**Experiment 8**

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

**Theory**:

### **Cross-Site Scripting (XSS) Vulnerability**

Cross-Site Scripting (XSS) is a prevalent web security vulnerability that enables attackers to inject malicious scripts into content that is served to users. Since web applications often allow users to input data that gets displayed on webpages, XSS vulnerabilities can arise when this data is not properly validated or sanitized before being rendered.

The essence of XSS lies in its exploitation of the trust a user has in a specific website. For instance, when a user visits a website that includes untrusted data—such as comments, user profiles, or other forms of dynamic content—an attacker can inject malicious scripts into these inputs. When other users load the affected page, the malicious scripts execute in their browsers as if they originated from the trusted website.

XSS attacks can be categorized into three main types: stored XSS, reflected XSS, and DOM-based XSS. In stored XSS, the malicious payload is permanently stored on the server (for example, in a database) and served to users whenever they access that particular page. This makes it particularly dangerous, as the payload can affect multiple users over time. Reflected XSS, on the other hand, involves the immediate reflection of the injected script back to the user. This typically happens through a crafted URL that includes the malicious payload. When the user clicks the link, the server processes the request and sends back the malicious code without validation. Finally, DOM-based XSS occurs purely on the client side, where the vulnerability exists in the web application's client-side JavaScript, manipulating the Document Object Model (DOM) in the user's browser.

The impact of XSS can be severe. Attackers can steal sensitive data, including cookies or session tokens, allowing them to impersonate users and gain unauthorized access to their accounts. Furthermore, they can perform actions on behalf of users, leading to account compromise, defacement of web content, and redirection to malicious websites that can phish for credentials or spread malware.

To mitigate XSS vulnerabilities, developers must prioritize secure coding practices. Input validation and output encoding are crucial. Input validation ensures that only expected and safe data formats are accepted. This can be achieved by implementing strict checks on the data type, length, format, and range. Output encoding involves converting potentially dangerous characters in user input into a safe format that will not be executed as code when rendered in a web browser. Additionally, employing Content Security Policies (CSP) helps restrict the execution of scripts from untrusted sources, providing an extra layer of security.

### **OS Command Injection Vulnerability**

Operating System Command Injection is another critical security vulnerability that occurs when an application takes user input and uses it to construct operating system commands without proper validation or sanitization. This can allow an attacker to execute arbitrary commands on the host system, leading to severe consequences.

Applications that incorporate features requiring system-level commands, such as file manipulation or process execution, are particularly susceptible to OS Command Injection. When user input is directly embedded in these commands, an attacker can manipulate the input to execute commands that were not intended by the application developer. For instance, if a web application takes a filename from user input and constructs a command to open that file, an attacker might input a filename that includes command-line separators (like ; on Unix-based systems) followed by malicious commands.

The impact of OS Command Injection can be devastating. Attackers can gain unauthorized access to sensitive files, modify or delete critical data, and even take control of the entire system if the application is running with elevated privileges. This vulnerability can also lead to privilege escalation, where attackers leverage their ability to execute system commands to gain higher levels of access than intended.

To safeguard against OS Command Injection, developers should follow secure coding practices. Input validation is paramount; all user inputs must be rigorously checked to ensure they conform to expected formats and do not include potentially harmful characters. It's advisable to use predefined commands or parameterized commands rather than constructing them dynamically. Using high-level APIs that abstract away direct command execution can also significantly reduce risk.

Furthermore, adhering to the principle of least privilege is crucial. Applications should run with the minimum privileges necessary to perform their functions, limiting the potential damage if an attacker successfully exploits a vulnerability. Regular security audits and penetration testing can help identify and mitigate these vulnerabilities before they can be exploited by malicious actors.

### 

Both Cross-Site Scripting (XSS) and OS Command Injection vulnerabilities represent significant threats to web applications. Understanding their mechanisms and potential impacts is essential for developers and security professionals. By implementing robust security measures, such as rigorous input validation, output encoding, and adopting best practices in application design, organizations can better protect their applications and users from these pervasive vulnerabilities.

**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> |
| --- |

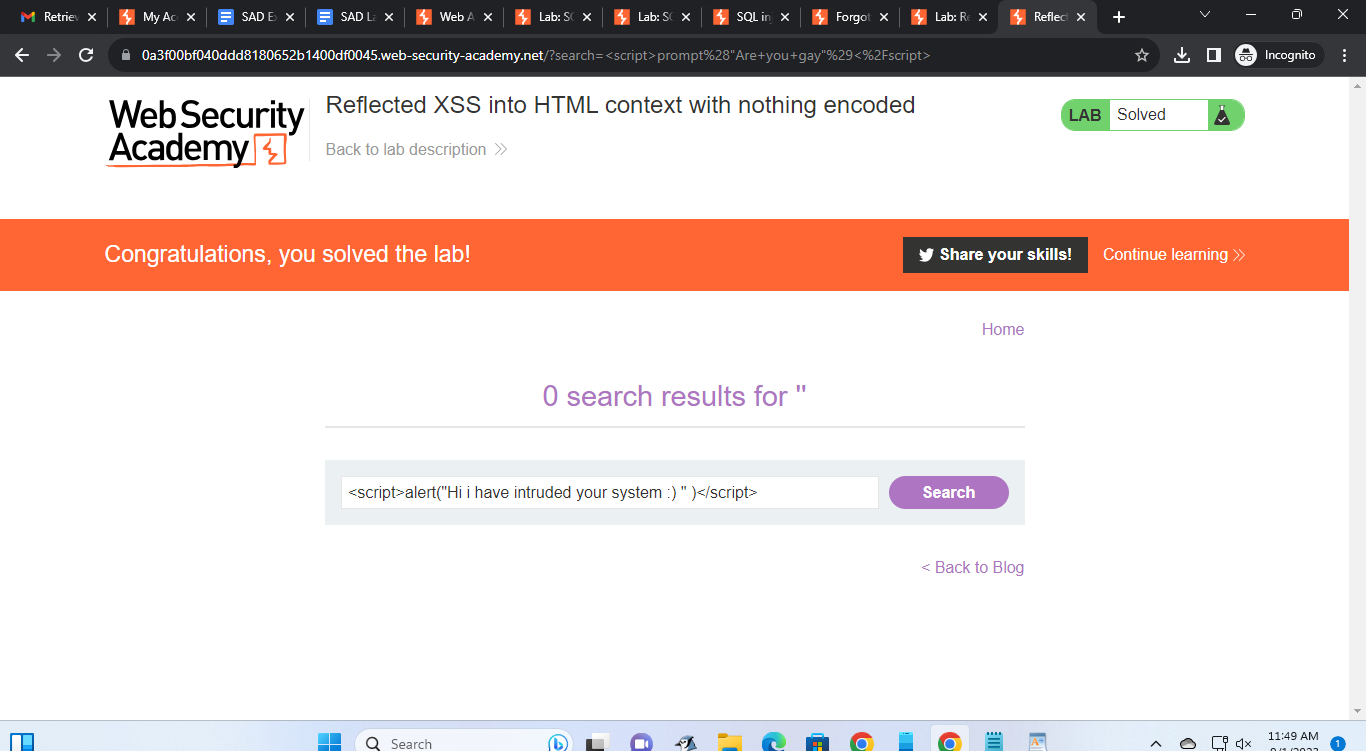
Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

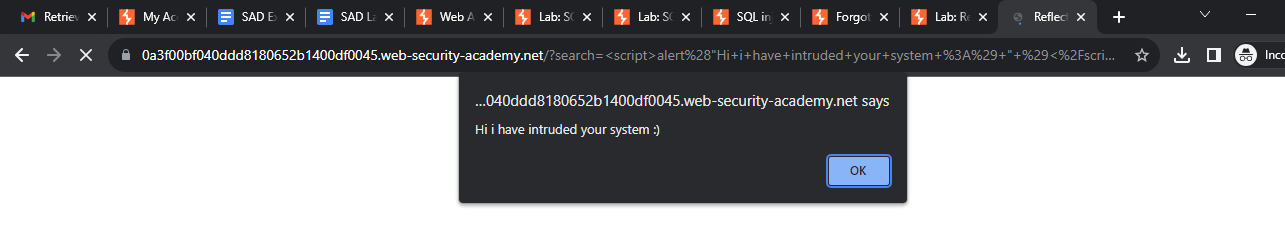
<https://insecure-website.com/search?term=><script>/\*+Bad+stuff+here...+\*/</script>

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> |
| --- |

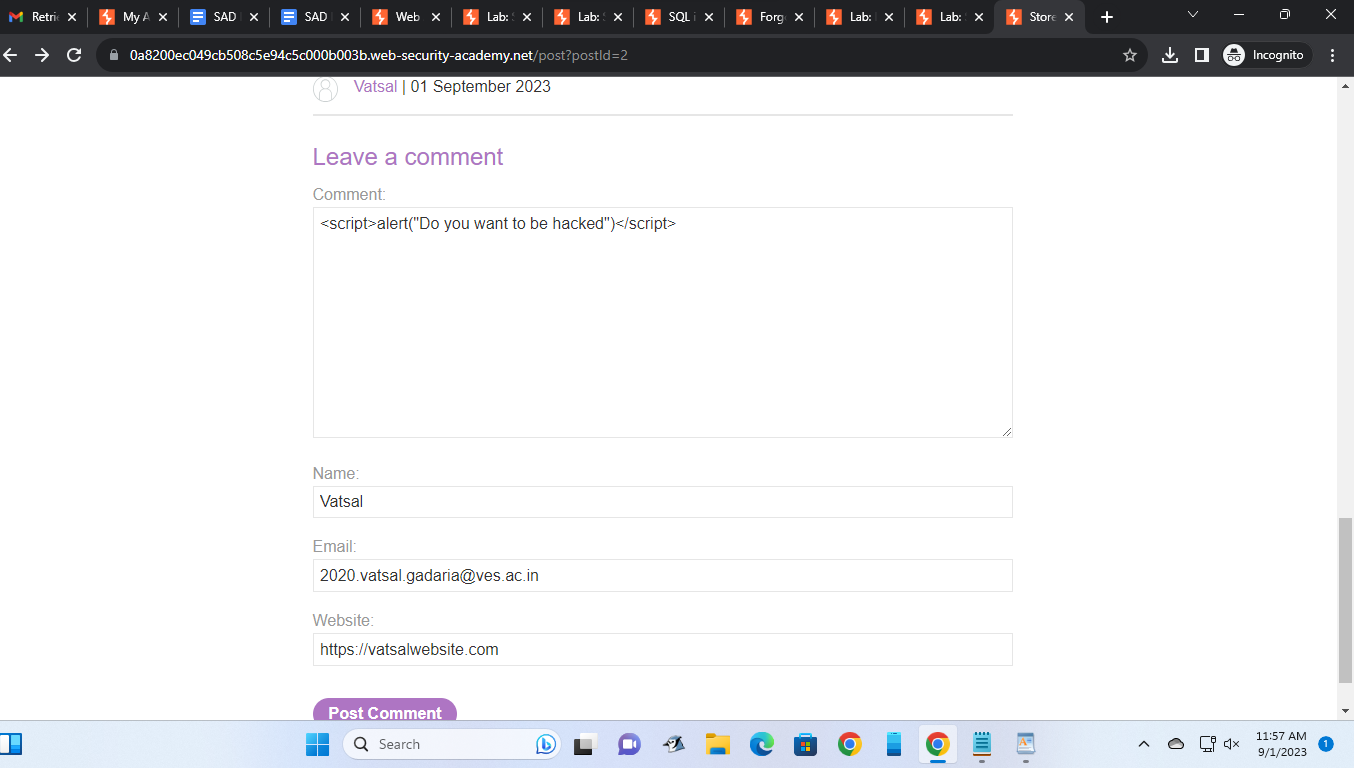
Within the attacker's request, this comment would be URL-encoded as:

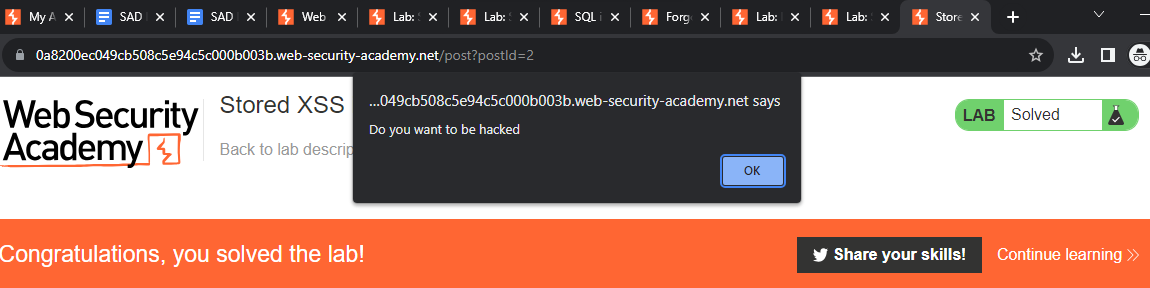
*comment=%3Cscript%3E%2F\*%2BBad%2Bstuff%2Bhere...%2B\*%2F%3C%2Fscript%3E*

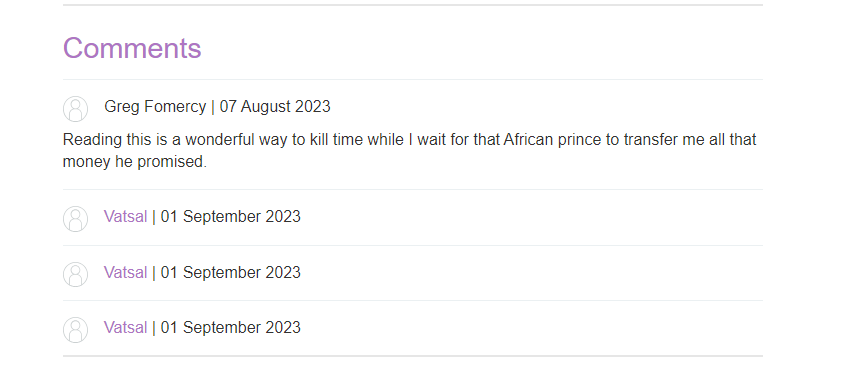
Any user who visits the blog post will now receive the following within the application's response:

| <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.





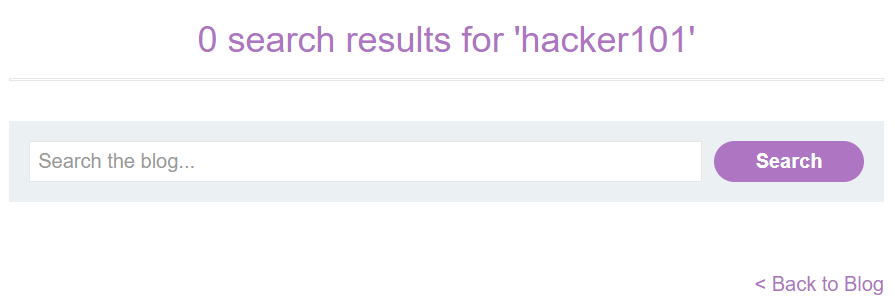


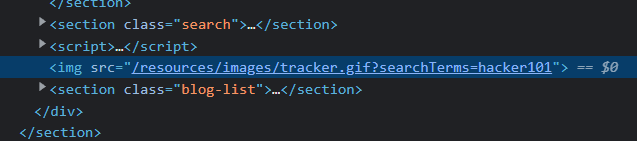
**DOM-based XSS**

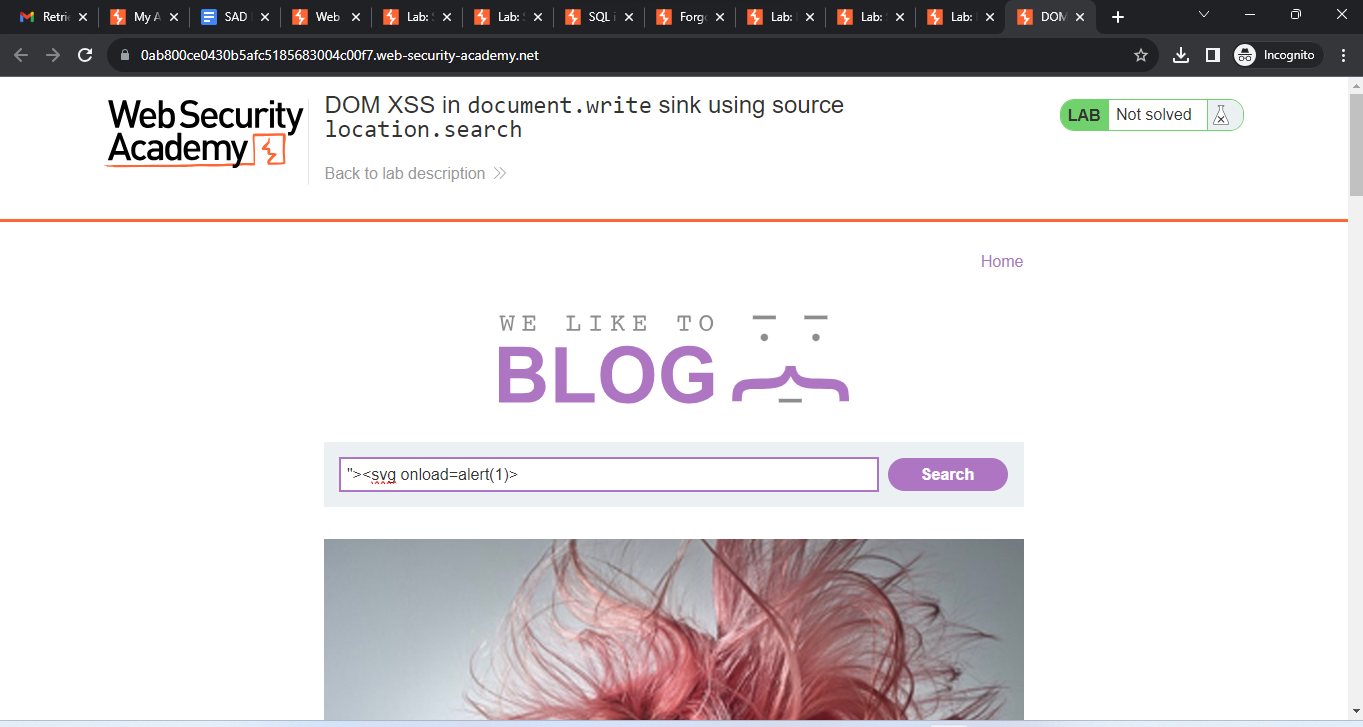
The document.write sink works with script elements, so you can use a simple payload, such as the one below:

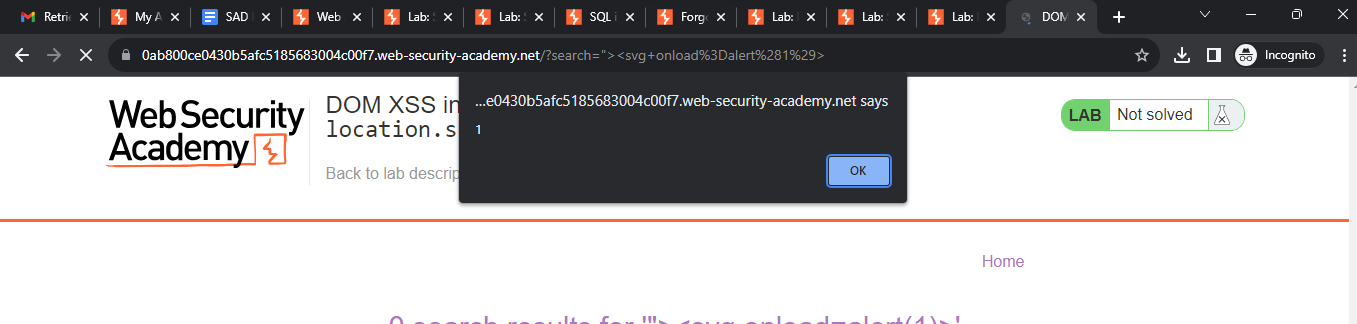
| document.write('... <script>alert(document.domain)</script> ...'); |
| --- |

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.









**Conclusion**:

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