



Relazione Progetto Internet Security: XML Injection

Rendo Salvatore 1000009510

Sommario

Introduzione	2
Cosa è Xml?	2
Tipologie di XML injection	3
Tag injection / User injection	3
Malformed Input injecton	4
External Entity injection attacks (XXE injection attacks).....	4
XML BOMBS	5
Difesa da questi tipi di attacchi	6
Test sulla difesa di un applicativo web	7
Software utilizzato	7
Prova di attacchi e rispettive risposte dai servizi	9
Risultati.....	12
Conclusioni.....	13
Bibliografia	13

Introduzione

Le vulnerabilità XML injection si verificano quando l'input dell'utente viene inserito in un documento XML lato server in modo non sicuro, i risultati di tale injection sono vari, è possibile interferire con la logica dell'applicazione, eseguire azioni non autorizzate o accedere a dati sensibili lato server.

Cosa è Xml?

XML è l'acronimo di eXtensible Markup Language, utilizzato per rappresentare oggetti di dati strutturati come testo leggibile dall'uomo. XML è concepito come un formato per l'archiviazione e la trasmissione dei dati, ed è estensibile in modo da poter essere personalizzato per qualsiasi applicazione definendo come i dati sono organizzati e rappresentati.

Un documento XML è un file di testo che contiene una serie di tag, attributi e testo secondo regole sintattiche ben definite.

```
<?xml version="1.0" encoding="UTF-8"?>
<reminder>
    <to>you</to>
    <from>me</from>
    <body>Smile!</body>
</reminder>
```

Il pezzo di codice soprastante rappresenta un elemento “reminder” formato dagli elementi “to” “from” e “body”.

Le varie applicazioni, utilizzano un XML parser per leggere, convertire e modificare file e/o input di tipo XML, vedremo successivamente quanto il parser influisca alla prevenzione per i vari attacchi.

Tipologie di XML injection

Esistono vari tipi di XML injection ed hanno tutti come scopo quello di manipolare o compromettere la logica di un'applicazione e/o servizio XML, causando l'inserimento di contenuto malevolo all'interno dei documenti risultanti o l'accesso a dati sensibili.

Gli attacchi possono essere effettuati con un input valido o XML malformato.

Tipi:

1. Tag injection e/o User injection
2. Malformed Input injection
3. Entity expansion Attack o XML BOMBS
4. XXE injection Attacks (External Entity Injection Attacks)

I vari modi per proteggersi da questi tipi di attacchi verranno elencati nel capitolo successivo.

Tag injection / User injection

Considerando un documento XML del genere:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <uname>joepublic</uname>
    <pwd>r3g</pwd>
    <uid>0</uid>
    <mail>joepublic@example1.com</mail>
  </user>
  <user>
    <uname>janedoe</uname>
    <pwd>an0n</pwd>
    <uid>500</uid>
    <mail>janedoe@example2.com</mail>
  </user>
</users>
```

Un attacco di Tag/User injection da parte di un attaccante può essere effettuato qualora dovesse iniettare o dare in input i seguenti dati:

```
Username: alice
Password: iluvbob
E-mail:
alice@example3.com</mail></user><user><uname>Hacker</uname><pwd>l33tist</pwd><uid>0</uid><mail>hacker@exmaple_evil.net</mail>
```

In questo esempio, l'attaccante, qualora l'input venga salvato all'interno di un file di login.xml, senza alcun controllo da parte del server, è in grado di inserire nel campo MAIL un altro user con User ID 0, tipicamente utilizzato da parte dell'admin di sistema.

In molti casi nelle applicazioni e servizi XML, la seconda istanza di un id sovrascrive la prima, risultando cioè nella creazione di un user 'Hacker' con privilegi di Amministratore.

Malformed Input injecton

Questo genere di attacchi ha come scopo quello di iniettare input appositamente creato per corrompere file e/o servizi XML.

Per Questa tipologia di attacchi vengono utilizzati metacaratteri quali < , > , < - - , & .

Vari esempi :

input	XML risultante
Username = foo<	<pre><user> <username>foo</username> <password>Un6R34kb!e</password> <userid>500</userid> <mail>s4tan@hell.com</mail> </user></pre>
Username = foo<!--	<pre><user> <username>foo<!--</username> <password>Un6R34kb!e</password> <userid>500</userid> <mail>s4tan@hell.com</mail> </user></pre>
Username = &foo	<pre><user> <username>&foo</username> <password>Un6R34kb!e</password> <userid>500</userid> <mail>s4tan@hell.com</mail> </user></pre>

Le parentesi angolate < ,> vengono utilizzate per l'apertura e chiusura dei tag, qualora non sanificate, l'inserimento di un tag di apertura senza quello di chiusura o viceversa rende il file risultante malformato.

La serie di caratteri <!--/--> stanno ad indicare l'inizio e la conclusione di commenti. Il secondo estratto di XML risulterebbe malformato.

Il carattere & nella sintassi XML viene usato per rappresentare le entità (descritte nel prossimo paragrafo), il format corretto per le entità è &symbol;. Il terzo XML risultante non è valido poiché &foo non termina con ; e l'entità &foo; non è definita.

External Entity injection attacks (XXE injection attacks)

Una feature di Xml che può essere utilizzata per degli attacchi sono le Entità Esterne.

Prima di vedere un esempio è opportuno spiegare i concetti di DTD ed Entità Esterne.

DTD sta per Document Type Definition, e serve per definire la struttura e gli attributi di un documento XML.

All'interno delle DTD possono essere definite le entità, shortcut per caratteri speciali che possono essere referenziati all'interno del documento XML.

Esistono nativamente, le Entità Esterne, delle Entità che fanno riferimento a dei file o URL dove sono definite.

Sono proprio loro ad essere utilizzate per questa tipologia di attacchi.

Un utilizzo non malevolo di una Entità esterna:

```
<!--?xml version="1.0" ?-->
<!DOCTYPE replace [<!ENTITY example "Doe"> ]>
<userInfo>
  <firstName>John</firstName>
  <lastName>&example;</lastName>
</userInfo>
```

Un esempio di Xml malevolo che sfrutta una entità esterna **&xxe;** per referenziare il file **/etc/passwd**

```
<?xml version="1.0" ?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<note>
<to>Alice</to>
<from>Bob</from>
<header>Sync Meeting</header>
<time>1200</time>
<body>Meeting time changed &xxe;</body>
</note>
```

Un'ipotetica risposta di un server debole a questi attacchi potrebbe essere:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=UTF-8
Server: *****
Date: Sat, 19 Apr 2021 13:08:49 GMT
Connection: close
Content-Length: 1039

Note saved! From Bob to Alice about "Sync Meeting" at 1200: Meeting
time has changed
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
sys:x:3:3:sys:/dev:/usr/sbin/nologin etc.....
```

XML BOMBS

Una XML Bomb è un input XML sia ben formato che valido, ma progettato per l'utilizzo eccessivo di risorse per mandare in crash l'applicazione o il suo XML parser.

Un famoso attacco di questo tipo è il Billion Laughs Attack:

```
<?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>
```

Quando il parser xml riceve questo documento, processa un elemento “lolz” che contiene un riferimento all’entità “lol9”, che una volta espansa fa riferimento a 10 “lol8” ed ogni “lol8” fa riferimento a 10 “lol7” e così via. Dopo l’espansione di tutte le entità, questo piccolo blocco di XML conterrà 10^9 = un miliardo di “lol”, utilizzando circa 3 gigabyte di memoria, questo attacco può essere sfruttato per un DoS attack.

Difesa da questi tipi di attacchi

Come abbiamo visto queste tipologie di attacchi possono essere molto pericolosi, possono intaccare le proprietà di Confidenzialità e Disponibilità di una applicazione.

Dipendentemente dalle funzionalità dell’applicazione e di come essa sia stata messa in sicurezza da questi attacchi, un attaccante potrebbe mal formare un file .xml o leggere file di sistema all’interno del server.

La prevenzione da questi tipi di attacchi può essere varia, generalmente è opportuno classificare qualsiasi tipo di input che riceve il servizio come un input potenzialmente malevolo e che quindi deve essere sanificato e controllato.

L’applicazione, soprattutto per gli attacchi di Tag injection e Malformed input injection, DEVE sanificare l’input ricevuto prima di salvarlo tramite un parser XML in un documento XML o mandarlo ad un altro servizio dell’applicazione.

Per le XML BOMS e XXE Attacks invece è opportuno disabilitare qualsiasi DTD inclusa nel documento ricevuto e se non necessarie, rimuovere l’opportunità di utilizzare le entità esterne e l’espansione delle entità, e quindi di configurare l’Xml parser in modo adeguato, qualora non sicuro di default, per non vulnerabile il servizio da questi tipi di attacchi.

La facilità con cui prevenire questo ultimo tipo di attacchi dipende molto dal linguaggio, dalle librerie e soprattutto dal XML parser utilizzato dal servizio, per esempio molti Parser XML di Java hanno le entità esterne abilitate di default, rendendo così anche framework per applicativi o servizi online come Springboot soggetti a XXE attacks o XML Bombs, qualora non vengano prese le opportune misure per prevenirli.

Test sulla difesa di un applicativo web

In questo capitolo vedremo i risultati di questi attacchi verso dei piccoli applicativi creati ad hoc che cercano di emulare il comportamento di un web service per il login in un'applicazione, sfruttando XML per lo scambio dei dati con il client ed immagazzinamento dati lato server.

Software utilizzato

Per questo Test ho utilizzato php 8.0 per creare all'interno della mia macchina:

UN CLIENT: Hostato nella porta 7778:

```
1 <?php
2 $xml = <<<XML
3 <login>
4   <user>hack</user>
5   <pass>123</pass>
6   <mail>hack.com</mail>
7 </login>
8 XML;
9
10 // Sends XML to SAFE SERVER
11 $ch = curl_init();
12 curl_setopt($ch,CURLOPT_HEADER,0);
13 curl_setopt($ch,CURLOPT_RETURNTRANSFER,1);
14 curl_setopt($ch,CURLOPT_URL,"http://localhost:7777/safe-server.php");
15 curl_setopt($ch,CURLOPT_POST,1);
16
17 curl_setopt($ch,CURLOPT_POSTFIELDS, $xml);
18 $data = curl_exec($ch);
19
20 if(curl_errno($ch)){
21     print curl_error($ch);
22 }else{
23     echo "Response BY Safe Server: <br> ". $data;
24 }
25 curl_close($ch);
26
27 //Send XML to UNSAFE server
28
29 echo "<br><br><br>";
30 $ch = curl_init();
31 curl_setopt($ch,CURLOPT_HEADER,0);
32 curl_setopt($ch,CURLOPT_RETURNTRANSFER,1);
33 curl_setopt($ch,CURLOPT_URL,"http://localhost:7777/unsafe-server.php");
34 curl_setopt($ch,CURLOPT_POST,1);
35
36 curl_setopt($ch,CURLOPT_POSTFIELDS, $xml);
37 $data = curl_exec($ch);
38
39 if(curl_errno($ch)){
40     print curl_error($ch);
41 }else{
42     echo "Response BY UNSAFE Server: <br> ". $data;
43 }
44 curl_close($ch);
45
```

Manda varie tipologie di input Xml tramite curl verso due tipi diversi di applicativi creati e hostati in locale nella porta 7777, essi cercano di emulare un servizio web per il login ad una applicazione, prendendo in input un determinato documento XML per poi salvarlo all'interno di un file XML (che in questo piccolo test possiamo pensare come un Database di logins):

1) Un safe-server.php:

```

1 <?php
2 // SAFE SERVER against XXE: PHP
3
4 //Creates a variable for the XML input
5 $xmlfile = file_get_contents('php://input');
6 $dom = new DOMDocument();
7 try{
8     //Dom loading
9     $dom->loadXML($xmlfile);
10    $login = simplexml_import_dom($dom);
11
12    //Saves sanitized input XML in local variables
13    $user = filter_var($login->user,FILTER_SANITIZE_STRING);
14    $pass = filter_var($login->pass,FILTER_SANITIZE_STRING);
15    $mail = filter_var($login->mail,FILTER_SANITIZE_STRING);
16
17    //Xml file opening + check if all attributes were sent
18    $file = 'safe-server.xml';
19    if(!empty($user) && !empty($pass) && !empty($mail)){
20
21        $xml = simplexml_load_file($file);
22
23        //creates the new Xml object and appends it on file
24        $userxml= $xml->addChild('login');
25        $userxml->addChild('user', $user);
26        $userxml->addChild('pass', $pass);
27        $userxml->addChild('mail', $mail);
28
29        //saves the edited file
30        $xml->asXML($file);
31
32        //prints login
33        echo "You have logged in as user $user and mail $mail";
34    }else{
35        echo " BAD REQUEST";
36    }
37
38    //Check if the document is a valid XML document
39    $doc = new \DOMDocument();
40    $doc->Load($file);
41    if ($doc->validate()){
42        echo "<br><br>The XML file is valid";
43    }else{
44        echo "<br><br>The XML file is NOT valid : ERROR";
45    }
46
47    // Exception Handling
48 }catch (Exception $e){
49     console.log($e);
50     echo " Cannot load XML input, try using validate input";
51 }

```

2) Un unsafe-server.php

```

1 <?php
2 //enable External Entities
3 libxml_disable_entity_loader(false);
4 //Creates a variable for the XML input
5 $xmlfile = file_get_contents('php://input');
6 $dom = new DOMDocument();
7 try{
8     //Dom loading
9     $dom->loadXML($xmlfile, LIBXML_NOENT | LIBXML_DTDLOAD);
10    $login = simplexml_import_dom($dom);
11
12    //Saves input XML in local variables
13    $user = $login->user;
14    $pass = $login->pass;
15    $mail = $login->mail;
16
17    //Xml file opening + check if all attributes were sent
18    $file = 'unsafe-server.xml';
19    if(!empty($user) && !empty($pass) && !empty($mail)){
20
21        $xml = simplexml_load_file($file);
22
23        //creates the new Xml object and appends it on file
24        $userxml= $xml->addChild('login');
25        $userxml->addChild('user', $user);
26        $userxml->addChild('pass', $pass);
27        $userxml->addChild('mail', $mail);
28
29        //saves the edited file
30        $xml->asXML($file);
31
32        //prints login
33        echo "You have logged in as user $user and mail $mail";
34    }else{
35        echo " BAD REQUEST";
36    }
37
38    //Check if the document is a valid XML document
39    $doc = new \DOMDocument();
40    $doc->Load($file);
41    if ($doc->validate()){
42        echo "<br><br>The XML file is valid";
43    }else{
44        echo "<br><br>The XML file is NOT valid : ERROR";
45    }
46
47    // Exception Handling
48 }catch (Exception $e){
49     console.log($e);
50     echo " Cannot load XML input, try using validate input";
51 }

```

1) Il primo : safe-server.php:

- Non ha abilitato le entità esterne, le DTD esterne né l'espansione delle entità (infatti su php questo avviene di default).
- Sanifica l'input ricevuto, eliminando i vari metacaratteri non consentiti

2) Il secondo: unsafe-server.php:

- Ha le entità esterne abilitate tramite il codice:

```

//enable External Entities
libxml_disable_entity_loader(false);

```

- Ha le DTD esterne e l'espansione delle entità abilitate, settando i flag LIBXML_NOENT e LIBXML_DTDLOAD durante il loading dell'XML ricevuto dal client.

```

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

```


3) Per ultimo, è presente un altro piccolo servizio “logins.php” che cerca di emulare un servizio di login tramite form HTML dal quale, una volta compilato, il servizio crea un elemento XML “login” per poi salvarlo nel file “safe-login.xml”:

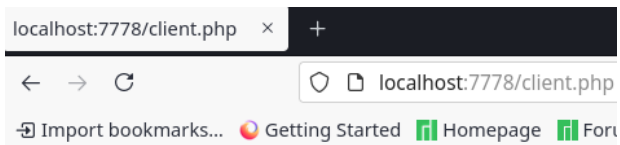
```
1 <?php
2 // LOGINS.PHP
3 //checks if user mail and pass exists.
4 if (isset($_POST) && isset($_POST['user']))
5     && isset($_POST['mail']) && isset($_POST['pass'])) {
6     //Saves sanitized input XML in local variables
7     $user = filter_var($_POST['user'], FILTER_SANITIZE_STRING);
8     $mail = filter_var($_POST['mail'], FILTER_SANITIZE_STRING);
9     $pass = filter_var($_POST['pass'], FILTER_SANITIZE_STRING);
10
11     // opens sample.txt and stores the recived sanitized data
12     $file = 'safe-login.xml';
13     if ($file){
14         $xml = simplexml_load_file($file);
15         $userxml = $xml->addChild('login');
16         $userxml->addChild('user', $user);
17         $userxml->addChild('pass', $pass);
18         $userxml->addChild('mail', $mail);
19
20         //saves the edited file
21         $xml->asXML($file);
22     }
23 }else{
24     echo " Fill Form Correctly";
25 }
26 ?>
27 <html>
28 <head>
29 <title>HTML data: Storing in XML</title>
30 </head>
31
32 <body>
33 <!-- Our page HTML form -->
34 <form action="" method="post">
35     User: <input type="text" name="user" id="">
36     <br><br>
37     pass:<input type="text" name="pass" id="">
38     <br><br>
39     mail:<input type="text" name="mail" id="">
40     <br>
41     <input type="submit" value="Submit">
42 </form>
43
44 </body>
45 </html>
```

```
46 <?php
47 //Check if the document is a valid XML document
48 $file = 'safe-login.xml';
49 $doc = new \DOMDocument();
50 $doc->Load($file);
51 if ($doc->validate()){
52     echo "<br><br>The XML file is valid";
53 }else{
54     echo "<br><br>The XML file is NOT valid : ERROR";
55 }
```

NOTA BENE: L’xml parser utilizzati sono: DOM: per estrapolare le informazioni dall’input dell’utente, simpleXML per salvare i dati ricevuti nei vari file di login.

Prova di attacchi e rispettive risposte dai servizi

Per primo proviamo un semplice login qualsiasi:



Response BY Safe Server:
You have logged in as user hack and mail hack.com

```
$xml = <<<XML
<login>
  <user>hack</user>
  <pass>123</pass>
  <mail>hack.com</mail>
</login>
XML;
```

The XML file is valid

Response BY UNSAFE Server:
You have logged in as user hack and mail hack.com

The XML file is valid

Come vediamo entrambi hanno risposto con il corretto login e dopo il salvataggio dell'XML ricevuto all'interno del file XML usato per salvare i login, il file è ancora valido.

Proviamo ora ad effettuare un XXE injection attack su questi due servizi trami il seguente XML payload:

```
$xml = <<<XML
<!DOCTYPE own [ <!ELEMENT own ANY >
<!ENTITY own SYSTEM "file:///etc/passwd" >]>
<login>
  <user>&own;</user>
  <pass> 123</pass>
  <mail>hack.com</mail>
</login>
XML;
```

Risposta Dei 2 servizi:

localhost:7778/client.php

← → ↻ 🛡️ 📄 localhost:7778/client.php ☆ 📌 ☰

🔗 Import bookmarks... 🌐 Getting Started 📁 Homepage 📁 Forum 📁 Wiki >> 📁 Other Bookmarks

Response BY Safe Server:
BAD REQUEST

The XML file is valid

Response BY UNSAFE Server:
You have logged in as user root: [REDACTED]

[REDACTED]

/etc/passwd DEL SERVER

[REDACTED] and mail hack.com

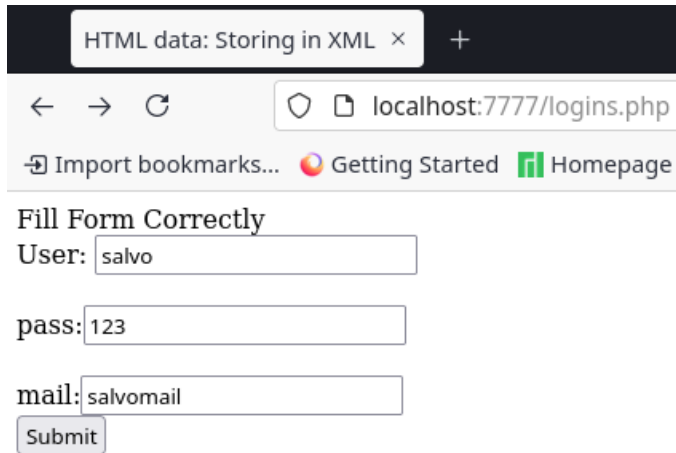
The XML file is valid

Come possiamo vedere, entrambi i file di login rimangono validi, ma il servizio debole a questi attacchi, salva e soprattutto mostra come input valido per il campo username il file /etc/passwd del server.

Questo attacco, pur non mostrando password, può essere utile per l'attaccante per ricavare informazioni alle quali non sarebbe dovuto accedere. (potrebbe leggere altri file di sistema come /etc/hosts ecc..).

Mentre il servizio "safe-server" ha semplicemente risposto BAD REQUEST e non ha nemmeno salvato i dati all'interno del file XML dei logins.

Proviamo ora invece ad effettuare un semplice login verso “logins.php”:



HTML data: Storing in XML x +

localhost:7777/logins.php

Import bookmarks... Getting Started Homepage

Fill Form Correctly

User:

pass:

mail:

Submit

login with username : salvo and mail : salvomail

Risposta del server:

Proviamo ora a fare l’injection della seguente stringa nel campo “mail:” :

```
ee</mail></login><login><user>Hackerthat cuold change id</user><pass>mine</pass><mail>hackergmail
```

In un sistema poco sicuro e senza controlli/sanificazione dell’input da parte dell’applicativo e/o del parser XML, questo codice inserirebbe un nuovologin con:

user: Hackerthat cuold change id

pass: mine

mail: hackergmail

La risposta del nostro sistema di login tramite form html è:

login with username : salvo and mail : eeHackerthat cuold change idminehackergmail

The XML file is valid

Come possiamo vedere accetta comunque l’input, ma lo sanifica dagli eventuali tag malevoli grazie alla funzione :

```
$user = filter_var($_POST['user'],FILTER_SANITIZE_STRING);  
$mail = filter_var($_POST['mail'],FILTER_SANITIZE_STRING);  
$pass = filter_var($_POST['pass'],FILTER_SANITIZE_STRING);
```

rimuove dai vari input ricevuti tutte le tipologie di caratteri non permessi, facendo rimanere valido il file XML dei login.

Per php è possibile, ed è raccomandato per tutti i linguaggi, creare filtri ad hoc o utilizzare filtri già presenti per la sanificazione dell’input utente.

Risultati

In questo caso siamo riusciti a ricreare il successo ed il fallimento di un attacco XXE injection ed un fallimento di un User injection.

Avendo creato i vari servizi con php, è stato relativamente semplice difendersi da queste tipologie di attacchi poiché i parser utilizzati sono di default sicuri contro questa tipologia di attacchi, ricordiamo però che questo varia da linguaggio a linguaggio e soprattutto dal XML parser scelto.

Segue la lista di vulnerabilità a XXE attacks o XML Bombs dei vari parser XML configurati con i setting di default dei seguenti linguaggi:

Python 3:

The following table gives an overview of various modules in Python 3 used for XML parsing and whether or not they are vulnerable.

Attack Type	sax	etree	minidom	pulldom	xmlrpc
Billion Laughs	Vulnerable	Vulnerable	Vulnerable	Vulnerable	Vulnerable
Quadratic Blowup	Vulnerable	Vulnerable	Vulnerable	Vulnerable	Vulnerable
External Entity Expansion	Safe	Safe	Safe	Safe	Safe
DTD Retrieval	Safe	Safe	Safe	Safe	Safe
Decompression Bomb	Safe	Safe	Safe	Safe	Vulnerable

.Net:

The following table lists all supported .NET XML parsers and their default safety levels:

XML Parser	Safe by default?
LINQ to XML	Yes
XmlDictionaryReader	Yes
XmlDocument	
...prior to 4.5.2	No
...in versions 4.5.2+	Yes
XmlNodeReader	Yes
XmlReader	Yes
XmlTextReader	
...prior to 4.5.2	No
...in versions 4.5.2+	Yes
XPathNavigator	
...prior to 4.5.2	No
...in versions 4.5.2+	Yes

Conclusioni

Questa tipologia di attacchi è molto semplice da prevenire ma potenzialmente pericolosa per un applicativo non configurato opportunamente per prevenirli.

Qualora un'applicazione e/o servizio dovesse gestire o immagazzinare XML, a prescindere dal linguaggio, è necessario durante la scelta di un parser XML, controllare che non sia soggetto per default ad attacchi XXE o XML BOMBS e se necessario manualmente disattivare le entità esterne.

Per le injection di tag o di input mal formato l'applicativo deve sanificare l'input dell'utente prima di processarlo o salvarlo.

Bibliografia

1. [https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/07-Testing for XML Injection](https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/07-Testing_for_XML_Injection)
2. [https://cheatsheetseries.owasp.org/cheatsheets/XML External Entity Prevention Cheat Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html)
3. <https://asfiyashaikh.medium.com/xml-injection-5e6336b95274>
4. <https://www.whitehatsec.com/glossary/content/xml-injection>
5. <https://www.immuniweb.com/vulnerability/xml-injection.html>
6. <https://depthsecurity.com/blog/exploitation-xml-external-entity-xxe-injection>