet with the NBA (client/server architecture). In a classic business-to-consumer (B2C) web system design, an n-tier architecture (a system where parts of an application reside on some number, or *n* computer systems) would be used. Web services can therefore be seen as a third evolution of system architecture, evolving from client/server and n-tier architecture. Figure 5-1 shows this evolution in context.

The difference between "client" and "end user" is very important in web services security. If a web service is to be locked down such that only certain end users (human beings) can access it, XML messages passed to that web service must contain information about the end user, such as where they were authenticated or what attributes they have (e.g., Manager). This is more complex than authenticating an end user at a web site, since the end user directly "touches" the web site. Standards such as SAML and WS-Security

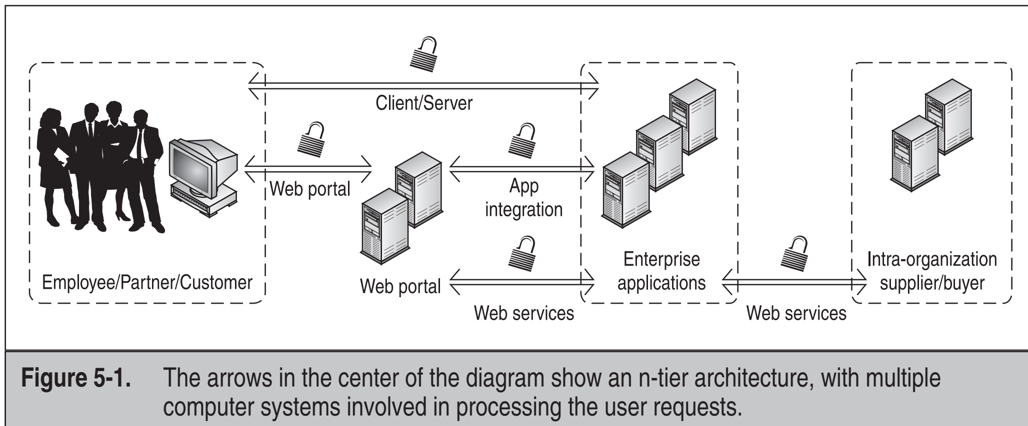

Figure 5-1. The arrows in the center of the diagram show an n-tier architecture, with multiple computer systems involved in processing the user requests.

are relevant for solving this architectural problem, as you will see in the sections "Use Principal Propagation to Permit Identification of Authenticated and Unauthenticated Messages" and "Understand and Use Standards, Profiles, and Specifications Such as W3C, OASIS, and WS-I" later in this chapter. Techniques such as credential propagation and credential mapping are also options to consider.

Web Services Definition Language

The second sentence of the W3C definition states that the interface of a web service can be described using WSDL. This is not the same as saying that a web service *must* advertise its interface using WSDL. Clearly, from a security point of view, it is not wise to give an attacker information about your web service. If you use WSDL to describe the web service interface for authorized users, you should protect it. If you have created a policy to protect your web service's SOAP interface, it makes sense to also bring the WSDL interface under the same cloak of protection.

As well as being the subject of security protection, WSDL itself can be used to express security policies. WS-PolicyAttachment is a specification that adds security policy information to WSDL files, including directives on what parts of SOAP messages must be signed, what parts must be encrypted, and which keys and token formats to use. This information is used by clients who must send secure messages.

REST and Plain-XML Web Services

The last sentence of the W3C definition ventures into somewhat controversial territory: "Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed with HTTP using an XML serialization in conjunction with other XML-related standards." Many people would argue with the contention that using SOAP is a hard-and-fast requirement for a system to qualify as a

"web service." Many early adopters of web services chose not to use SOAP, and instead sent XML messages without SOAP "envelopes." Proponents of REST (REpresentation State Transfer) would argue that a web service should be called by sending HTTP GETs and POSTs to a URI, which simply returns XML (not SOAP). From a security point of view, you should be aware that the choice to not use SOAP means that security standards that rely on SOAP, such as WS-Security, cannot be used.

RPC and Document-Based SOAP

A full name is not given for SOAP in the preceding section because SOAP is no longer an acronym. In the days when web services were synonymous with object orientation and remote procedure calls, and were seen as being "CORBA through firewalls" or "DCOM with angle brackets," SOAP stood for Simple Object Access Protocol. It was intended as a way for the methods of objects to be accessed in a simplified fashion, using XML to serialize the parameters passed to them. Some of the earliest SOAP applications would simply take an EJB or COM component and create a separate SOAP operation for each method of the object. They would then send and receive SOAP messages and "marshal" the data to and from the "web service enabled" object.

Early adopters using these first SOAP tools quickly realized the shortcomings in this map-SOAP-operations-directly-to-the-methods-of-an-object model. The first was interoperability. It was one thing to say "anyone can access my application because it talks XML, and you can make XML in Notepad," but it was quite another thing to create a SOAP message containing serialized Java objects as its parameters. Similarly, although web services were supposed to be "loosely coupled," early SOAP products placed a lot of demands on the web service requester, such as a requirement to be online at the same time as the service that they were accessing. It was not qualitatively better than using an API. Finally, creating a web service at the fine-grained method level was not optimal for message transfer, given the size of XML messages. It was a better option to create a higher-level web service that in turn called the lower-level methods directly. This use of high-level web services also makes web services more portable and less tied to a specific implementation.

Because of the preceding shortcomings in the model, a backlash caused movement away from "RPC-style" and "encoded" SOAP and toward "messaging-style" and "document-based" SOAP. *Document-based SOAP* means that an XML message is not an encoded object but rather is an XML document that is independent of the application used to create it and is generally definable by an XML Schema using simple XML elements such as strings and integers. *Messaging-style SOAP* means that the SOAP message can be sent in a store-and-forward manner, like e-mail, and can involve the kinds of asynchronous exchanges that are very difficult to implement in an RPC system. It is not unreasonable to now think of SOAP as a kind of e-mail between applications. Ensure that the people charged with developing web services for your organization see the value of loosely coupled architecture and are not just developing web services for the sake of using the latest technology.

Transport Independence

The final point regarding the W3C definition is that while XML is "typically conveyed with HTTP," HTTP is not mandatory. SOAP is designed to be transport independent. XML being sent using FTP or SMTP also counts as a web service. That is why the name "web service" is misleading; "Internet service" would have been better. That said, HTTP accounts for the vast majority of web services traffic today, giving rise to the most often cited web services security issue: firewalls have difficulty distinguishing web services traffic from the normal web browsing traffic that passes through the same TCP ports.

This issue is not quite as dramatic as it is sometimes claimed to be, since web services traffic is distinguishable by its Content-type HTTP header directive (text/xml or application/soap+xml) and has other characteristics that are noticeable by firewalls, such as the fact that its first line is an XML directive (either <?xml version="1.0"?> or <?xml version="1.1"?>, depending on the XML version). The difficulty for firewalls is not blocking SOAP messages but rather selectively allowing SOAP messages through, based on their originator and their content. This involves the use of the standards (WS-Security and others) and techniques (Schema validation and others) that are described in the next section.

Understand and Use Standards, Profiles, and Specifications Such as W3C, OASIS, and WS-I

XML and SOAP have been developed within the framework of the W3C. When it comes to the security of web services, the W3C is joined by the Organization for the Advancement of Structured Information Standards (OASIS). The W3C is the development base for XML Encryption and XML Signature, as well as the XML Key Management Specification (XKMS). OASIS is the development base for SAML and WS-Security.

In addition to these standards bodies, the Web Services Interoperability (WS-I) organization takes the standards from OASIS and W3C and produces usage "profiles" that describe how standards are to be used together to achieve common tasks. A simple explanation of the profiles is that while the standards contain many uses of the word "may" (a certain algorithm *may* be used, a certain security token format *may* be used, etc.), the WS-I specifies which of these *must* be used if a vendor is to claim interoperability. Picture the WS-I lining up the standards on top of each other and cutting out a cross section through them—this cross section is that piece that each WS-I-compliant vendor must implement. A vendor is not required to be a member of the WS-I to claim WS-I interoperability.

The WS-I has produced a SOAP Basic Profile that web services vendors can use to test their products' interoperability with the products of other vendors. The WS-I is also producing a WS-I Basic Security Profile, which includes profiles for SSL and HTTP-Auth, alongside profiles that use WS-Security. This shows that security for web services is not limited to new web services security technologies, but that older technologies still apply. Interestingly, the WS-I Basic Security Profile draft devotes a lot of attention to the security of the security technologies themselves. For example, a vendor implementing authentication using the WS-Security X.509 Certificate Token profile must ensure that an attacker cannot intercept a valid message and replay it (a *capture-replay attack*) to gain entry to a protected web service. A capture-replay attack is the information security equivalent of finding a signed document sitting beside a fax machine and re-faxing it. If the recipient is not careful about examining timestamps or numbering in the messages they receive, then they may trust this second message, which in turn calls into question the trust of the first message. The WS-I describes best practices with regard to timestamps and message numbering, since this is a vague area in the standards themselves.

TIP When choosing a vendor solution for web services security, place a higher value on WS-I compliance than on simple conformance to standards. Two vendors that both claim WS-Security conformance may not be interoperable, because they may implement different portions of WS-Security. However, two vendors that both claim WS-I compliance will be interoperable.

The WS-* Specifications

A large group of specifications, commonly called the WS-* specifications because they all begin in WS-, is presently outside the bounds of standards bodies. These WS-* specifications include WS-Trust, WS-Policy, and WS-SecureConversation, all of which are examined in this chapter. These specifications are implemented in toolkits such as the Microsoft Web Services Enhancements (WSE) 2.0 toolkit, and therefore are relevant to the discussion of web services security. These WS-* specifications can be "composed" together to implement security integration, such as using WS-Trust to request security tokens that are used by WS-SecureConversation, so they are sometimes called the "composable architecture."

Implement Security Requirements for Your Web Services

Now that you know what web services are, you are in a position to examine their security requirements. The requirements for web services are not all that different from the requirements for any other distributed messaging system. In addition, the technologies used to meet the requirements are familiar—cryptography, message validation, and others.

Any system, including a system based on web services, must fulfill these basic requirements of security:

- **Authentication** Who is sending this message?
- **Authorization** Is the authenticated subject allowed access to this target?
- **Integrity** Was the message, or system, tampered with?
- **Confidentiality** Can the information be read while in transit or in storage?
- **Audit** Is a secure store of transactions recorded?
- **Administration** How straightforward is policy management?
- **Availability** Is this system vulnerable to a denial of service (DoS) attack?

It is important to keep these general requirements in mind and avoid fixating on specific standards. This means thinking of ways in which you can guarantee integrity of these messages rather than thinking of how you can build XML Signature into this system.

Implement Authentication for Your Web Services

If you reexamine Figure 5-1, you will see that two types of authentication are important for web services:

- **Client authentication** Authenticating the application that is sending the XML message to the web service
- **End-user authentication** Embedding information about the end user into the messages sent to the web service

When implementing you should choose between transport- or message-based client authentication, make sure you remove credentials after authentication, and create a system that allows the identification of authenticated and unauthenticated messages.

How to Choose Between Transport- and Message-Based Client Authentication

Client authentication may be performed using transport security technologies such as mutual SSL, SSL with HTTP-Authentication, or SSL with HTTP-DigestAuthentication. Client authentication may also be performed using message-level authentication, which is independent of the transport that is used. WS-Security is used for message-level authentication; in particular, the WS-Security X.509 Certificate Token profile for certificate-based authentication, the WS-Security Kerberos profile for Kerberos authentication, and the WS-Security UsernameToken profile for password-based authentication.

So, when should you use transport-based authentication and when should you use message-based authentication? The answer is that when your web services are point to point, with clients sending XML directly to web services, then transport-level authentication is sufficient. When SOAP intermediaries are involved, creating multiple

hops between the client and web service, then message-level authentication should be used for end-to-end security.

TIP HTTP-Auth and HTTP-DigestAuth, which are challenge-response authentication systems, both involve two round trips to the target web server: the first to pick up the challenge, and the second to send back the response. Therefore, HTTP-Auth and HTTP-DigestAuth are not good choices for authenticating large XML messages, since the messages must be sent up to the web service twice.

Authentication is based on *credentials* that are used to establish identity. Examples of credentials include username/password combinations and X.509 certificates (when combined with a proof of possession of the corresponding private key). Following authentication, the *principle name* is proven. This is the identity of the authenticated party. Examples of principle names include a username, the distinguished name (DN) of an X.509 digital certificate, or a Windows logon name such as \\ServerI\JoeUser.

Remove Credentials After Authentication

A WS-Security UsernameToken structure that contains both a username and a password is a credential. If you do not trust the applications processing the message downstream, consider removing the credentials from the message following authentication. The following Extensible Stylesheet Language Transformations (XSLT) transform will remove the contents of a SOAP header, including the authentication credentials, from a SOAP message:

```
<?xml version="1.0"?>
<xsl:stylesheet exclude-result-prefixes="SOAP-ENV SOAP"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml"/>
<xsl:strip-space elements="*"/>
<xsl:template match="/">
      <xsl:apply-templates select="//SOAP-ENV:Envelope"/>
</xsl:template>
<xsl:template match="*">
  <xsl:if test="not (name(.)='soapenv:Header')">
    <xsl:element name="{name(.)}" namespace="{namespace-uri(.)}">
      <xsl:apply-templates select="@*|*|text()"/>
    </xsl:element>
  </xsl:if>
</xsl:template>
<xsl:template match="@*">
  <xsl:copy/>
</xsl:template>
</xsl:stylesheet>
```

Use Principal Propagation to Permit Identification of Authenticated and Unauthenticated Messages

Following authentication, an SAML token may be issued to assert that the client was authenticated, is authorized for a particular service, or has a certain attribute (such as a credit limit). In this way, authentication information, such as the Principle name (the name of the client that was authenticated) can be propagated to the next web service that processes the XML message. Without this "principle propagation" functionality, information about the client may be lost following the authentication event. Other benefits of this principle propagation approach include

- An application can ensure that XML messages passed to it have come through an XML security gateway, by examining the message for a security token that was digitally signed by the XML security gateway. If this security token is not present, the message may be passed up to the XML security gateway for processing, may be quarantined, or may be dropped.
- An application may enforce a rule such as "only clients who authenticated with X.509 certificates are allowed to access this service," since information about the authentication method is included inside an SAML Authentication Statement.
- Messages may be correlated together, across multiple XML-processing applications, based on having identical security tokens inside

An example of a SOAP message that uses principle propagation, by embedding an SAML authentication token, is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
"http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
<soap:Header>
<wsse:Security soap:actor="current" xmlns:wsse="http://docs.oasis
-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
<saml:Assertion AssertionID=
"vordel-1087530254730" IssueInstant="2004-06-18T03:44:14Z"
Issuer="Corporate CA" MajorVersion="1" MinorVersion=
"1"xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
<saml:Conditions NotBefore="2004-06-18T12:00:00Z"
NotOnOrAfter="2004-06-18T12:10:00Z"/>
<saml:AuthenticationStatement
AuthenticationInstant="2004-06-18T03:44:14Z"
AuthenticationMethod="urn:ietf:rfc:2246">
<saml:SubjectLocality DNSAddress="client.com" IPAddress="192.168.0.1"/>
    <saml:Subject>
    <saml:NameIdentifier Format=
"urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName">
CN=Client Name, O=Client Organisation,
```

```
C=SE</saml:NameIdentifier>
    </saml:Subject>
    </saml:AuthenticationStatement>
</saml:Assertion>
</wsse:Security>
</soap:Header>
<soap:Body>
<StockQuoteRequest xmlns="http://www.mystockquoteexample.com">
<symbols>
<Symbol>BENO</Symbol>
<Symbol>KSTN</Symbol>
</symbols>
</StockQuoteRequest>
</soap:Body>
</soap:Envelope>
```

In the preceding code example, the "urn:ietf:rfc:2246" in the AuthenticationMethod attribute refers to SSL/TLS (Transport Layer Security) client authentication. The client's information is contained in the Subject/NameIdentifier element (CN=Client Name, O=Client Organization, C=SE). Applications that process this XML message know the identity of the authenticated party, how they were authenticated, and when they were authenticated (via the AuthenticationInstant attribute). It is recommended that SAML assertions be digitally signed, together with the SOAP messages to which they are bound, to detect a trivial copy-and-paste attack whereby an SAML assertion is hijacked for use in a different SOAP message.

Embed User Authentication Tokens

End-user authentication is achieved through a similar means to principle propagation. A token, embedded inside the SOAP message, conveys information about the end user. This technique is the basis of Project Liberty (a project defined by the Liberty Alliance, http://www.projectliberty.org/), whereby a user can log on once to an authentication provider, who then issues tokens that are passed to other services used by the same user. Therefore, the user does not have to be reauthenticated. This solution is safer than the cruder forms of single sign-on, such as credential propagation (passing the user's password with onbound requests, to log the user into the remote service) or the use of cookies (which are exclusive to the HTTP transport method and thus are unusable for SOAP messages, which do not use HTTP).

When security tokens must be converted from one format to another, for example to convert an incoming Kerberos ticket into a UsernameToken structure to place into outbound messages, WS-Trust may be used. WS-Trust includes a RequestSecurityToken message that may be used to request a certain security token, based on a different security token. WS-Trust is designed to enable trust relationships that span multiple security domains.

TIP To test a web service for compliance with SSL, HTTP-Auth, WS-Security, or SAML, or to generate example XML messages that use these standards, use the free SOAPbox tool from Vordel that is available at http://www.vordel.com.

Implement Web Services Authorization Using SAML and Link to Web Site Authorization

Once the sender or end user is authenticated, the next step is to decide if they allowed to access the resource that they are requesting. This step is called authorization. Authorization typically follows authentication—however, with role-based access control (RBAC), there is an intermediate step. In RBAC, the authorization isn't directly based on the user's identity; instead, it is based on an attribute of the user (e.g., their role) or on an attribute of the "environment" (e.g., whether or not the day is a public holiday). This is more scalable and manageable than the use of access control lists (ACLs).

You saw in the previous section, "Implement Authentication for Your Web Services," that SAML can be used to "assert" attributes of the user, which are used for authorization. SAML can also be used to request authorization. Many existing tools exist for web site authorization (such as Netegrity SiteMinder, Entrust GetAccess, RSA ClearTrust, and Oblix COREid). Some, such as the Entrust GetAccess product, include an SAML interface that may be used to request authorization.

The XML Access Control Markup Language (XACML) may be used to express authorization rules in XML format. XACML is a good example of an XML security specification that is not specifically designed to secure a web service; instead, it is an example of XML technology that is used to solve a security interoperability problem. Before XACML, there was no standard method of expressing access control information in a structured, vendor-neutral format. Now that XACML exists, it is an obvious choice for organizations that wish to submit their access control policies to third-party audit, or for organizations that wish to migrate access control rules from one system to another without the need for rekeying.

Ensure Message Integrity—XML Signature, PKCS#7 Signature, SSL/TLS, and IPSec

It is a common misconception that when cryptography is used, message integrity is a side benefit that is obtained "for free." This is not true—an encrypted message may be subtly changed without impacting its ability to be decrypted. Therefore, it is important to ensure that any messaging system, including web services, guarantees message integrity. SSL/TLS guarantees integrity at the transport layer. IP Security (IPSec) also guarantees

integrity, at a lower level. Digital signatures may be used at the message level. PKCS#7 signature is the basis of digitally signed Secure/Multipurpose Internet Mail Extensions (S/MIME) e-mails and is sanctioned by the WS-I draft Basic Security Profile for use with SOAP messages.

XML Signature is a W3C and IETF signature standard that allows a digital signature to be expressed using XML. XML Signature does not invent any new signature algorithms and therefore cannot be said to be "stronger" or "weaker" than signature formats that preceded it, such as PKCS#7 signature. WS-Security describes how a SOAP message is signed. In particular, WS-Security specifies that the XML Signature structure is placed inside the Security element, which in turn is placed into the SOAP header (SOAP messages, like TCP packets, have headers and bodies).

XML Signature supports the signing of more than one "reference." This is very useful for digitally signing SOAP messages, since, as well as signing the message body, a timestamp must be signed in order to avoid a capture-replay attack. XML Signature provides "persistent" integrity for XML documents, meaning that the integrity is still enforced after the XML message has been received, processed, and stored.

Inform Clients of Security Requirements Using WS-Policy

If you have set up a web service that requires incoming message bodies and message timestamps to be digitally signed, how can you convey that information to clients? This is where WS-Policy comes into play. WS-Policy allows a web service to convey information about its security policy to clients. The following WS-Policy document describes a policy for a web service endpoint at the fictitious web site http://www.mycompany.com/StockService. The web service policy states that all messages sent to the web service must have their SOAP bodies digitally signed and must contain digitally signed timestamps.

```
<?xml version="1.0" encoding="utf-8"?>
<policyDocument xmlns="http://schemas.microsoft.com/wse/2003/06/Policy">
<mappings xmlns:wse="https://schemas.microsoft.com/wse/2003/06/Policy">
<endpoint uri=" http://www.mycompany.com/StockService ">
<defaultOperation>
<request policy="#Sign-X.509" />
<response policy="" />
<fault policy="" />
</defaultOperation>
</endpoint>
</mappings>
<policies xmlns:wsu=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
xmlns:wssp="http://schemas.xmlsoap.org/ws/2002/12/secext"
xmlns:wse="http://schemas.microsoft.com/wse/2003/06/Policy"
xmlns:wsse="http://docs.oasis-open.org
/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
<wsp:Policy wsu:Id="Sign-X.509">
<!--MessagePredicate is used to require headers.
```

```
This assertion should be used along with the Integrity
assertion when the presence of the signed element is required. -->
<wsp:MessagePredicate wsp:Usage="wsp:Required" Dialect=
"http://schemas.xmlsoap.org/2002/12/wsse#part" >
wsp:Body() wsp:Header(wsa:MessageID) wse:Timestamp()
</wsp:MessagePredicate>
<!--The Integrity assertion is used to ensure that the message
 is signed with X.509. Many Web services will also use the token
for authorization, such as by using the <wse:Role> claim or
specific X.509 claims.-->
<wssp:Integrity wsp:Usage="wsp:Required">
<wssp:TokenInfo>
<wssp:SecurityToken wse:IdentityToken="true">
<wssp:TokenType>http://docs.oasis
-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
1.0#X509v3</wssp:TokenType>
<wssp:Claims>
<wssp:SubjectName MatchType="wssp:Exact">
StockClient
</wssp:SubjectName>
<wssp:X509Extension OID="1.2.23.44"MatchType="wssp:Exact">
fpmjxsEad4rl2dms0d
</wssp:X509Extension>
</wssp:Claims>
</wssp:SecurityToken>
</wssp:TokenInfo>
<wssp:MessageParts Dialect=
"http://schemas.xmlsoap.org/2002/12/wsse#part">
 wsp:Header(wsa:MessageID) wse:Timestamp()
</wssp:MessageParts>
</wssp:Integrity>
</wsp:Policy>
</policies>
</policyDocument>
```

In the preceding WS-Policy code listing, notice the stipulation that the SOAP body and the WS-Security timestamp are both signed. In addition, there is a stipulation that an X.509 certificate is the token format that is to be sent with the incoming messages to the web service. The Microsoft WSE 2.0 toolkit supports WS-Policy and allows you to automatically generate a WS-Policy document like the preceding example and attach it to web services that use the WSE.

Guarantee Integrity by Validating the Integrity of the System as a Whole

You can guarantee the "integrity" of the message only if you can guarantee the integrity of the system as a whole. This means you must provide intrusion detection safeguards, as well as tamper-control tools such as Tripwire.

TIP In addition to checking XML Signature over incoming XML messages, it is also a good policy to digitally sign outgoing messages. This ensures that no recipient can claim to have received a different XML message from your web service. Consider performing outbound signing (or response signing) as a matter of course, unless the processing overhead is too great (and in that case, consider using a cryptographic acceleration card from a vendor such as nCipher or SafeNet).

Implement Confidentiality: XML Encryption, SSL/TLS, and IPSec

Many people new to information security believe incorrectly that confidentiality is synonymous with security and that encryption is synonymous with confidentiality. In this view, a "secure" document means an encrypted document. Therefore, XML Encryption is often singled out as being the most important web services security standard. XML Encryption is certainly important, since it allows part of a SOAP message to be encrypted, while the rest may be in plain text. The reason for not encrypting an entire SOAP message is to allow SOAP intermediaries to read the message and discern the routing information that they require. Like XML Signature, XML Encryption is bound to SOAP using WS-Security. The "confidentiality" WS-Security element is where the XML Encryption structure is placed, and this in turn is placed inside the SOAP header. WS-Policy may also be used to convey a rule that a certain part, or parts, of a SOAP message must be signed. WS-Policy can also be used to convey a public key that is to be used to encrypt the SOAP Body parts of incoming SOAP messages.

XML Encryption operates at the message level and is independent of the underlying transport. Another message-level specification that is relevant here is WS-SecureConversation, which allows a security session to be set up between a web service requester and a web service. This session typically is initiated by negotiating a symmetric key, which is then used for encryption of all subsequent messages that are part of that session. This is directly analogous to how SSL works, and carries the same benefits as SSL in terms of speed of throughput.

IPSec and SSL/TLS also provide confidentiality, although they do so only on a per-hop basis. If you do not trust the intermediaries who are processing your XML messages, or if you require complete transport independence of your XML messages, then consider using XML Encryption for end-to-end security. One particular advantage of XML Encryption is that its confidentiality extends beyond the time when the message is transported, to the time when an XML document is stored. In this way, XML Encryption provides *persistent* encryption.

Privacy is sometimes confused with confidentiality. Privacy implies consideration of the context and semantics of the data to be protected. Typically this means that user preferences or a legal requirement specify that certain data must be kept confidential. Although privacy is different from confidentiality, it is often enforced using confidentiality. For example, a WS-Policy document may stipulate that the PatientRecord block of XML messages sent to a web service is always encrypted, and the WS-Policy document may contain the key used for this encryption. That is an example of privacy rules being enforced using web services security.

Provide Web Services Auditing Using XML Signature and XAdES

The ability to write an audit trail is important for any security system, including a web services security system. XML Signature is clearly useful for ensuring that audit trail information cannot be unknowingly tampered with. An audit trail must be long-running, however, and that is why the European Union Digital Signatures Directive was created. The European Telecommunications Standards Institute (ETSI) developed a standard named XML Advanced Electronic Signatures (XAdES), which is now ETSI standard TS 101 903. XAdES extends XML Signature by defining XML formats for advanced electronic signatures that remain valid over long periods of time (e.g., 30 years). This involves storage of the key and all key-validation information with the signature. If you wish to store web services audit logs over a long period of time, consider using XAdES.

Avoid Solutions that Require You to Configure Security Manually by Editing XML Files

Administrating a security policy for web services involves two broad activities: *provisioning*, the task of assigning new security policies and new client profiles to the system, and ongoing *policy management*, which involves changing policies, rolling back policies, and adding new policies. Policy management becomes an urgent task when a security policy must be changed in a hurry—for example, when a system is under attack. Therefore, the administration of your web services security policies must be as intuitive and straightforward as possible. It is strongly recommended that you do not rely on the direct use of XML files to configure your web services security policies. Although you have seen in this chapter that technologies such as WS-Policy and XACML can be used to express security policies, that does not mean that you must construct these XML files manually. XML, like web services, is designed for machine-to-machine communication, not for human consumption. Therefore, steer clear of solutions that require you to manually manipulate XML files to set security policies.

Ensure Web Services Availability

Availability involves protecting a web service by blocking unwanted message “storms.” That is a job for a traditional network firewall. In addition, messages that are designed to cause XML processing problems must be blocked. These are called XML DoS attacks, but may also be unintentional. An XML DoS attack differs from a traditional DoS attack because it is asynchronous. A single XML message can do significant damage, as we will see in the following sections.

Block "Malicious XML" Attacks

The use of XML introduces new forms of attack. These attacks can bring an XML-processing system to a halt or provide unauthorized access to data. This section examines these attacks and shows what you can do to block them. Many XML firewall products block these attacks. The following sections describe what these attacks are, what the countermeasures are, and how you can ensure that your XML Firewall supports them.

Prevent Web Services SQL Injection

SQL Injection attacks involve the insertion of SQL statements into web forms in order to force a database to return inappropriate data or to produce an error that reveals database access information. A web services SQL Injection attack works on the same principle, except that it involves appending SQL data to parameters inside a SOAP message, in the hope that the SQL will be interpreted by a back-end database.

A successful SQL Injection attack requires two factors to be in place:

- Data received from a network connection is inserted directly into a SQL statement.
- The SQL statement is run in the context of a user with sufficient privileges to execute the attack.

The following is an example:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body
<BookLookup:searchByIBSN
xmlns:BookLookup="https://www.books.com/Lookup">
<BookLookup:IBSN>0072224711<BookLookup:IBSN>
</BookLookup:searchByIBSN>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

This message is processed by the following VB.NET code, which inserts the content of the ISBN element into a SQL statement:

```
Set myRecordset = myConnection.execute("SELECT * FROM myBooksTable WHERE
ISBN ='" & ISBN_Element_Text & "'")
```

In the case of the preceding SOAP message, this becomes

```
SELECT * FROM myBooksTable WHERE IBSN = '0072224711'
```

Now consider what happens when the following SOAP message is received:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header></SOAP-ENV:Header>
<SOAP-ENV:Body
<BookLookup:searchByIBSN
xmlns:BookLookup="https://www.books.com/Lookup">
<BookLookup:IBSN>0072224711'; exec master..xp_cmdshell 'net user Joe
pass /ADD'; --<BookLookup:IBSN>
</BookLookup:searchByIBSN>
</SOAP-ENV:Body></SOAP-ENV:Envelope>
```

In this case, the SQL statement will read

```
SELECT * FROM myBooksTable WHERE IBSN = '0072224711'; exec
master..xp_cmdshell 'net user Joe pass /ADD'; --
```

The code after the SELECT statement attempts to create a user called Joe with password of *pass*. An attacker could then attempt to use this new user account to gain access to the target machine.

To block this attack, ensure that data received from untrusted users is not directly placed into SQL statements. You can achieve this by using stored procedures rather than SQL statements built on-the-fly. SQL Injection can also be blocked by enforcing content-validation rules over incoming content. In this case, an XML security gateway would enforce a Schema-based rule, containing a regular expression (RegEx), such as the following:

<simpleType name=“ibsn”> <restriction base=“string”> <pattern value=“[0–9]{10}”/> </restriction> </simpleType>

This Schema would be validated against data isolated by the following XPath expression:

```
/Body/BookLookup:searchByIBSN/BookLookup:IBSN
```

In this case, `0072224711` passes the regular expression in the Schema, whereas `0072224711'; exec master..xp_cmdshell 'net user Joe pass /ADD'; --` fails.

It is likely that XPath Injection, which is analogous to SQL Injection, can be used to “harvest” information from an XML database. XPath Injection can be blocked by ensuring that data passed into an XPath expression does not itself contain XPath.

The use of SQL stored procedures is another good solution to the problem of SQL Injection. As a security professional, ensure that the developers in your organization are familiar with SQL Injection and are not implementing web services that are vulnerable to this attack. If you are not 100 percent assured that SQL interfaces will be implemented securely in your organization, then ensure that your XML firewall solution blocks SQL Injection by detecting SQL statements inside incoming XML messages.

Protect Web Services from Capture-Replay Attacks

Imagine this scenario: a web service is being protected by an XML gateway that scans incoming requests for X.509 certificates contained within SOAP messages and makes sure the messages are encrypted and signed. This system is vulnerable to a capture-replay attack, which captures a valid message and then submits it to gain unauthorized access.

TIP Do not confuse capture-replay attacks with "flooding" DoS attacks. Although both involve a message being replayed, the DoS attack is a brute-force attack designed to disable the target system, whereas the capture-replay attack is (arguably) a more clever attack that exploits a flaw in the target system's authentication scheme.

The solution to this problem involves the use of timestamps. WS-Security includes support for timestamps and, as you have seen in the WS-Policy example earlier in this chapter, the protected web service can mandate that a signed timestamp be present in incoming messages. A replayed message will include the same timestamp as the original message. If the message arrives a short time after the first message, both messages must be discarded, because it cannot be established which message is the original and which is the copy. This is why you must carefully decide the timestamp trust interval. It must be short enough that an attacker does not have time to capture, decrypt, and replay a valid message yet it must be long enough that slight discrepancies between the system clocks of the web service and the web service requester do not result in valid messages being blocked.

TIP Beware of any solution that claims "this is secure because all incoming messages are signed." Such a solution is a prime candidate for a capture-replay attack.

Ensure that Your Application Server Is Not Processing DTDs

The XML External Entity (XXE) attack takes advantage of the fact that outside data can be embedded into an XML document via a document type definition (DTD) entry. The following is an example of this type of DTD entry:

```
<!ENTITY name SYSTEM "URI">.
```

By specifying a URI that points to a local file, some XML engines could be made to access unauthorized information or to cause a denial of service by spending cycles "slurping up" large files from the local file system. Ensure that the XML platform used for your web services is not vulnerable to this attack. SOAP is not allowed to use DTDs, so a fully SOAP-complaint web services product *should* not be vulnerable to this attack. However, some SOAP-compliant web services products are vulnerable. Therefore, ensure that the SOAP stack used by your company's web services is not processing DTDs.

HEADS UP!

The XDoS attack, described next, proved that many vendors (including IBM and Microsoft) overlooked the facts of the SOAP requirement and processed DTDs anyway. In addition to ensuring that your web services do not process DTDs, you should not simply rely on a standard to ensure the security of an implementation. Although a standard may promote sound security practices, the *implementation* of the standard may, for whatever reason, produce a product that does not exactly follow the standard and thus may fall victim to attacks that exact implementation of the standard might have repulsed.

Apply Patches for XML Denial of Service Attacks

The XDoS attack is like the XXE attack because it also uses a feature of DTDs. However, the XDoS attack is based on the fact that entities defined in DTD can be pulled in. For example, HTML developers who display HTML source code in a web page are familiar with the requirement to use `<` and `>` instead of angle brackets so that the web browser does not parse the HTML source code. The use of the ampersand instructs the browser to look up `lt` and `gt` in the HTML DTD and replace them with whatever it finds. A similar technique is used for `£` to display the British pound sign.

Now, think about what happens if an entity is defined recursively. Suppose `&x100` is looked up in a DTD and it is found to be `&x99;&x99;` (i.e., the entity called `&x99;` appended to itself). The DTD parser must then look up `&x99;`. Suppose that it finds that it is `&x98;&x98;`. You can imagine the rest—the XML message explodes in memory (hence the term "XML bomb") and causes a denial of service. The entire XML bomb DTD is shown in the following code listing:

```
<!DOCTYPE foobar [
        <!ENTITY x0 "hello">
        <!ENTITY x1 "&x0;&x0;">
        <!ENTITY x2 "&x1;&x1;">
        <!ENTITY x3 "&x2;&x2;">
        <!ENTITY x4 "&x3;&x3;">
         ...
        <!ENTITY x98 "&x97;&x97;">
        <!ENTITY x99 "&x98;&x98;">
        <!ENTITY x100 "&x99;&x99;">
    ]>
    <foobar>&x100;</foobar>
```

As in the XXE attack, the fact that SOAP implementations are required by the SOAP specification *not* to process DTDs did not deter many SOAP vendors from including DTD processing in their products anyway, making them vulnerable to XdoS attacks. Patches

against the XDoS attack are available from Microsoft, IBM, Macromedia, Sybase, and others. Ensure that your company's SOAP implementation is patched against this attack, which can severely reduce the performance of a web services platform.

Do Not Blindly Process SOAP Attachments, Which May Harbor Viruses

SOAP messages may contain attachments, which may be threatening if they are very large and difficult to process (a "clogging attack") or if they harbor viruses. Attacks using SOAP attachments are somewhat mitigated by the fact that competition between two different specifications for SOAP attachments (one using Multipurpose Internet Mail Extensions, MIME, and the other Direct Internet Message Encapsulation, DIME) means that many organizations are eschewing the usage of SOAP attachments entirely until the Message Transmission Optimization Mechanism (MTOM) becomes a W3C recommendation. However, even if an organization chooses not to use SOAP attachments, it must ensure that its web services platform does not process attachments that the company unwittingly receives.

The solution to the security problem of SOAP attachments is to ensure that SOAP attachments are blocked entirely, filtered based on their MIME type, or passed through a virus scanner from a vendor such as McAfee or Symantec.

Ensure that You Are Not Vulnerable to an XML Signature Redirection Attack

An XML Signature includes a Reference element that points to the data that has been signed. To validate the XML Signature, the parsing application must "dereference" (i.e., pull down the content at) the reference URI. Consider what happens if the Reference element contains the following:

```
<dsig:Reference URI =
"http://ardownload.adobe.com
/pub/adobe/acrobatreader/win/5.x/5.1/AcroReader51_ENU_full.exe">
```

The XML Signature recommendation states that "XML signature applications MUST be able to parse URI syntax. We RECOMMEND they be able to dereference URIs in the HTTP scheme." (*XML-Signature Syntax and Processing*, W3C Recommendation 12 February 2002 [emphasis in the original text].) So, if an XML Signature processor follows the preceding URI reference, it must download a 20MB file and then compute the signature over it. This uses up the bandwidth of the target application and locks threads in a long wait for the 20MB file to download. An XML Signature dereference attack takes advantage of a web service that naively implements URI dereferencing. But an XML Signature engine for a SOAP application does not have to blindly download data from any URI in a Reference element, because the *XML-Signature Syntax and Processing* specification states the following:

> Note, there may be valid signatures that some signature applications are unable to validate. Reasons for this include failure to implement optional parts of this specification, inability or unwillingness to execute specified algorithms, or inability or unwillingness to dereference specified URIs (some URI schemes may cause undesirable side effects), etc.

So, vulnerability to this bandwidth-clogging attack is unnecessary. Ensure that the XML Signature protecting your web services is not vulnerable to this attack.

Implement a Policy to Secure Your Organization's Services-Oriented Architecture

Service-oriented architecture (SOA) and "software as services" are widely regarded as opportunities to ease integration problems and to improve business productivity. Although definitions vary, an SOA is basically a distributed-computing model that uses services as fundamental elements for developing applications. Services become the basic building blocks with which new applications are created. Web services are a prime example of such services. Although arguably SOA principles predated web services and could be implemented using CORBA, the invention of XML and web services has meant that SOA is within reach of many more organizations than before. Services implemented as web services in an SOA support composition into distributed applications. The services interoperate at run time to achieve the system's objectives.

Figure 5-2 shows the location of an SOA in an organization's IT infrastructure. An SOA places a services layer in front of back-office systems and is often implemented using an application server or a message bus. This services layer protects the developer from the complexity at lower layers and allows applications to be developed rapidly. SOAs are particularly suited to "dashboard" applications, which allow access to information that was previously "locked up" inside back-office systems. At the top left of Figure 5-2, you see such a "dashboard" for employees to access information through a portal.

Applying a security policy to the SOA in Figure 5-2 involves the following challenges:

- Persisting a "security context" from the end user or client to the business systems (i.e., from the top of the diagram to the bottom), so that you can control who accesses the web services
- Ensuring that an attacker sending malicious XML (such as an XDoS attack) is blocked at the services layer
- Blocking rogue XML messages that bypass the XML gateway

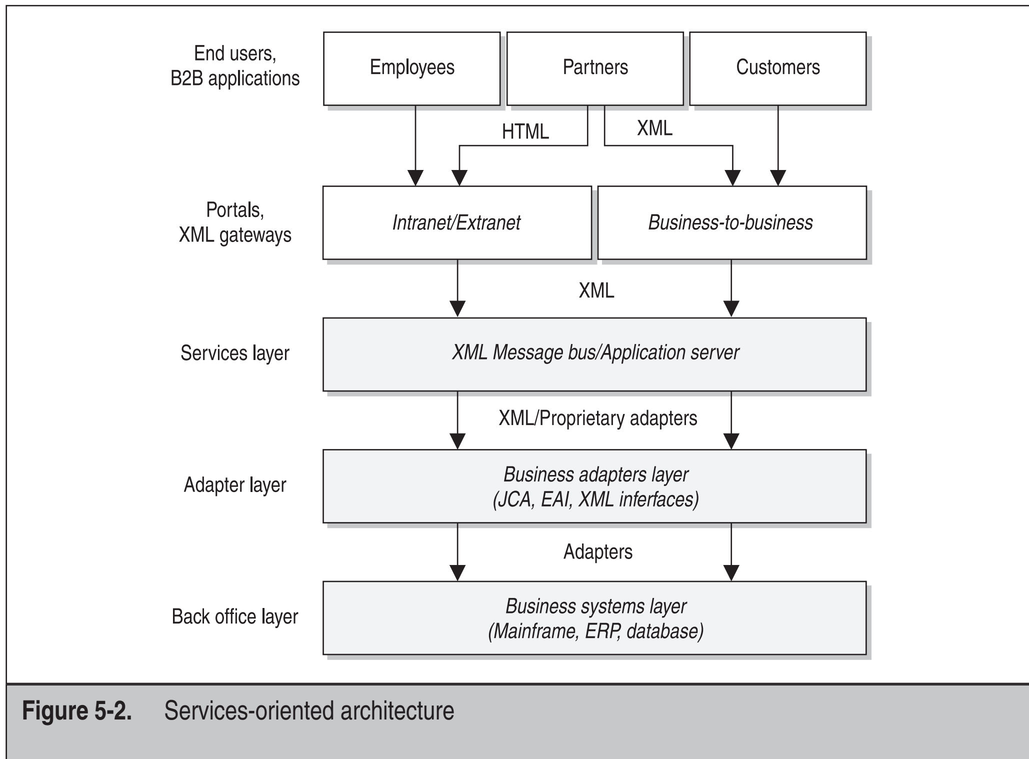

Figure 5-2. Services-oriented architecture

Use Security Tokens to Pass the Security Context to the Services Layer

At the systems layer, business applications may have security profiles that must be mapped to the end users or web service client applications. To avoid losing the client/user security context between the access layer (where the identity of the client or user is authenticated) and the services layer (where applications authorize and tailor responses to the end user or client), security tokens (such as an SAML assertion or WS-Security UsernameToken) should be "injected" into XML messages that pass to the services layer.

Deperimeterize Security

Wireless LANs, open VPNs, and other remote access options have produced a general breakdown in perimeter security models. A widespread use of malicious code, such as Trojan horses, contributes to the need for a new approach to security. The arrival of XML traffic at the services layer through an XML gateway cannot always be guaranteed. Therefore, you must "deperimeterize" security by enforcing security rules at the services layer, close to the web service endpoint. You can achieve this by inserting a "security agent" into the web services containers, to run in-process at the web service itself. Messages that have not arrived through the XML gateway may be routed back to the XML gateway

for processing, or they may simply be dropped. This caters to "deperimeterization" attacks where the attacker has bypassed the firewall.

When security is being enforced in mulitple places, it is important not to create duplicate stores of security policies and credentials. This duplication can be avoided by using a central XML Security Server to provide security services to all the applications that are applying security to XML documents. This is actually the web services model; rather than duplicate functionality in multiple applications, deploy it as a service and reuse it.

Ensure SOA Availability

Remember that availability and administration are aspects of security, too. Existing network management tools can be reused to aggregate alerts and quality-of-service (QoS) indicators generated at the services layer. Some existing network management tools now support the option to enforce QoS rules over web services. This is often called "web services management."

Review and Implement Products that Protect Web Services

Although web services are new technologies, a number of vendor products exist to protect them. This section examines three of them, each of which attacks the problem from a different angle.

Vordel—XML Gateway/Firewall and XML Security Server

Vordel's products perform both access control and content filtering for XML traffic. Access control is performed by using SSL, HTTP-Auth, WS-Security, SAML, and WS-Trust, as well as through adapters to the identity management infrastructure such as directories and web access control tools. XML content filtering is implemented by scanning XML messages for content-based threats such as executable attachments, malformed or invalid XML, unexpected MIME types in SOAP attachments, SQL Injection, or buffer-overflow attempts. Service scanning and brute-force "flooding" DoS attacks are also blocked. Blacklisting is performed by alerting upstream network firewalls and load balancers of the IP addresses of senders of malicious XML.

Vordel has two products. VordelSecure is an XML gateway/firewall, while VordelDirector is an XML security server that is designed to secure an SOA. Figures 5-3 and 5-4 show the Vordel Management Console.

In Figure 5-4, we see the pipeline of available security filters on the left-hand side of the screen. This is where the administrator configures the rules for XML messages that are being used by their organization. The rules are enforced either at the VordelSecure XML Gateway, or in software agents that are embedded into Web Services containers.

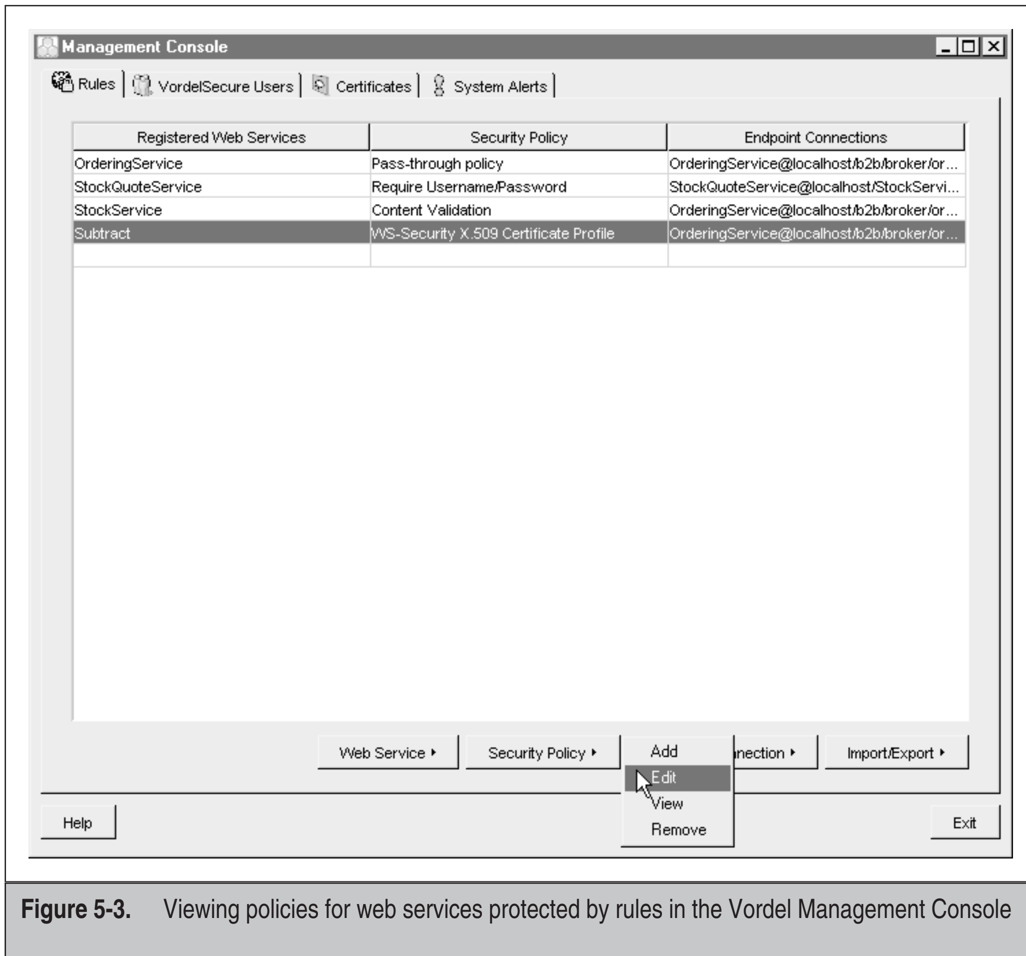

Figure 5-3. Viewing policies for web services protected by rules in the Vordel Management Console

Teros—Web Application Security

In the SQL Injection example in this chapter, you saw that web services security inherits a number of characteristics from web application security. In fact, when deployed on a web server, a web service may be seen as an example of a web application. Web application security products ensure that carelessly written web applications are not vulnerable to attack. Arguably, it would be better to simply educate developers to not develop web applications that expose vulnerabilities, but this is easier said than done. Only recently have books been published that educate developers about writing secure code.

Web application security products provide a "shield" in front of insecure web applications. One such application is the Teros Secure Application Gateway, a hardened security appliance that is deployed directly in the data path of application traffic and

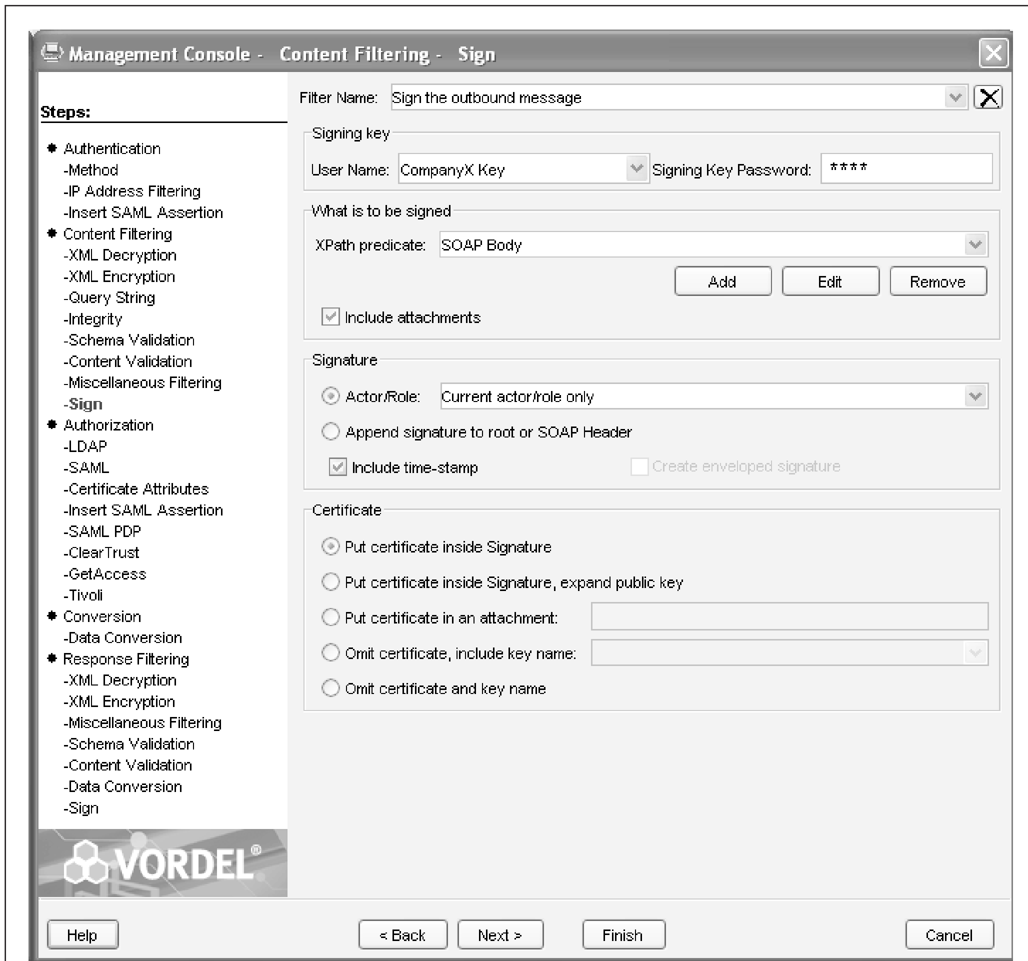

Figure 5-4. A client authentication rule in the Vordel Management Console

blocks application layer attacks that are not detected by network-based firewalls or intrusion detection systems. The Teros Secure Application Gateway permits only correct application behavior and blocks anomalous behavior, without the use of attack signatures. It includes support for the configuration of regular expressions over incoming XML traffic. In this chapter, you saw a regular expression in an XML schema definition being used to block a SQL Injection attack.

The Teros product works by ensuring that the content of incoming XML messages is appropriate. As such, it is analogous to a traditional firewall or application-level gateway,

which knows what valid traffic looks like and therefore can block invalid traffic. The product also includes a safeguard to ensure that sensitive data, such as credit card numbers, is not sent as output back to the client. As such, the Teros product is most suited to web services that are publicly available and therefore susceptible to attackers sending malicious XML. This is the same as the requirements for web application security, where, by definition, security products protect resources that are publicly available and therefore open to attack by being passed malicious data.

Sarvega—XML Router

Sarvega has pioneered the concept of the XML router. The Sarvega XML Router supports end-to-end reliable delivery of asynchronous messaging, with Gigabit Ethernet performance for XPath-based routing. For security, the XML Router performs fine-grained inspection, filtering, and transformation of SOAP message headers and content.

The Sarvega XML Router offers a mesh-based XML network topology with XML context routers deployed as peer edge devices, as well as a hierarchical network topology for scalable XML routing. The XML Router enforces QoS rules and supports delivery guarantees. The XML Router supports asynchronous (store and forward) messaging as well as synchronous messaging.

An XML router is suitable for situations where the primary requirement is to control XML data flows, based on the content of the XML data.