Truth OF
Cross Site Scripting (xss)

- Abishekraghav Murugeashan

Introduction

✓ Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

✓ It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other.

✓ Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data.

✓ If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

Process of XSS
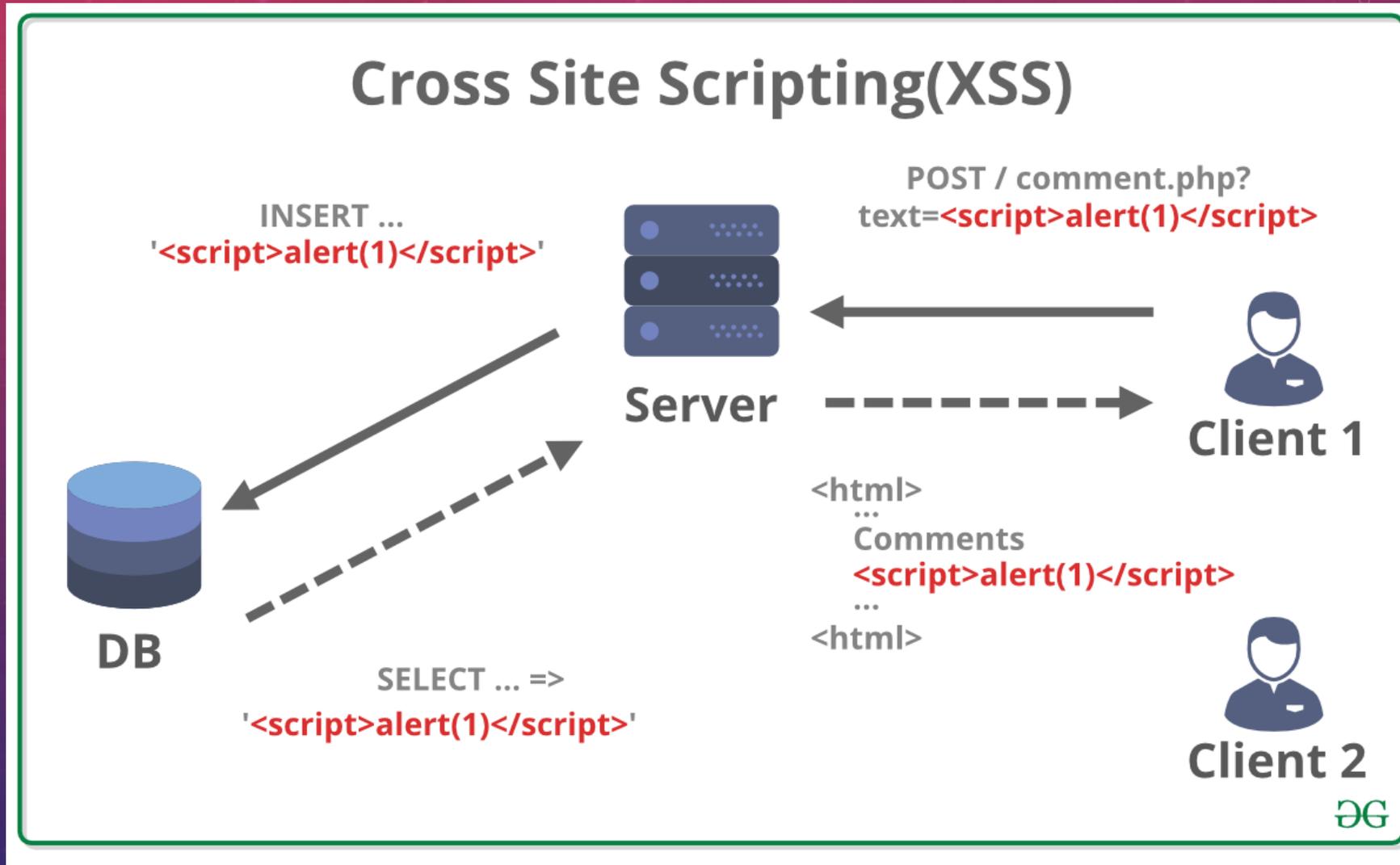
XSS work
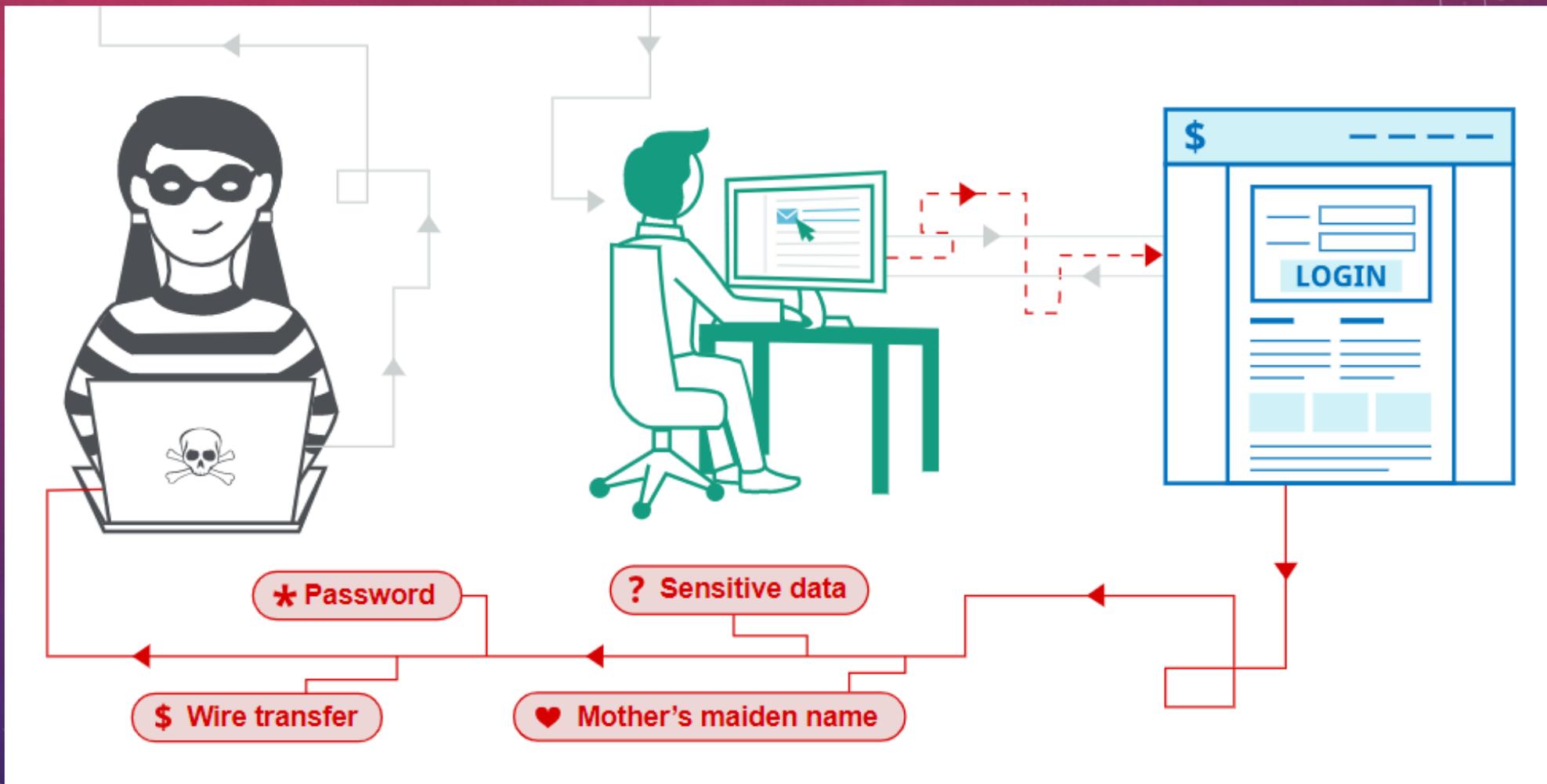
XSS ATTACK VECTORS

- **JS within HTML using <script> tag**

- **JS from external source using src tag**

- **JS can be embed into HTML with event handlers**

- **Ex : onload, onmouseover**

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
    alert("Website has been hacked");
}
</script>
</head>

<body onmouseover="myFunction()">
<h1>Hello World!</h1>
</body>

</html>
```

Website has been hacked

OK

Types of XSS

- ✓ Reflected XSS

- ✓ Stored XSS

- ✓ DOM based XSS

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.

Here is a simple example of a reflected XSS vulnerability:
https://attacker-website.com/status?message=All+is+well. <p>Status: All is well.</p>

✓ The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:
https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script> <p>Status: <script>/* Bad stuff here... */</script></p>

✓ If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.

Test for reflected XSS

• Test every entry point. Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path.

• It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.

• Submit random alphanumeric values. For each entry point, submit a unique random value and determine whether the value is reflected in the response.

• Determine the reflection context. For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.

• Test alternative payloads. If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS

Stored XSS

- Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

- 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+Mon
toya&email=carlos%40normal-user.net
```

Test for stored XSS vulnerabilities

•Parameters or other data within the URL query string and message body.

•The URL file path.

•HTTP request headers that might not be exploitable in relation to <u>reflected XSS</u>.

•Any out-of-band routes via which an attacker can deliver data into the application.

• The routes that exist depend entirely on the functionality implemented by the application: a webmail application will process data received in emails; an application displaying a Twitter feed might process data contained in third-party tweets; and a news aggregator will include data originating on other web sites.

 The exit points for stored XSS attacks are all possible HTTP responses that are returned to any kind of application user in any situation.

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.

In the following example, an application uses some JavaScript to read the value from an input field and write that value to an element within the HTML:

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

You searched for: <img src=1 onerror='/* Bad stuff here... */'>

In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

Additionally, the website's scripts might perform validation or other processing of data that must be accommodated when attempting to exploit a vulnerability.

There are a variety of sinks that are relevant to DOM-based vulnerabilities.

Please refer to the list below for details. 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> ...');

The following are some of the main sinks that can lead to DOM-XSS vulnerabilities:

- ✓ document.domain some

- ✓ DOMElement.innerHTML

- ✓ someDOMElement.outerHTML

- ✓ someDOMElement.insertAdjacentHTML

- ✓ someDOMElement.onevent

- ✓ document.writeln()

XSS MITIGATIONS & BYPASSES

- **Base64 Encoding**
  - javascript:eval(atob('YWxlcnQoZG9jdW1lbnQuY29
    va2llKTs='));
  - Javascript:alert(document.cookie);
- **Avoiding quotes**
  - javascript:eval(String.fromCharCode(97,108,
    101,114,116,40,100
    ,111,99,117,109,101,110,116,46,99,111,11
    1,107,105,101,41,59))

```
<script>
    var name = document.location.hash.substr(1);
    document.write("<h1>Hello, " + name.replace(/<\/?[^>]+>/gi, "") + "</h1>");
</script>
```

```
> String.fromCharCode(97,108,101,114,116,40,100,111,99,117,109,101,110,116,46,99,111,111,107,105,101,41,59)
< "alert(document.cookie);"
```