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

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>
```

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

Exercise 2.1

1. Identify places where user input is *directly* included in the output
2. Perform a successful *Reflected XSS* attack (★)
3. Perform a successful *DOM 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!** 💯
- **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>:
<!-- ... -->
```

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.*

CSS

```
<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

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

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

Exercise 2.2

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

ℹ️ *If your attack seems to be blocked or otherwise prevented, you can either try to beat the security mechanism or just find an easier target!*