**Description**

XML External Entity Expansion (also referred to as XXE) attacks are used against applications that process XML inputs by exploiting XML external entity support. By supplying hostile XML input containing a specification of an external entity to a weakly configured XML parser, attackers may be able to view files on the application server filesystem, conduct denial-of-service attacks, and interact with any external or backend systems the application has access to.

XXE vulnerabilities occur when the widely used XML format (a protocol normally used to transmit data between the browser and the server) contains various potentially dangerous features. Due to the potential severity of XXE attacks and their ongoing prevalence, these attacks make an appearance on the [OWASP Top 10](https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A4-XML_External_Entities_%28XXE%29) list of web application security risks.

As with many of the vulnerabilities on this list, prevalence would markedly decrease with more comprehensive and continuously updated developer training.

**Impact**

XXE attacks can include conducting denial-of-service attacks and disclosing local files containing sensitive data such as passwords or private user data. As the attack occurs relative to the application processing the XML document, it can enable attackers to laterally traverse to other internal systems to potentially stage Server-Side Request Forgery (SSRF) attacks against unprotected internal services.

XML attacks have been understood for almost 20 years and yet even in recent years, powerhouses like Google and Facebook are known to have faced issues with these types of attacks. This serves as a stark reminder that chaos can occur (and take the form of potentially massive fines) simply due to misconfiguration and poorly implemented code.

**Scenarios**

The widespread use of XML as a data structure within many application areas ensures that the vulnerability persists throughout numerous applications in a variety of different flavours. At its most benign, XXE attacks can cause denial of service attacks, for example when an attacker has embedded entities within entities within entities, causing the memory of the XML parser to overload. The so called ''[Billion Laughs attack](https://en.wikipedia.org/wiki/Billion_laughs_attack)' shown below takes advantage of a Document Type Definition called *foo*, and an element called *bar*, replaced in this case with the name of a fine security training platform! Anytime &bar; is used, the XML parser replaces it with SecureFlag.

**Request**

POST http://www.vulnerableapp.com/xml HTTP/1.1

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE foo [

<!ELEMENT foo ANY>

<!ENTITY bar "SecureFlag ">

<!ENTITY t1 "&bar;&bar;">

<!ENTITY t2 "&t1;&t1;&t1;&t1;">

<!ENTITY t3 "&t2;&t2;&t2;&t2;&t2;">

]>

<foo>

Join &t3;

</foo>

**Response**

HTTP/1.0 200 OK

Join SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlagSecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag SecureFlag

Unfortunately, this attack is not limited solely to spread annoyance. The below example highlights the ramifications of not having defined XML entities in the XML document. As the XML External Entity Expansion name suggests, even external sources can offer up XML entities, thus enabling an attacker to make requests to a web server using a URI which, if configured to process external entities (and they are by default) will return the contents of a file on the system.

**Request**

POST http://www.vulnerableapp.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

(...)

An attacker could perform a Server-Side Request Forgery attack, pointing the URI to an external resources, such as an HTTP location. This, in turn, can be used to pivot and interact with any external or backend systems the application has access to.

**Request**

**POST** http://www.vulnerableapp.com/xml HTTP/1.1

**<?xml version="1.0" encoding="ISO-8859-1"?>**

**<!DOCTYPE foo [**

**<!ELEMENT foo ANY>**

**<!ENTITY xxe SYSTEM**

**"http://internal.vulnerableapp.com:8443">**

**]>**

<foo>

&xxe;

</foo>

**Response**

HTTP/1.0 200 OK

(.. result of the request to http://internal.vulnerableapp.com:8443 ...)

**Prevention**

Disabling the Document Type Definitions (DTDs) function will effectively prevent most attacks.

When possible, handling data using simpler formats like JSON is recommended. For almost a decade, JSON has been seen as preferable to the use of XML due to its lightweight syntax and newer construction.

Of course, exceptions exist to prove rules, and in cases where it is absolutely not possible to switch off DTDs within the business parameters nor use another format, the following measures must be applied by developers.

When the entire XML document is transmitted from an untrusted client, it's not usually possible to selectively validate or escape tainted data within the system identifier in the DTD. Therefore, the XML processor should be configured to use a local static DTD and disallow any declared DTD included in the XML document.

**Testing**

Verify that the application correctly restricts XML parsers to only use the most restrictive configuration possible and to ensure that unsafe features such as resolving external entities are disabled to prevent XML eXternal Entity (XXE) attacks.

* **OWASP ASVS**: [5.5.2, 5.5.3](https://github.com/OWASP/ASVS/releases/download/v4.0.2_release/OWASP.Application.Security.Verification.Standard.4.0.2-en.pdf)
* **OWASP Testing Guide**: [Testing for XML Injection](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/07-Testing_for_XML_Injection.html)

**.NET**

**Vulnerable example**

The following C# snippet parses the content of an XML file using the [XmlDocument](https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmldocument) class:

XmlDocument document = **new** XmlDocument();

document.XmlResolver = **new** XmlUrlResolver();

document.Load(model.MyFileInput.OpenReadStream());

Notice that a non-default resolver ([XmlUrlResolver](https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlurlresolver)) is used to resolve external entities during the parsing. The snippet parses XML data in an unsafe way and can be exploited to inject a specially forged XML to disclose the /etc/passwd system file referring its path as an external entity, for example:

**<!DOCTYPE d [<!ENTITY e SYSTEM "/etc/passwd">]>**<t>&e;</t>

**Prevention**

**XMLDocument**

In versions preceding .NET version 4.6, XMLDocument sets the XmlResolver property to URLResolver, allowing DTD parsing that could lead to an XML Entity Expansion attack. To prohibit this, the XmlResolver should be explicitly set to null.

Starting from .NET version 4.6, XmlDocument's XmlResolver is set to null, defaulting to a safe configuration. When parsing external entities is not needed, it is advisable to use the default behavior, i.e., by omitting the line:

*// Vulnerable setting*

document.XmlResolver = **new** XmlUrlResolver();

If instead the parsing of external entities is required by the semantics of the web application, it is possible to use an instance of [XmlSecureResolver](https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlsecureresolver) in place of [XmlUrlResolver](https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlurlresolver) as it provides a means to fine-tune the allowed external entities and their loading behavior.

**XMLReader**

In versions preceding .NET version 4.0, the XMLReader's ProhibitDtd property is used to determine if a DTD will be parsed. It is set to True by default, which throws an exception if a DTD is identified.

Starting from .NET version 4.0, it is possible to configure the DtdProcessing property to Ignore to ignore the DTD without throwing an exception.

**XmlTextReader**

In versions preceding .NET version 4.0, the XmlTextReader's ProhibitDtd property is used to determine if a DTD will be parsed. It is set to False by default, allowing dangerous parsing of DTD documents.

Starting from .NET version 4.0, it is possible to configure the DtdProcessing property to Prohibit to throw an exception if a DTD is identified or Ignore to ignore the DTD without throwing an exception.

**References**

[OWASP - XML External Entity (XXE) Processing](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing)  
[OWASP - XML External Entity Prevention Cheat Sheet](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.md)  
[GitHub - .NET XXE Learning Tests](https://github.com/samuel-knutson/dotnet-xxe-learning-tests)