GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Hacker Tools, Techniques, and Incident Handling (Security 504)"
at http://www.giac.org/registration/gcih

Vulnerabilities in Web Services

by

William A. Shaffer

Option 1
GCIH Practical Assignment for
Hacker Techniques, Exploits, and Incident Handling
Version 2.1 (April 2002)

Table of Contents

Executive Summary

Web services are predicted to become a major capability of the Internet. Web services are allow systems, often in different enterprises, to communicate with one another using XML messages over HTTP. The linking of enterprise systems in this way may become one of the most significant developments for the Internet and for the economy in general.

Unfortunately, like the World Wide Web, Web services pose a variety of security problems because they allow access from the Internet into internal servers. They can therefore be attacked in a variety of ways. Because of the potentially large amount of commerce supported by Web services, it is vital that these services be as secure as possible.

This paper examines the vulnerability of one component that could be used in Web services, the Microsoft XML Core Services, formerly the Microsoft XML Parser, or MSXML. MSXML is used in various Microsoft products and is likely to be used in custom-developed Web services running on the Windows platform. XML is a rich, complex standard. Character-based, XML uses strings in a variety of ways. There are ample opportunities for the writer of an XML parser to allow buffer overflows. If the component is susceptible to a buffer overflow, it could provide an attacker a means to gain control of a server.

To test the robustness of MSXML, I performed 21 types of tests that gave some promise of exceeding array bounds in MSXML. Fortunately for Web service developers, MSXML proved quite robust and no buffer overflows were encountered.

The remainder of this paper is divided into three parts.

- Part 1, the **Exploit**, describes Web services and some of the security vulnerabilities that could occur. This part also summarizes vulnerabilities already identified in certain products used in Web services.

- Part 2, the **Attack**, describes in detail how the tests were performed to try to uncover buffer overflows. It outlines the analysis of the XML protocol to determine various roles for strings. It lists the results of the tests performed. The part describes how a buffer overflow attack might be launched if one had been found in MSXML. This part also makes some recommendations for protecting Web services against buffer overflow.

- Part 3, the **Incident Handling Process**, focuses on how the incident handling process might be adapted to handle attacks launched through malicious XML messages at Web services. Because of the potential financial impact of an attack on Web services, an organization should assume they will have incidents and should organize and prepare to handle these incidents. The nature of Web services provides some interesting variants on the incident handling process.

Emphasis is placed on instrumenting the Web service application so that potential attacks can be detected by the software and appropriate alerts issues.

The research for this paper suggests several recommendations.

I believe the MSXML component is reliable. Although I did not perform anywhere near all possible tests, the tests performed did indicate that the component was robust in the face of large or mal-formed XML documents. It also provided fairly detailed error messages.

Although it requires some extra time, enterprises should perform buffer overflow tests on both purchase components and custom developed software. In particular, developers of XML parsers should perform similar tests to the ones here to help insure that buffer overflows do not occur. XML developers are accustomed to test suites for conformance to the XML standards. Perhaps these suites could be expanded to include tests of robustness.

When developing Web services, developers should prepare a document type definition (DTD) or schema as a formal, precise specification of each XML message. Developers should consider whether the system performance will permit turning on XML validation in production. Validation provides various checks of the XML message's structure before the message data are passed to other components of the Web service. This validation could help protect other components from malicious attack.

When developing Web services, thought should be given to pre-screening the XML messages to avoid bogging down the parser and the rest of the service with overly long or complex messages intended to provide a denial of service attack.

Web services need to be instrumented to identify potentially malicious XML messages and provide alerts, since malicious messages may not be of consistent enough format to be recognized by signature-based intrusion detection systems. The applications also need to provide ways for response teams to back out unauthorized transactions entered into the system via malicious XML messages.

Enterprises deploying Web services need to assume that they will have incidents, and need to prepare properly for incident handling.

Part 1 – The Exploit

Overview

Web services have recently gained considerable attention as a mechanism for building distributed computer applications. Almost all major vendors of enterprise software, including Microsoft, IBM, and Oracle, have embraced Web services and are providing products. Like the World Wide Web, Web services create vulnerabilities by their very nature. First, Web services are often used to exchange data over public networks like the Internet. Therefore, these services are accessible by potential attackers. Second, an increasing amount of vital information is or will be exchanged via Web services. Disruption of the service or impairment of the integrity and confidentiality of the information could potentially be very costly.

This paper examines a possible exploit that could be used against certain applications of Web services: buffer overflows in an XML parser. This part of the paper also describes some other types of vulnerabilities of Web services.

Figure 1-1 below summarizes the exploits examined.

Figure 1-1: Summary of the Exploit

Name	XML Parser Buffer Overflow
Operating System	Windows NT 4.0, Windows 2000, Windows XP
Protocols/Services/Applications	Web Services (XML over HTTP) Microsoft XML Core Services 4.0 (MSXML)
Brief Description	Ability to gain access to, and control of, a server supporting Web services by exceeding the memory bounds of a storage area in an XML parser. An XML parser is a good candidate for such problems, since it must perform numerous string manipulations in a variety of contexts.
Variants	Attack against SOAP::Lite (this is not a buffer overflow, but is a situation where the supporting component does allow execution of arbitrary Perl subroutines, which could lead to an attacker controlling the server.

References	(There are no reported incidents of buffer overflows in MSXML.) The following references are relevant: "Microsoft XML Core Services 4.0" [MICR02], page that briefly describes MSXML and provides links to download the software development kit and the component. "Microsoft Security Bulletin MS02-008" [MICR02b], security bulletin describing a vulnerability in MSXML that allows a Web site to read files from a client computer running Internet Explorer. CAN-2002-0057 [MITR02] describing the above vulnerability in MSXML. "RPC without borders (surfing USA ...)" [STEA01], an article in *Phrack* describing vulnerabilities in Web services and in SOAP::Lite in particular.

What Are Web Services?

The term "Web Services" is used in narrow sense and broad sense. In the narrow sense, it represents a network service in which Extensible Markup Language (XML) messages are exchanged among computers over the Hypertext Transport Protocol (HTTP). In the broad sense, it is a philosophy of software architecture in which application software is divided into subsystems that communicate with each other using XML messages.

Web services provide a way to interconnect various computer systems and avoid the problems of "stovepipe" systems that have plagued enterprises in the past. "Stovepipe" systems are ones that cannot easily exchange data with other systems, thus making integration of systems costly or impossible.

Web services hold out the promise of interconnecting the systems of multiple enterprises to support better supply chain management and other business transactions. In this role, Web services could facility billions of dollars of commerce on the Internet. Many people expect business-to-business commerce to far exceed sales to consumers on the Internet. For example, Michael Hammer, in *The Agenda: What Every Business Must Do to Dominate the*

Decade, lists this interconnection of enterprises as one of eleven agenda items of high importance for businesses in the next decade. Much of this interconnection will be done with Web services.

Extensible Markup Language

Extensible Markup Language (XML) is a meta-language that allows developers to define markup to represent the structure of documents, messages, or other data. Through a mechanism called the document type definition (DTD) or a mechanism called schemas, a person or organization can define a set of "tags" that describe the content of the message. Tags typically consist of a word or term enclosed in angle brackets. A piece of content is enclosed between a beginning tag and an ending tag. An example of an XML document is shown in Figure 1-2. As the example shows, the data are always in character format as opposed to the binary format of many data exchange formats. The tags often provide valuable information about the data content.

Figure 1-2: An Example of an XML Document

```
<?xml version="1.0" ?>
  <!DOCTYPE Article ()>
<Article>
<Title>Test XML Document</Title>
 <Sect1>
<Title>Test of Processing Instructions</Title>
   <Para>
   <?xm-replace_text {Paragraph}?>
   </Para>
   </Sect1>
<LastModDate>Saturday, April 13, 2002 4:14:18 PM</LastModDate>
   </Article>
```

A developer can define a DTD to match the type of data that needs to be transported. Thus, XML does not specify any specific set of tags. Instead, the developer can specify a set of tags that she believes best describes her data. Discussion of XML's features is contained in Robert Eckstein, *XML Pocket Reference* [ECKS99] and Elliotte Rusty Harold and Scott Means, *XML in a Nutshell* [HARO01].

XML allows the creation of parsers that can pull apart messages and make the data available for applications. These components are reusable with any DTD. In the last three years, several XML parsers have been created and made available to developers at no cost. These include the Microsoft XML Core Services 4.0 [MICR02] and the Xerces XML Parser [APAC01]. These parsers are used in a number of commercial and custom-developed XML-based applications.

Simple Object Access Protocol

In the last few years, a number of organizations have developed specific applications of XML in an effort to provide standard message formats for exchanging information. One of these standards is the Simple Object Access Protocol (SOAP). The World Wide Web Consortium describes SOAP as:

> SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and HTTP Extension Framework. [W3C00]

Hypertext Transport Protocol

Hypertext Transport Protocol (HTTP) is a Transport Control Protocol/Internet Protocol (TCP/IP) based protocol widely used in the World Wide Web. A server accepting HTTP requests typically listens on TCP port 80 or TCP port 443 (for HTTP over secure socket layer), although other ports can be used.

Benefits of Web Services

Web services offer several potential benefits:

- XML offers the ability to represent a rich set of data structures for transmission.
- The data description capabilities of XML improve the understanding of the data exchange.
- The availability of low-cost and free XML parsers and other tools reduce the cost of development using XML and avoid the costs of developing custom parsers for other message structures.
- Organizations have experience using HTTP over the Internet.
- The Web services protocols can be used over the Internet to interconnect multiple enterprises.

Security Problems with Web Services

By its very nature, Web services pose a number of security problems. These problems are heightened when Web services are used to transact high dollar amounts of transactions.

- Web services create open ports (port 80, 443, or other port) that can be accessible from public networks and can be attacked remotely.
- The XML messages can be monitored by unauthorized persons. Confidentiality may be compromised.
- Web services could allow fraudulent or other unintended transactions to be executed. Authentication of messages may fail.
- Web services use relatively new software that could be compromised allowing the execution of unintended code on the servers supporting the Web services.
- Web services can be impaired by passing messages that cause excessive processing time (i.e. denial of service attack)
- Other problems can also occur.

A literature review turned up relatively few reported exploits against Web services. However, there is increasing concern about their vulnerability. Ravi Razdan writes in ZDNet:

> COMMENTARY--The hype surrounding Web services has reached crescendo proportions. That's not surprising given how eager some big information-technology companies are to find some sort of recurring, high-margin business in a down tech economy.
>
> But in their rush, an important data security issue is being ignored: Confidential information is vulnerable to malicious employees or hackers because customer data, which gets stored in applications or databases operated by the Web services provider, still exist in clear or unencrypted form. [RAZD02]

These vulnerabilities turned up in the literature search:

Name	Description
XML Core Services CAN-2002-0057 [MITR02] [MICR02a]	XMLHTTP control in Microsoft XML Core Services 2.6 and later does not properly handle IE Security Zone settings, which allows remote attackers to read arbitrary files by specifying a local file as an XML Data Source.
Oracle 9iAS SOAP Default Configuration Vulnerability [BUGT02]	It is possible for remote attackers to deploy and undeploy SOAP providers and services without valid credentials by default.
SOAP::Lite [STEA01]	SOAP::Lite allows execution of unauthorized Perl subroutines and can give an attacker access to a command shell on the server.

Part 2 – The Attack

Description and Diagram of the Network

Figure 2-1 depicts a simplified configuration for a Web service. At the left, an enterprise partner access the Web service through the Internet. The HTTP connection passes through a company's router/firewall to an application server. The software application runs on the application server and typically (but not necessarily always) talks to a database server.

Figure 2-1: Example of Web Service Hardware Configuration

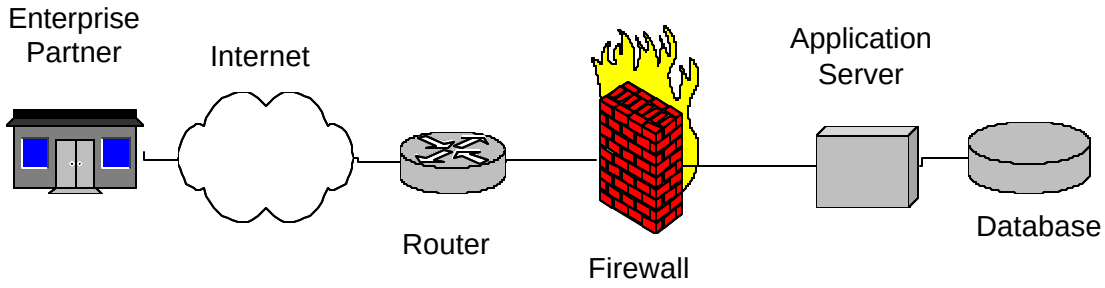
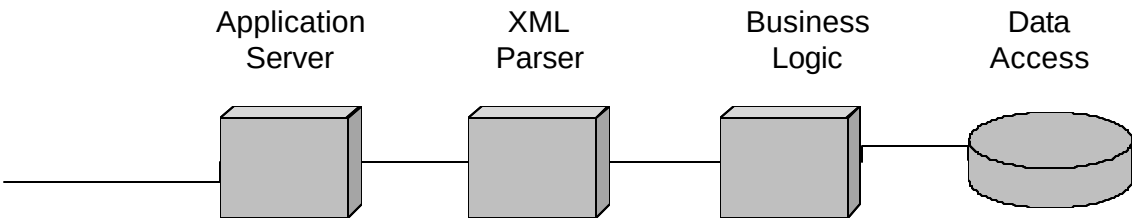


Figure 2-2 depicts a simplified view of the software configuration on the Application Server. The Application Server software, products like IBM WebSphere, BEA Weblogic, or similar software, manages the HTTP connection and passes along the XML message. Strickly speaking, many application servers operate along with a Web server that manages the HTTP connection. For the purposes here, the Web server is considered part of the application server. The message is then parsed by an XML parser that can separate the information in the message and make it available to the business logic. The business logic knows how to process this information, update the database, and return a response to the original sender.

Figure 2-2: Example of Web Services Software Configuration



Protocol Description

The exploit takes advantage of the fact that the XML message is passed through the enterprise fire wall to the application server. There, the XML message is configured to attack a weakness in the software processing the message.

XML Features

An XML message consists of strings that play many roles in the message. These roles include element tags, entities, processing instructions, and content. The table in Figure 2-3 lists many of these various roles.

Figure 2-3: The Many Roles of Strings in an XML Document

Name of Role	Example
In Primary Document	
Processing Instruction	<? Processing instruction here ?>
XML Instruction	<?xml version=”1.0” ?>
DOCTYPE Declaration	<!DOCTYPE ……. >
Public Identifier	<!DOCTYPE PUBLIC “-//public declaration//EN” >
System Idnetifier	<!DOCTYPE Article SYSTEM 'xmltest.dtd' >
Element	content/p>
Element Name	<ELEMENTNAME>
Content	CONTENT
Attributes	
Attribute Name	
Attribute Value	
Entity Reference	&entity;
In External Document Type Definition	
Element Declaration	<!ELEMENT para (#PCDATA) >
Attribute Declaration	<!ATTLIST para frame (yes|no) “yes” >
Entity Declaration	<!ENTITY name “abc” >
Parameter Entity Declaration	<!ENTITY % name “cde” >
Notations	<!NOTATION GIF89a SYSTEM “-//Compuserve//…” >
In Internal Document Type Definition Subset	(similar constructs to the External Document Type Definition)

The table is divided into three: the primary document, the external document type definition, and the internal document type definition subset. The primary document is the tagged content that follows the regular XML syntax. However, an XML document can reference an external DTD in the DOCTYPE declaration. The parser will then read and parse the DTD,

so that it can validate the primary document. The DTD follows a syntax that differs from the regular XML syntax, thus presenting more opportunities for buffer overflows. Besides an external DTD, an XML document can contain DTD declarations within the DOCTYPE declaration. These are called the internal DTD subset.

How the Exploit Could Work

Each role opens the possible opportunity to exploit a buffer overflow in the XML parser. Because there are so many different roles for strings, there are a number of places where buffer overflows could occur. Once a buffer overflow is found, an attacker can craft an XML message that could overwrite the pointers in the execution stack of the parser and could place hostile code for the parser to execute. As a result, the parser will start to execute the attacker's code. The code, in turn, could open up a backdoor for the attacker to use to further control the server.

The Document Object Model

The Document Object Model (DOM) is a standardized application programming interface for accessing and updating XML documents. When a program uses DOM, the parser loads the entire XML document into memory. Then the calling program can navigate through the contents of the document, moving from element to element.

Component to be Probed

The XML parser selected for probing is the Microsoft XML Core Services. This component was formerly called the Microsoft XML Parser or MSXML. We use the term MSXML to refer to the component in this paper. MSXML is used in a number of Microsoft products including Internet Explorer (Versions 6), Windows XP, and SQL Server 2000. Available as a free, separate download from the Microsoft site, MSXML is likely to be used in custom-built Web services constructed using Microsoft Component Object Model architecture. A vulnerability in this component could seriously compromise several Web-services implementations.

For testing purposes, I downloaded the most recent version which is Version 4.0 Service Pack 1. I then downloaded and applied the Microsoft Security Update Q317244 [MICR02b] which is listed as fixing the vulnerability allowing a Web site to access arbitrary files on a client machine, but which could fix other problems. The desire was to work with the most up-to-date and probably most reliable version.

Tactics for Probing MSXML

The following steps were used to probe MSXML:

- A list of various roles for strings were developed. (See Figure 2-3.)

- A list of anomalies for each role were developed. The focus was on testing strings that were uncommon in actual practice in XML. Anomalies include excessively long strings, strings incorrectly terminated, and excessive numbers of strings. Although XML files used for documents are often fairly big (1 megabyte or more), XML messages used for data exchange are usually much smaller. There are a number of typical styles observed in XML. Element tag names are usually ten or fewer characters. Similarly, attribute names are usually ten or fewer characters. The content of a title element is usually not 10 million characters. These common practices create the possibility that a large string or unusual XML structure may uncover defects in the parser. Therefore, the focus was testing MSXML with unusually long strings or an unusual number of elements.

- A simple DTD was developed to support the tests. This DTD is shown below:

```
<!-- ********************************************************************
        Name:                   XMLTEST.DTD
        Author:                 W. A. Shaffer
        Version:                1.0
        Description:

            This is a document type definition to test XML parsers for
            buffer overflows.                                          ->

<!--  ********************************************************************
                          Modification History

Date           Person          What Was Done
====           ======          =============
14-APR-2002    W.A.Shaffer    File created

********************************************************************** -->

<!-- ********************************************************************
                               Entities
********************************************************************** -->


<!-- ********************************************************************
                               Notations
********************************************************************** -->


<!-- ********************************************************************
                               Elements
********************************************************************** -->

<!--       ELEMENT       CONTENT                                       -->

<!ELEMENT  Article       (Title,Sect+)                                  >
```

```
<!ELEMENT  Title         (#PCDATA)                                  >
<!ELEMENT  Sect          (Header, Para+, Sect*)                     >
<!ELEMENT  Header        (#PCDATA)                                  >
<!ATTLIST  Header
           ATTR           CDATA   "0"                               >
<!ELEMENT  Para          (#PCDATA|Emp)*                             >
<!ELEMENT  Emp           (#PCDATA)                                  >
```

- A C program was constructed to produce XML files with various anomalies. (See Appendix A.)

- A Jscript script was written and included in an HTML page and executed from Internet Explorer (Version 6). Although this is not a Web services environment, the script produces a similar effect by loading an XML message into the MSXML parser. This is the HTML file and script.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
      <head>
            <title></title>
            <meta  name="GENERATOR"  content="Microsoft  Visual  Studio.NET
7.0">
            <meta                                      name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
      </head>
      <body>
            <script language="jscript">
      var xml = new ActiveXObject("Msxml2.DOMDocument.4.0");
      xml.async = false;
      xml.validateOnParse = false;
      var isload = xml.load("sample2.xml");
      if (isload)
            {
            var err = xml.validate();
            if (err.errorCode != 0)
                  {
                  alert(err.reason);
                  }
            //alert(xml.xml);
            alert("Load is Completed");
            }
      else
            {
            alert(xml.parseError.errorCode+xml.parseError.reason);
            }
            </script>
      </body>
</html>
```

In the above script, the line

```
var xml = new ActiveXObject("Msxml2.DOMDocument.4.0");
```

loads the MSXML parser and creates an empty DOM object. The line

```
var isload = xml.load("sample2.xml");
```

causes the parser to load the contents of the XML file and populate the DOM object. If the loading is successful, the parser is told to validate the file against the DTD and then reports "Load is Completed" in a dialog box. If the loading is not successful, the script reports the error from the parser.

- The C program is modified to generate various XML files that are then loaded into MSXML using Internet Explorer.. The execution is checked for the execution of an illegal operation.

- If a buffer overflow did appear, then an attacker could possibly use the vulnerability to execute code on the server. There was no attempt to develop such code.

Findings

MSXML proved to be quite robust and handled a wide variety of anomalous files. The table in Figure 2-4 lists the types of probes attempted. As the table shows, no buffer overflows were uncovered.

Figure 2-4: Table of Results in MSXML Probe

XML Feature	Anomaly	Result
DOCTYPE Public Identifier	Public identifier is 2000 characters long.	Handled normally as part of document.
DOCTYPE System Identifier	System identifier is 2000 characters long	Parser correctly reports error that syntax is incorrect. (System identifier needs to be a legal URI).
DTD External DTD	DTD is actually a file with binary numbers not characters.	Parser correctly reports illegal character in DTD
DTD Element declaration	Name of element is over 1000 characters long.	Handled as legal.

XML Feature	Anomaly	Result
DTD Entity declaraction	Name of entity is over 1000 characters long	Handled as legal
Attribute	Name of attribute is 2000 characters long	Handled without error (validation turned off)
Attribute	Value of attribute is 2000	Handled without error
Attribute	2000 character attribute value is not terminated with quote.	Parser correctly reports the < (character starting the next element) cannot be in an attribute value
Element Content	Mixed content contains 10,000 embedded tags. (Mixed content exists when an element contains both parsed character data and other elements interspersed.)	Handled without error.
Element Content	Title element contained 10 million characters.	Handled without error.
Element Content	Overlapping Tags (one element starts before another is ended)	Parser correctly reports an error.
Element Content	Elements are nested 10,000 levels deeps	Handled without error.
Element Tag	Begin tag is not terminated with >	Parser correctly reports an error that an element name contained an illegal character.
Element Tag Name	Element name is 5 million characters.	Handled without error.
Element Tag	32000 Article (top level) elements in XML document	Parser correctly reports that there are multiple top level elements in document.
Entity Reference	10,000 character entity reference.	Parser correctly reports that entity is not recognized, since the entity is not defined in the DTD.

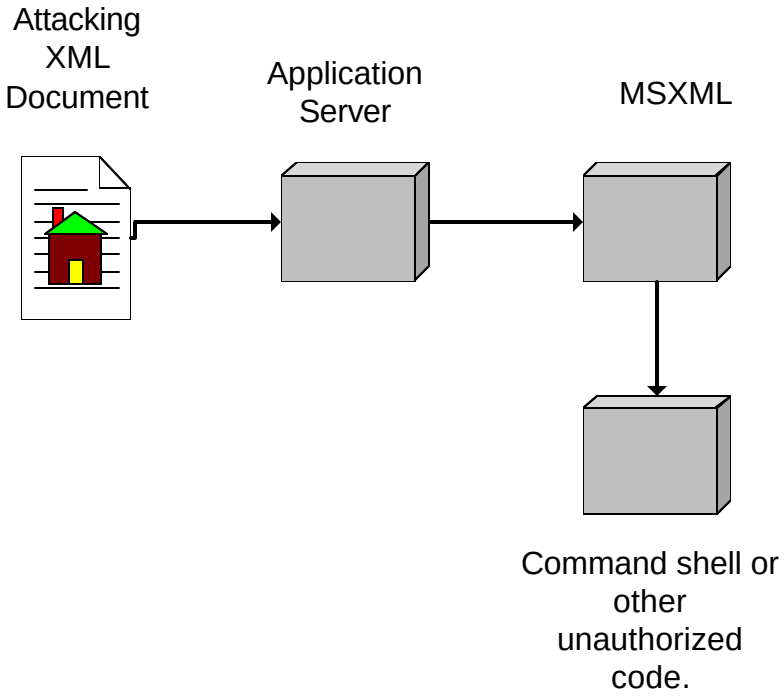
XML Feature	Anomaly	Result
Entity Reference	The semi-colon terminator on a 10,000 character entity reference is omitted.	Parser recognizes the end of the entity anyway and correctly reports that entity is not recognized, since the entity is not defined in the DTD.
XML Instruction	2000 character encoding attribute	Parser correctly reports an error that encoding attribute is not a valid value.
XML Instruction	Non-standard attribute is added with 200 character value.	Parser correctly reports that the attribute is illegal.
XML Instruction	Standalone attribute is given value with 6000 characters.	Parser correctly reports that the attribute must have a value of "yes" or "no".
XML Instruction	Version number is given various values other than "1.0".	Parser correctly reports error that Version number is incorrect. Only the text "1.0" is legal.

In none of the tests did the Parser exhibit a buffer overflow or fail to perform as required.

Description of the Attack

Buffer Overflow. If a buffer overflow had been found, an attack could be launched against a Web service using MSXML as shown in Figure 2-5.

Figure 2-5: Diagram of Web Services Attack in the Wild



In the diagram, the attacker develops an XML document that could cause a buffer overflow and contains code that could be executed. The document is passed to the application server via HTTP and the PUT operation. The application server passes the document to the application code (not shown) that invokes the MSXML parser. The parser attempts to read the document and suffers the buffer overflow. The malicious code in the document gets control and invokes a command shell or other unauthorized code that can then be manipulated to open a backdoor to allow the attacker to access the system through another route.

Denial of Service. One result of the MSXML tests showed that the parser can take a long time to process a complex document. Web services can be vulnerable to sending a number of complex documents, tying up the server as it spends its resources parsing the documents.

Signature of Attacks

Since the attacks are predicated on various sizes of strings in the XML document and may have various patterns, no single set of signatures would guard against an attack. If a particular automated attack were used, it is likely to have a consistent message that an Intrusion Detection System could detect. If the messages are developed for a specific attack, the format of the actual attacking message could vary widely.

How to Protect Against the Attack

For the Web Services Operator. An important defense against attack messages is to formally define legal input messages. XML has two levels of validations: well-formed and validated. A well-formed message obeys the rules for nesting of tags and a few other rules, but is not validated against a document type definition or schema. A validated message is well-formed and also is validated against a document type definition or schema.

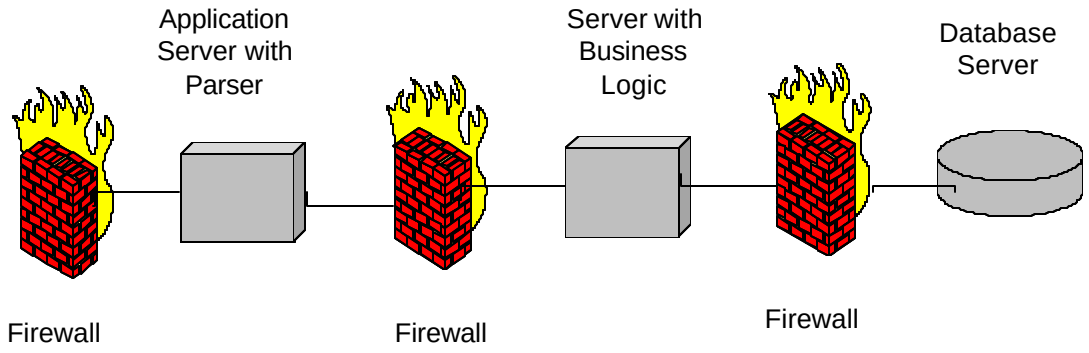
It is common practice to use XML messages without formally defining the DTD. This can be dangerous, since the developers may not have the benefit of a precise definition of a valid message. This makes it easier to produce code that fails to handle all the correct messages and may not report errors properly on invalid messages.

An interesting question is whether validation should be turned on when the XML message is loaded into MSXML. MSXML has an option that provides for validation against the DTD upon loading. Validation requires more computing resources. However, the parser will stop processing the message once it finds an invalid part, freeing up resources for processing other messages. Testing with specific Web services should reveal whether validation of good messages is justified to prevent excessive time spent on bad messages.

Another benefit of validating the XML message is that subsequent processing code can then depend on the DOM object being properly formed. Determining if an XML message has a correct structure is not a trivial task, but it is a task that validating parsers do well. If the parser passes along an invalid message, the incorrect DOM object could cause the business logic code to fail.

One protection that should be considered is to perform a rapid pre-processing of the message to check for obviously invalid messages. For example, the message could be scanned quickly for a count of tags and an overall number of characters. Messages that are obviously too long or have too many tags could be shuttled to one side and not parsed, thus avoiding a denial of service attack caused by the parser working too long to parse bad messages.

Figure 2-6: Using Extra Servers to Protect Against Attacker Gaining Access on Application Server



Another protective measure is to harden the exposed application server. Figure 2-6 depicts additional hardening that the Web service operator can perform. In this diagram, the code with the business logic and the database are placed on separate servers, following a three-tiered architecture. The application server along with the XML parsing code runs on a "bastion" host that offers few resources to an attacker who gains access to a shell on the server. Firewalls separate each of the servers. The connection from the application server to the server with the business logic could be accomplished using a protocol like the Common Object Request Broker Architecture (CORBA) which does not depend on XML or Web Services. The connection from the Business Logic Server to the Database Server could use SQL*Net or other protocol for database connection. The firewalls block all ports except those used by the specific connection protocols and any other protocols used for managing the server. The goal is to provide few computing resources for the successful attacker and to be able to restore the application server quickly.

For the Product Developer. The probes of MSXML indicated that the component developer had taken considerable care to make the parser robust. Not only did it not execute illegal operations when subjected to illegal and unusual XML, but the parser was able to handle much larger XML messages than are required.

However, Web services use a variety of components. Developers and purchasers of these components would benefit from performing the kinds of tests to which MSXML was subjected to see if there are buffer overflows. More important, developers need to follow best practices to avoid buffer overflows, since it is costly to perform extensive testing and such testing is unlikely to find every vulnerability. In *Writing Secure Code,* Howard and LeBlanc [HOWA02] outline several practices that developers need to follow. These include:

- Avoiding unsafe functions like *strcpy* (string copy) and *sprintf* in the programming language C.
- Avoiding confusion over character sizes when ANSI and Unicode characters are involved.

- Avoiding variables in place of constants for print formats.
- Using proper checks to insure that the bounds of arrays are not exceeded.
- Rigorous check of inputs to avoid unexpected inputs from overrunning storage areas.
- Using tools like StackGuard to assist in checking for buffer overflow.

Part 3 – The Incident Handling Process

Operators of Web services need to expect to respond to incidents. The enterprise that is dependent on the operation of one or more Web services to receive revenue or manage logistics must be able to respond rapidly to an attack. The possibility that large amounts of money could be involved means that the enterprise must restore operations rapidly while at the same time, it must be able to determine the cause and prevention of the attack. This section outlines steps that operators of Web services should take to handle attacks launched through the submission of XML messages to the Web service. The incident handling process should follow six stages:

- Preparation
- Identification
- Containment
- Eradication
- Recovery
- Lessons Learned
-

Preparation

Because of the need to respond rapidly to incidents and because of the possibility of large financial impact, preparation is probably the most important stage of the incident response. Preparation should include these items:

Organization. A written plan should set out who participates on the incident response team. Responsibilities should be described and assigned. Phone lists and contact procedures should be established.

Policies. Policies should be discussed and documented. The policies should address what constitutes an incident, escalation procedures, and when law enforcement should be contacted.

Contacts. The response team should have the names and phone numbers of appropriate people to contact. These people should include the appropriate technical support at the enterprises network provider and possible law enforcement contacts.

Equipment. Because of the possibility that a server could be disabled or seized as evidence, the enterprise should have available standby servers that could quickly be brought on line.

Forensic Tools. The enterprise should acquire extra disk drives, forensic software tools like Coroner's Toolkit, disk duplication equipment, and other tools needed to perform forensics. Also important is the establishment of properly protected logs for the various components. The application server logs should be stored in a protected manner and possibly copied automatically to a protected device. Particularly important is logging each incoming XML message (possibly on write-once media) so that the messages can be analyzed.

Test Plans and Scripts. The response team needs to have ready access to test plans, test scripts, and test equipment, so that if the Web service has to be rebuilt or modified, it can be readily tested before being put in production. By their nature, Web services lend themselves to automated testing. A set of automated tests should be developed for each Web service as part of the development processes.

Practice. The response team should practice the incident response plan and should be trained to be proficient in using the forensic tools, reading and understanding the logs and messages, and maintaining chain of custody for evidence. The system administration staff should practice restoring systems to operation and bringing the stand-by servers into operation.

Identification

If the enterprise operates a variety of Web services and some of them have high volume s of transactions, identification of some kinds of attacks could be difficult. Because of the variations in an XML message, use of current, signature-based intrusion detection systems may not give warnings of attacks via the XML messages. Firewall and router logs will not give warnings, because the traffic looks like normal traffic intended for the Web service.

One approach is to instrument the Web service software to recognize anomalies. As noted above, XML messages could be pre-scanned to recognize unusually long or complex messages. The parser itself can provide warnins by validating messages. Because Web services are intended for communication among cooperating software processes, any invalid XML message should set off an alert for someone to investigate. The business logic software should also be instrumented to recognize unusual transactions and issue alerts. These checks should be tuned over time and software updated to reduce false positives and recognize additional behaviors that could indicate an attack.

Checking software should be designed to send a page or other timely notification when an anomaly is identified.

Containment

Once an anomaly is identified, the incident response team needs to determine what, if any containment action is required. As set of questions needs to be addressed:

- Does the anomaly indicate an attack?

- What is the XML message promoting the attack?
- What is the attacker's intention: gain access to the server, initiate an unauthorized transaction, perform a denial of service attack?
- Did the attacker gain access to the server? If so, through what mechanism?
- How far did the attacker get? Did he get past the Application Server? Did he penetrate the business logic server or the database server?
- If the intent was to perform an unauthorized transaction, did the transaction occur?
- If the attack is a denial of service, where can it be blocked?

Containment Process

Containment steps can vary widely based on the type of attack and how successful the attacker was.

First, if the attacker has gained control or executed an unauthorized transaction, the affected systems need to be taken off line and the disks duplicated. The original disks should be stored in evidence bags and the duplicated disks used for analysis and restoration of the service.

If the attacker was able to gain control of one or more servers, the entire Web service may need to be taken off-line until the damage and countermeasures can be assessed. Management needs to make a decision about whether to bring the standby servers on-line. Doing so will reestablish the Web service on clean servers, but the attacker may be able to successfully repeat his attack to gain access on the new servers.

If the attacker is attempting to perform an unauthorized transaction, then the transaction can be backed out and the system closely monitored for new attempts. Hopefully, the designers of the application have provided appropriate ways to recognize transactions and back them out.

If the attacker is attempting a denial of service, it may be possible to block him at the router or firewall in front of the application server or at the network service provider's router.

Once the initial containment is accomplished, the attack needs to be analyzed. If the attack is through the XML message, the message needs to be retrieved and examined. Hopefully, all XML messages have been logged.

Jump Kit

The response team should have jump kits prepared ahead of time. Besides the usual contents for the jump kit (log books, tape recorder, extra disk drives, operating system media, pens, forensic software, evidence bags), the jump kit for Web services should contain software for scanning large amounts of text and viewing and validating XML:

- Easily used text scanning tools like *grep* or similar utilities operable on the operating system.
- Powerful text manipulation software like *Perl.*
- A validating XML parser and editor like *XMetal*.
- Document type definitions for all XML messages involved with the Web service.

These tools will be useful for searching through the XML messages and analyzing individual messages.

Eradication

If the XML message reveals the mechanism of the attack, such as a buffer overflow or unauthorized transaction, then the response team and system administrators need to take steps to eliminate the vulnerability and restore service. A complete fix may require updates from the vendors of any vulnerable components. These vendors should be contacted with requests for such updates.

Recovery

However, it is unlikely that the enterprise can wait for the fixes. The vulnerability can be blocked in several ways:

- The attacker's IP address can be blocked at the router or firewall in front of the application server. (However, the attacker may be able to easily attack from a different IP address, so this approach is not very secure.)
- The signature of the XML message can be checked and diverted before the message reaches the vulnerable component. This may require modification of the custom-developed software.
- The vulnerable component may be capable of being reconfigured to eliminate the vulnerability.

Following any changes to the configuration or code of the Web service, the response team should test the service using the scripts and test cases developed for this service.

When the response team and management determine that the Web service can safely be brought back up, the standby servers can be activated, or, if the original servers were not penetrated, the original servers can be restarted using the copies of the disks made in the containment stage. The original disks should be maintained in evidence bags and chain of custody maintained.

Additional investigation may be conducted using the copies of the disk and other servers.

Lessons Learned

Soon after recovery, the response team should conduct a Lessons Learned review, probably involving some of the software development team. Among the items that may be reviewed are:

- Coding practices that created vulnerabilities
- Ease with which transactions could be located and, if necessary, backed out
- Practices that would improve detection of anomalies
- Components that proved vulnerable. Can the vendor easily repair them? Can they be easily replaced by more robust components? Should they be avoided in future development?
- Ease of use of the test cases. Can they be installed and operated quickly? Are they automated as much as they should be? Do they provide adequate coverage?

References

[APAC01] Apache XML Project, "Welcome to the Apache XML Project", 2001, http://xml.apache.org/

[BUGT02] BugTrack, "Oracle 9iAS SOAP Default Configuration Vulnerability" BugTrack ID: 4289 http://online.securityfocus.com/bid/4289

[CPAN01] CPAN, SOAP-Lite-0.52.zip, http://search.cpan.org/search?dist=SOAP-Lite

[ECKS99] Robert Eckstein, *XML Pocket Reference* (Sebastopol, CA: O'Reilly & Associates, Inc., 1999) ISBN: 1-56592-709-5

[HAMM01] Michael Hammer, *The Agenda: What Every Business Must Do to Dominate the Decade* (New York: Random House, 2001) ISBN: 0609609661

[HARO01] Elliotte Rusty Harold and W. Scott Means, *XML in a Nutshell* (Sebastopol, CA: O'Reilly & Associates, Inc., 2001) ISBN: 0-596-00058-8

[HOWA02] Michael Howard and David LeBlanc, *Writing Secure Code* (Redmond, WA: Microsoft Press, 2002) ISBN: 0-7356-1588-8

[MICR02] Microsoft Corp., "Microsoft XML Core Services 4.0," http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/766/msdncompositedoc.xml

[MICR02a] Microsoft Corp., "XMLHTTP Control Can Allow Access to Local Files," Microsoft Security Bulletin MS02-008, http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/MSDN-FILES/027/001/887/msdncompositedoc.xml

[MICR02b] Microsoft Corp., "Security Update, February 13, 2002," Q317244 http://www.microsoft.com/windows/ie/downloads/critical/q317244/default.asp

[MITR02] Mitre Corporation, "XMLHTTP control in Microsoft XML Core Services 2.6 and later," CAN-2002-0057, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0057

[RAZD02] Ravi Razdan, "Web services: Security nightmare?" ZDNet, March 25, 2002, http://www.anchordesk.co.uk/anchordesk/commentary/columns/0,2415,7111979,00.html

[STEA01] Stealth, "RPC without borders (surfing USA ...)", *Phrack*, Volume 0x0b, Issue 0x3a, Phile #0x09 of 0x0e, http://www.phrack.com/show.php?p=58&a=9

[W3C00] World Wide Web Consortium, "Simple Object Access Protocol (SOAP) 1.1", http://www.w3.org/TR/SOAP/

Appendix A – C Program to Generate XML Files

```
//---------------------------------------------------------------
//
//   Name:             xmlex.c
//
//   Description:
//
//      This program is used to generate various absurd XML documents
//      to test for buffer overflows.
//
//   Author:          William A. Shaffer
//
//   Date:                    13 April 2002
//
//   Revision History
//   ----------------
//
//   Author          Date           Description
//   ------       ----        -----------
//
//   WAS               13-APR-2002 File created
//
//   Example:
//
//       xmlex.exe >  file.xml
//
//---------------------------------------------------------------
//   Include Files
//---------------------------------------------------------------

#include <stdio.h>
#include <string.h>
#include <assert.h>

//---------------------------------------------------------------
//   Defines
//---------------------------------------------------------------


//---------------------------------------------------------------
//   Typedefs
//---------------------------------------------------------------


//---------------------------------------------------------------
//   Globals
//---------------------------------------------------------------


//---------------------------------------------------------------
//   Function Declarations
//---------------------------------------------------------------

int writeXMLInstruction(char * encoding, char * standalone);
int putString(char * string);
int writeProcessingInstruction(char * content);
int writeDOCTYPE(char * root, char * dtd);
int writeBody(void);
int writeInternalSubset(void);
int writeElementBeginTagEnd(void);
int writeElementBeginTagStart(char * name, int repeat);
int writeElementBegin(char * name, int repeat);
int writeElementEnd(char * name, int repeat);
int writeTitle(void);
int writeSections(int repeat);
```

```
int writeHeader(void);
int writePara(void);
int writeEmp(char * content, int repeat);
int writeAttribute(char * name, int namect, char * value, int valuect);
int writeEntity(char * name, int repeat);

//---------------------------------------------------------------
//
//   Name:        main
//
//   Description:
//
//    The main function invokes other functions to build the XML document.
//
//   Return:
//
//       int result -   0 - terminated correctly
//                      1 - error occurred
//
//   Arguments:
//
//       int argc       -  number of arguments passed in from command line
//       char *argv[]   -  arguments from the command line
//
//---------------------------------------------------------------------

int main(void)
   {

   int result;

   result = writeXMLInstruction("US-ASCII", "yes");
   if (result) result = writeDOCTYPE("Article", "xmltest.dtd");
   if (result) result = writeBody();
   return (!result);
   }

//---------------------------------------------------------------------
//
//   Name:        writeXMLInstruction
//
//   Description:
//
//       This function outputs the XML instruction in the form:
//       <?xml version="1.0" encoding="YYYYY" standalone="XX" ?>
//
//       This function should be called as the first function to output an XML document.
//
//       Note:  Input arguments are not checked on purpose to allow invalid values.
//
//   Return:
//
//       int result -   1 if no errors occurred
//                      0 if an error occurred
//
//   Arguments:
//
//       char * encoding   - The type of encoding for this document, e.g.
//                           US-ASCII
//       char * standalone - "yes" if the document uses a DTD
//                           "no" if no DTD will be mentioned in the DOCTYPE specification.
//
//   Example:
//
//       result = writeXMLInstruction("US-ASCII", "no");
//
//---------------------------------------------------------------------

int writeXMLInstruction(char * encoding, char * standalone)
```

```
   {

   int result;
   char buffer[32000];

   // build content of processing instruction
   strcpy(buffer, "xml version='1.0' ");
   strcat(buffer, "encoding='");
   strcat(buffer, encoding);
   strcat(buffer, "' standalone='");
   strcat(buffer, standalone);
   strcat(buffer, "'");

   // output processing instruction
   result = writeProcessingInstruction(buffer);
   return (result);
   }

//-----------------------------------------------------------------
//
//   Name:        writeProcessingInstruction
//
//   Description:
//
//       Write out the processing instruction in the form:
//       <?CONTENT ?>
//
//       Where CONTENT is the content of the processing instruction.
//
//   Return:
//
//       int result     -  1 if no output errors occurred
//                         0 error if output error occurred
//
//   Arguments:
//
//       char * content - content of the processing instruction.
//
//   Example:
//
//       result = writeProcessingInstruction("xml encoding='US-ASCII' standalone='no'");
//
//-----------------------------------------------------------------------

int writeProcessingInstruction(char * content)
   {

   int result;

   result = putString("<?");
   if (result) result = putString(content);
   if (result) result = putString("?>\n");
   return (result);
   }

//-----------------------------------------------------------------
//
//   Name:        putString
//
//   Description:
//
//       Write a string to standard output.
//
//   Return:
//
//       int result     -  1 if write had no errors
//                         0 if an error occurred
//
//   Arguments:
```

```
//
//       char * string  - a null-terminated string
//
//   Example:
//
//       result = putString("this is a string");
//
//-----------------------------------------------------------------------

int putString(char * string)
   {
   int result;
   int isError;

   if (string != NULL)
      {
      isError = 0;
      while (*string && !isError)
         {
         (void)putchar((int)*string);
         isError = ferror(stdout);
         string++;
         }
      if (isError)
         {
         perror("Error writing string to stdout");
         result = 0;
         }
      else result = 1;
      }
   else
      {
      perror("Null pointer in putString");
      result = 0;
      }
   return(result);
   }

//-----------------------------------------------------------------
//
//   Name:           writeDOCTYPE
//
//   Description:
//
//       Write the DOCTYPE instruction in the following format:
//
//       <!DOCTYPE root SYSTEM 'dtd' >
//
//       Note: for these tests, we will always use the SYSTEM variant.
//
//   Return:
//
//       int result     - 1 if output was produced without an error
//                        0 if an output error occurred
//
//   Arguments:
//
//       char * root    - a string with the name of the root element.
//       char * dtd     - a string with the URI of the document type definition file
//
//-----------------------------------------------------------------

int writeDOCTYPE(char * root, char * dtd)
   {

   int result;

   // output beginning of instruction
   result = putString("<!DOCTYPE ");
```

```
   // output name of root element
   if (result) result = putString(root);
   // output document type definition name
   if (result) result = putString("  SYSTEM '");
   if (result) result = putString(dtd);
   // put in internal subset of DTD
   if (result) result = putString("'\n   [");
   if (result) result = writeInternalSubset();
   if (result) result = putString("   ]>\n");
   return(result);
   }

//---------------------------------------------------------------
//
//   Name:        writeInternalSubset
//
//   Description:
//
//       Output the internal subset of the DTD.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:                 (none)
//
//-------------------------------------------------------------------

int writeInternalSubset(void)
   {

   int result;


   result = putString("<!ENTITY jones 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'>");
   return(result);
   }


//---------------------------------------------------------------
//
//   Name:             writeBody
//
//   Description:
//
//       Output the root element and any subelements.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:                      (none)
//
//-------------------------------------------------------------------

int writeBody(void)
   {
   int result;
   int count;

   // output the top level element

   result = 1;
   for(count = 0; count < 1; count++)
      {
      result = writeElementBegin("Article", 1);
      if (result) result = writeTitle();
```

```
      if (result) result = writeSections(1);
      if (result) result = writeElementEnd("Article", 1);
      }
   return(result);
   }

//---------------------------------------------------------------
//
//   Name:        writeElementBegin
//
//   Description:
//
//       Write the start tag for an element with no attributes.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       char * name    - a string with the name of the element
//       int repeat     - number of times name is repeated to form the actual
//                        element name.  Can be used to generate really long names.
//
//   Example:
//
//-------------------------------------------------------------------

int writeElementBegin(char * name, int repeat)
   {
   int result;

   result = writeElementBeginTagStart(name, repeat);
   if (result) result = writeElementBeginTagEnd();
   return(result);
   }

//---------------------------------------------------------------
//
//   Name:        writeElementBeginTagStart
//
//   Description:
//
//       Output the start of an element tag in the form:
//
//       <NAME
//
//       Where NAME is the name of the element tag.
//
//   Return:
//
//       int result  -  1 if output occurred without error

//                      0 if an error occurred
//
//   Arguments:
//
//       char * name    - the name of the element
//       int repeat     - the number of times to repeat the name to generate really long
//                        element names
//
//-------------------------------------------------------------------

int writeElementBeginTagStart(char * name, int repeat)
   {
   int result;
   int count;
```

```
   result = putString("<");
   for(count = 0; (count < repeat) && result; count++)
      {
      result = putString(name);
      }
   if (result) result = putString(" ");
   return(result);
   }

//-----------------------------------------------------------------
//
//   Name:           writeElementBeginTagEnd
//
//   Description:
//
//       Complete a beginning element tag by putting the > at the end.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:                      (none)
//
//-----------------------------------------------------------------

int writeElementBeginTagEnd(void)
   {
   int result;

   result = putString(" >\n");
   return(result);
   }

//-----------------------------------------------------------------
//
//   Name:              writeElementEnd
//
//   Description:
//
//       Output the end tag of an element in the form:
//
//       </NAME>
//
//       Where NAME is the name of the element.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       char * name    - name of element
//       int repeat     - number of times to repeat the name to generate a really long
//                        name.
//
//
//-----------------------------------------------------------------

int writeElementEnd(char * name, int repeat)
   {
   int result;
   int count;

   result = putString("</");
   for(count = 0; (count < repeat) && result; count++)
      {
      result = putString(name);
```

```
      }
   result = putString(">\n");
   return(result);
   }

//---------------------------------------------------------------
//
//   Name:           writeTitle
//
//   Description:
//
//       Output the title for the XMLTEST DTD in the form:
//
//       <Title>header info</Title>
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//   Arguments:          (none)
//
//---------------------------------------------------------------------

int writeTitle(void)
   {
   int result;
   int repeat = 10;
   int count;

   result = writeElementBegin("Title", 1);

   for(count = 0; (count < repeat) && result; count++)
      {
      result = putString("A");
      result = writeEntity("jones", 1000);
      }
   if (result) result = writeElementEnd("Title", 1);
   return(result);
   }

//---------------------------------------------------------------
//
//   Name:           writeSections
//
//   Description:
//
//       Output a number of sections.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       int repeat  - the number of sections nested in one another
//
//---------------------------------------------------------------------

int writeSections(int repeat)
   {
   int result;
   int numPara = 1;
   int count;
   int count1;

   result = 1;
   for(count = 0; (count < repeat) && result; count++)
      {
```

```
      result = writeElementBegin("Sect", 1);
      if (result) result = writeHeader();
      //
      //  write out a number of paragraphs
      //
      for(count1 = 0; (count1 < numPara) && result; count1++)
         {
         result = writePara();
         }
      }

   for(count = 0; (count < repeat) && result; count++)
      {
      result = writeElementEnd("Sect", 1);
      }
   return(result);
   }

//-----------------------------------------------------------------
//
//   Name:           writeHeader
//
//   Description:
//
//       Write out the header to a section.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:          (none)
//
//-----------------------------------------------------------------

int writeHeader(void)
   {
   int result;
   int textCount = 1;
   int count;

   result = writeElementBeginTagStart("Header", 1);
   if (result) result = writeAttribute("ATTR", 1, "E", 1);
   writeElementBeginTagEnd();
   for(count = 0; (count < textCount) && result; count++)
      {
      result = putString("B");
      }
   if (result) result = writeElementEnd("Header", 1);
   return(result);
   }

//-----------------------------------------------------------------
//
//   Name:           writePara
//

//   Description:
//
//       Output the paragraph with content.
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       (none)
```

```
//
//   Example:
//
//------------------------------------------------------------------

int writePara(void)
   {
   int result;
   int textCount = 10;
   int count;

   result = writeElementBegin("Para", 1);
   for(count = 0; (count < textCount) && result; count++)
      {
      result = putString("C");
      if (result) result = writeEmp("C", 1);
      if (result) result = putString("C");
      }
   if (result) result = writeElementEnd("Para", 1);
   return(result);
   }

//------------------------------------------------------------------
//
//   Name:        writeEmp
//
//   Description:
//
//       Output text with Emphasis tag in form:
//
//       <Emp>content</Emp>
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       char * content - content to be put in Emp tag
//       int repeat     - number of times the content is repeated in the file
//
//------------------------------------------------------------------

int writeEmp(char * content, int repeat)
   {
   int result;
   int count;

   result = writeElementBegin("Emp", 1);
   for(count = 0; (count < repeat) && result; count++)
      {
      result = putString(content);
      }
   if (result) result = writeElementEnd("Emp", 1);
   return(result);
   }

//------------------------------------------------------------------
//
//   Name:        writeAttribute
//
//   Description:
//
//       Output an attribute name and value pair.
//
//   Return:
//
//       int result  -  1 if output occurred without error
```

```
//                      0 if an error occurred
//
//   Arguments:
//
//       char * name    - name of attribute
//       int namect     - number of times name string is output to form actual name
//                        of attribute
//       char * value   - attribute value
//       int valuect    - number of times value string is output to form actual value

//
//---------------------------------------------------------------------

int writeAttribute(char * name, int namect, char * value, int valuect)
   {
   int result;
   int count;

   result = 1;
   for(count = 0; (count < namect) && result; count++)
      {
      result = putString(name);
      }
   if (result) result = putString("='");
   for (count = 0; (count < valuect) && result; count++)
      {
      result = putString(value);
      }
   if (result) result = putString("'");
   return(result);
   }

//---------------------------------------------------------------------
//
//   Name:        writeEntity
//
//   Description:
//
//       Output an entity in the form of &name;
//
//   Return:
//
//       int result  -  1 if output occurred without error
//                      0 if an error occurred
//
//   Arguments:
//
//       char * name - name of entity
//       int repeat  - number of times to repeat name to produce actual entity reference.
//
//---------------------------------------------------------------------

int writeEntity(char * name, int repeat)
   {
   int result;
   int count;

   result = putString("&");
   for(count = 0; (count < repeat) && result; count++)
      {
      result = putString(name);
      }
   if (result) result = putString(";");
   return(result);
   }
```