**SAD Lab**

**EXPERIMENT NO. 8**

**Aim**: Cross-Site Scripting (XSS) vulnerability and OS Command vulnerability.

**Theory**:

1. What is Cross-Site Scripting?

**Cross-site Scripting** (XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code.

A web page or web application is vulnerable to XSS if it uses unsanitized user input in the output that it generates. This user input must then be parsed by the victim’s browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are most common in JavaScript, primarily because JavaScript is fundamental to most browsing experiences.

Cross-site Scripting may also be used to deface a website instead of targeting the user. The attacker can use injected scripts to change the content of the website or even redirect the browser to another web page, for example, one that contains malicious code.

1. What can XSS be used for?

An attacker who exploits a cross-site scripting vulnerability is typically able to:

* Impersonate or masquerade as the victim user.
* Carry out any action that the user is able to perform.
* Read any data that the user is able to access.
* Capture the user's login credentials.
* Perform virtual defacement of the web site.
* Inject trojan functionality into the web site.

1. What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

1. **Reflected XSS**: Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.
2. **Stored XSS**: Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.
3. **DOM-based XSS**: DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.
4. How to Prevent XSS Attacks?

To keep yourself safe from XSS, you must sanitize your input. Your application code should never output data received as input directly to the browser without checking it for malicious code. Effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:

1. **Filter input on arrival**: At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
2. **Encode data on output**: At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
3. **Use appropriate response headers**: To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.
4. **Content Security Policy**: As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

**Reflected XSS**

Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

<https://insecure-website.com/search?term=gift>

The application echoes the supplied search term in the response to this URL:

| <p>You searched for: gift</p> |
| --- |

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

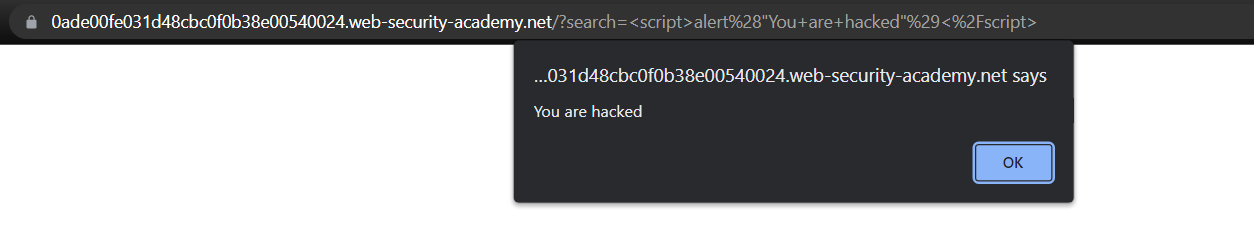
<https://insecure-website.com/search?term=><script>/\*+Bad+stuff+here...+\*/</script>

This URL results in the following response:

| <p>You searched for: <script>*/\* Bad stuff here... \*/*</script></p> |
| --- |

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.





**Stored XSS**

Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following:

*POST /post/comment HTTP/1.1*

*Host: vulnerable-website.com*

*Content-Length: 100*

*postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net*

After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

| <p>This post was extremely helpful.</p> |
| --- |

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

| <script>*/\* Bad stuff here... \*/*</script> |
| --- |

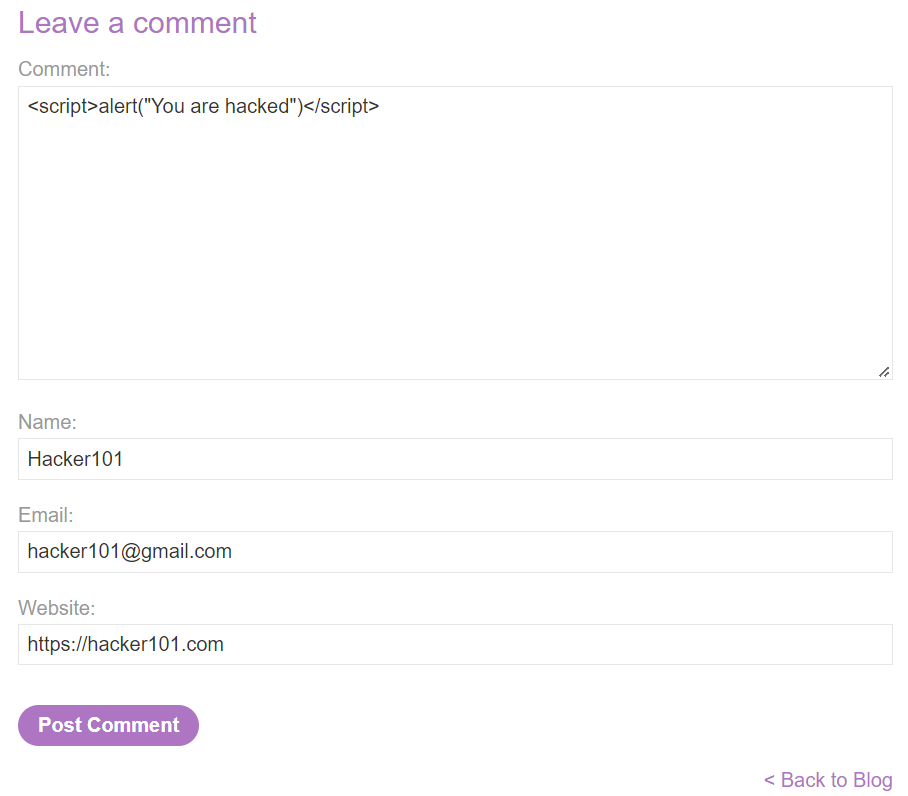
Within the attacker's request, this comment would be URL-encoded as:

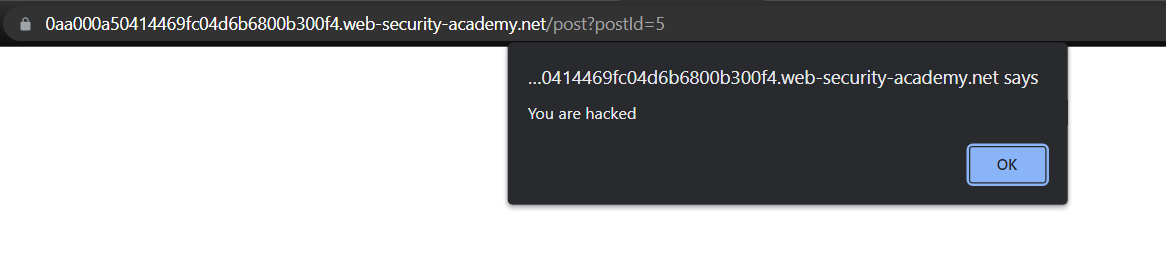
*comment=%3Cscript%3E%2F\*%2BBad%2Bstuff%2Bhere...%2B\*%2F%3C%2Fscript%3E*

Any user who visits the blog post will now receive the following within the application's response:

| <p><script>*/\* Bad stuff here... \*/*</script></p> |
| --- |

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.





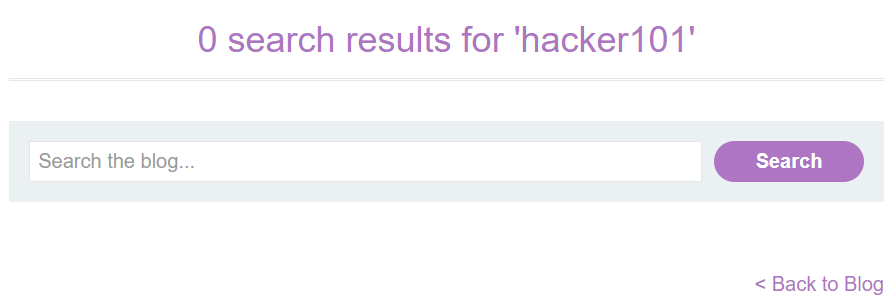


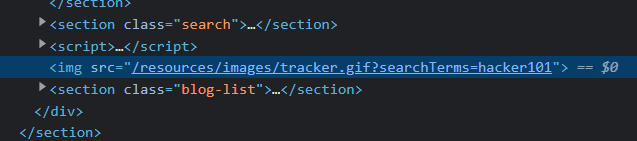
**DOM-based XSS**

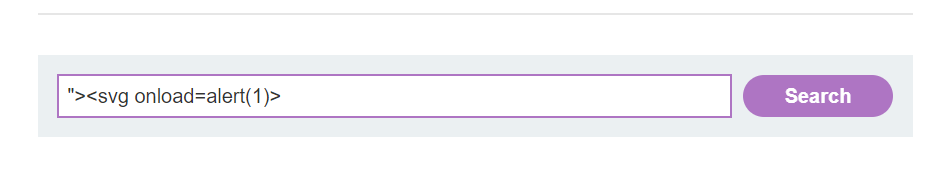
The document.write sink works with script elements, so you can use a simple payload, such as the one below:

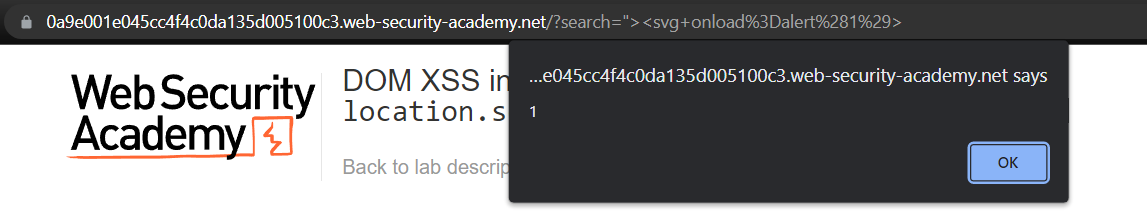
| document.write('... <script>alert(document.domain)</script> ...'); |
| --- |

Note, however, that in some situations the content that is written to document.write includes some surrounding context that you need to take account of in your exploit. For example, you might need to close some existing elements before using your JavaScript payload.









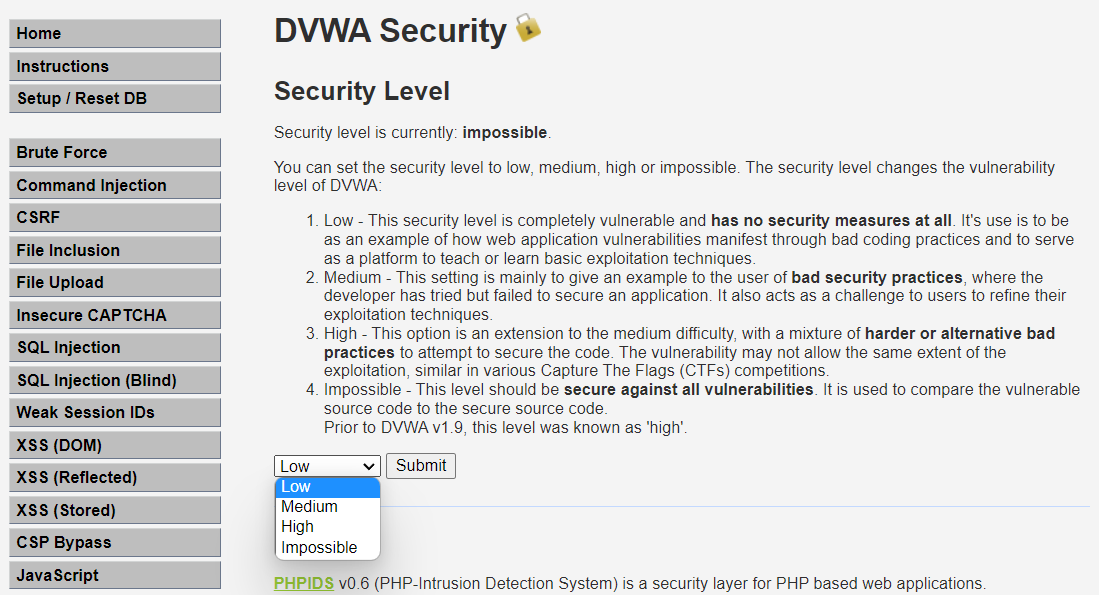
**SQL Injection Exploitation using DVWA**

Here, we will use the Damn Vulnerable Web Application (DVWA). It’s a web app developed in PHP and MySQL and intentionally made to be vulnerable.

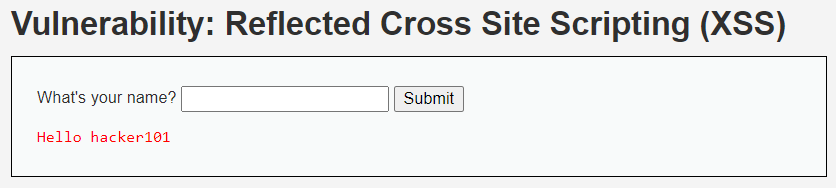
Use the default credentials below:

* *Username:* ***admin***
* *Password:* ***password***

After a successful login, you will see the DVWA main page. First, click on the DVWA Security on the bottom left, set security to Low, and click Submit.

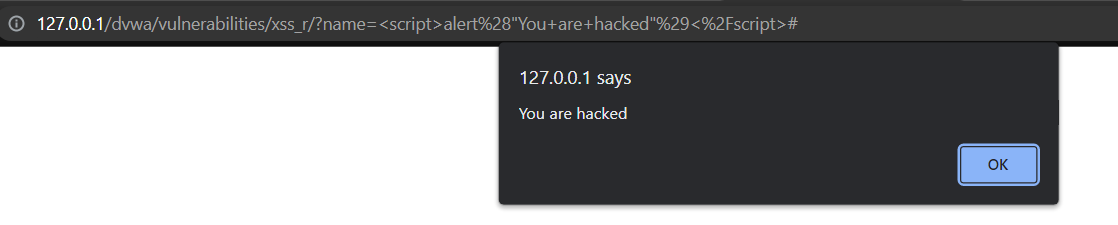


Now, click on XSS (Reflected) on the left pane to select the vulnerability to Reflected XSS because we are going to practice Reflected XSS attack. Input a unique string in the input section and submit it as shown below.

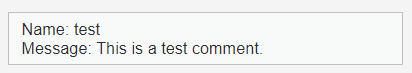


Now, press CTRL+U to view the page source and find the unique string which we have entered. Our unique string is present in the page source and is vulnerable to XSS attack. Now enter the payload <script>alert()</script> in the same field and submit the request.

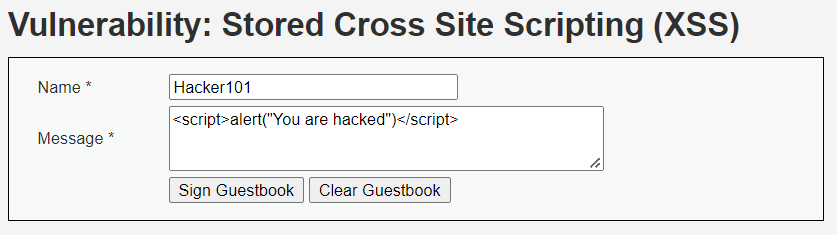


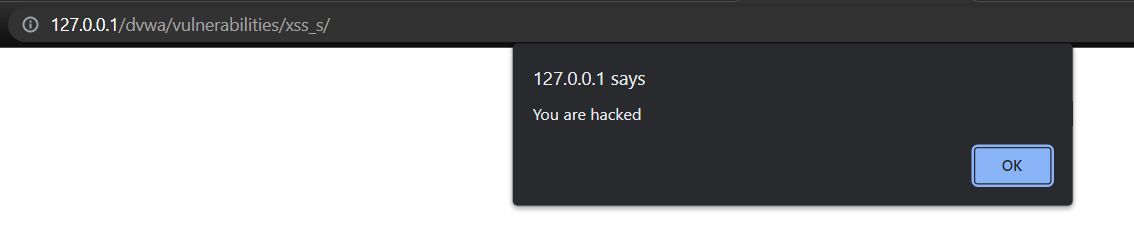


Now, click on XSS (Stored) on the left pane to select the vulnerability to Stored XSS because we are going to practice Stored XSS attack. Input a unique string in the input section and submit it as shown below.

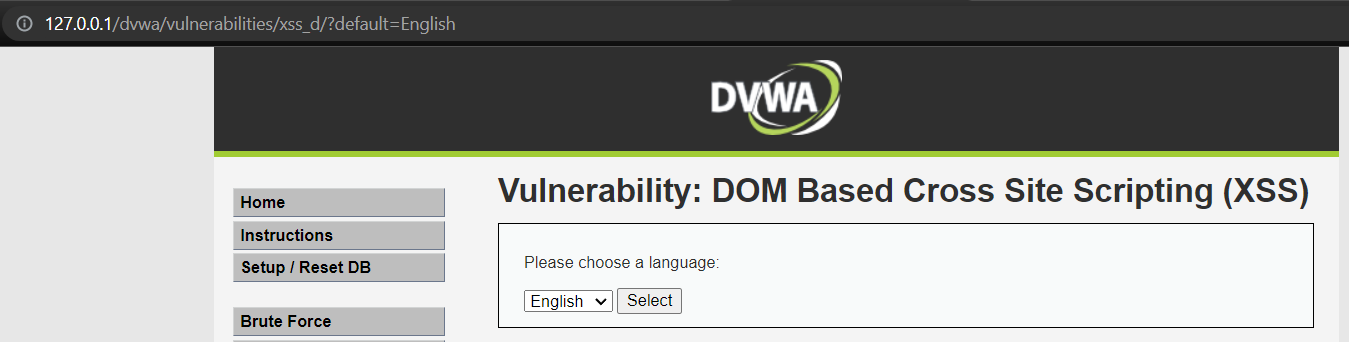


On pressing CTRL+U to check page source and then searching for test string revealed that test is reflecting. Therefore the field may be vulnerable to Stored XSS attack. Now enter the payload <script>alert()</script> in the same field and submit the request.



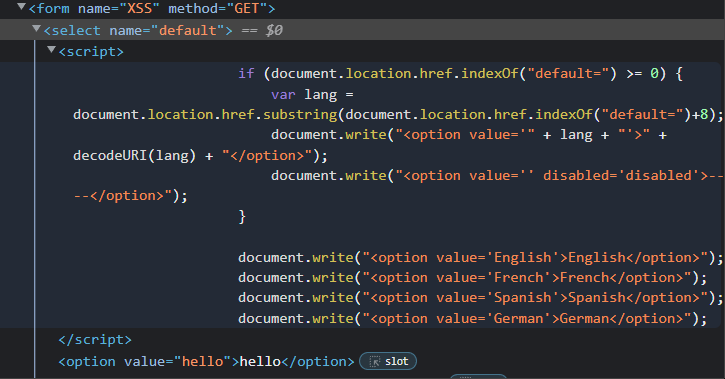


Further, click on XSS (DOM) on the left pane to select the vulnerability to DOM XSS because we are going to practice DOM XSS attack. We are on a challenge page. Click on the Select button to check how the application is behaving. On button click it sets the value of default parameter to English in the URL.

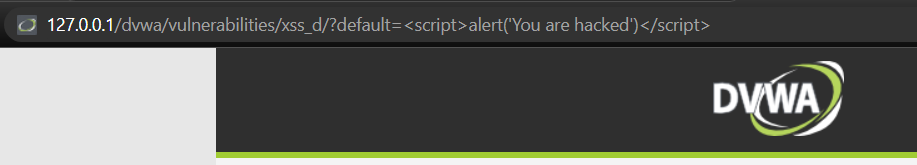


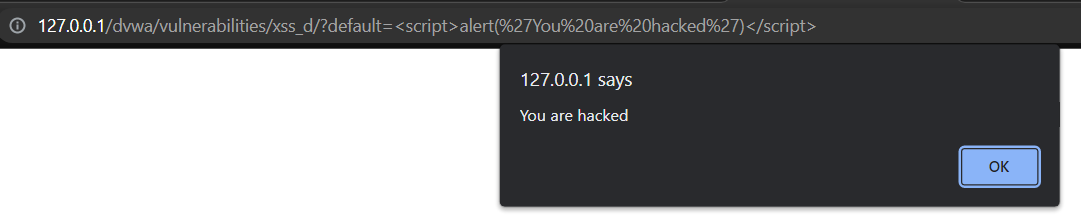
So let us modify the value of the default parameter in the URL with some unique string hello and check where our input is reflected in HTML DOM and how it is processed.

In the below screenshot document.location.href.substring() function is our potential source because it is accepting the user input inside the default parameter. document.write() as our potential sink because it reflects the entered value from source inside the <option>...</option> tag. Which means whatever we input in the default parameter will reflect back inside <option>...</option> tag inside HTML DOM.



Since our unique string is reflected back in HTML DOM, let us inject our basic XSS payload <script>alert('DOM XSS')</script> in place of hello in default parameter. We can clearly see in the screenshot that our injected payload got executed successfully and we got XSS pop up.





**Conclusion**:

Thus we have studied how to perform different types of Cross-Site Scripting (XSS) vulnerability and OS Command vulnerability.