<WA/>
2025

Browser security

(Accidentally) executing code in the browser

Enrico Masala
Antonio Servetti

Goal

- Understanding security implications in the browser
  - Interpreting HTML code
  - Interpreting JS code
- Common attack vectors
  - Cross-site scripting (“XSS”)
  - Mitigation techniques

Browser environment

- Basic browsing operation
  - The browser loads a page starting from HTML content
  - The browser processes HTML content, loads additional resources (images, CSS, JS code, …), and processes/executes them
  - Repeats until all additional resources are processed/executed
- Code can read/write/modify elements in the page and sensitive information in the browser environment
  - The browser is executing somebody else's code!
  - Thus, the code must be trusted to avoid malicious behaviors such as stealing/sending sensitive information to another site: Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS)

- Cross-Site Scripting (XSS) attacks are a type of injections where **malicious scripts** (HTML / JS code) are inserted in a trusted website
- The user and the browser has no way to know that the script should not be trusted, since it comes from a trusted source
- Code is executed in the browser, and it can:
  - **access sensitive information** in the browser such as cookies, tokens, secrets, etc. (depending on where they are stored), and send them to the attacker
  - even **modify/rewrite page content** (DOM manipulation) changing what is shown and the behavior of the web page

https://owasp.org/www-community/attacks/xss/

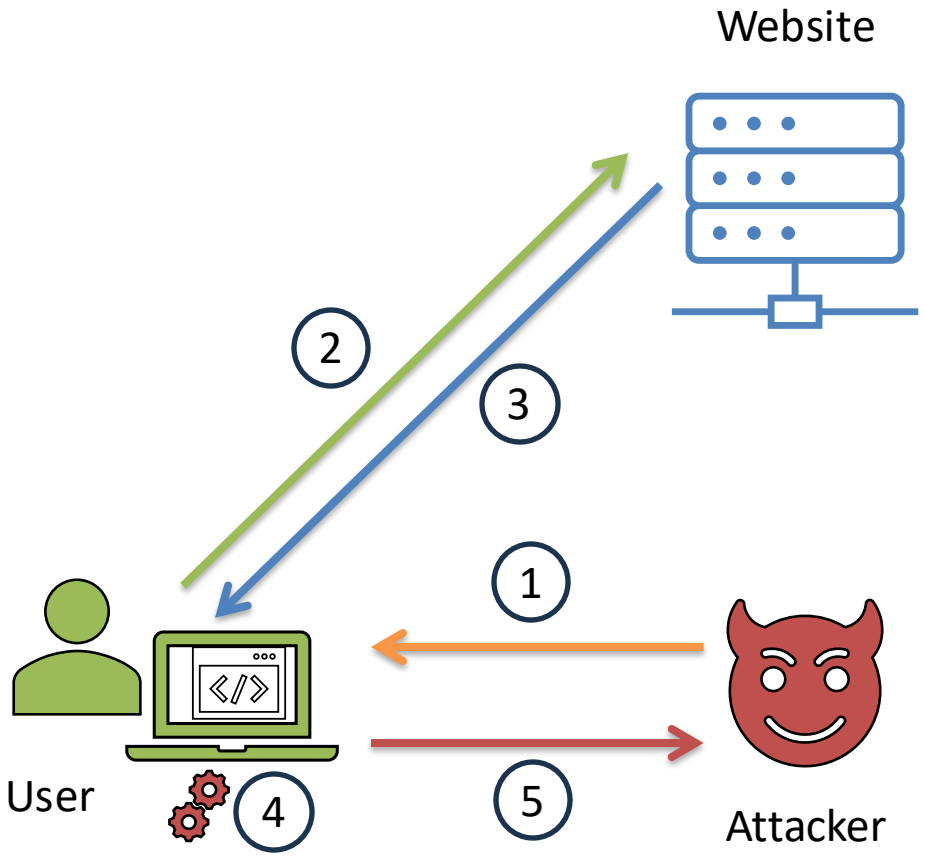
Classification

- How they work
  - Reflected XSS attack
  - Stored XSS attack
  - DOM-based XSS attack
- Where they happen
  - Server XSS
  - Client XSS

https://owasp.org/www-community/attacks/xss/

Reflected XSS Attack

https://owasp.org/www-community/attacks/xss/



1. Malicious string is sent to the victim (link in email or in another website)
2. The user opens the link which sends a request to the server
3. The server reflects it back in the response, e.g., in a search result, error message, ...
4. The browser executes the malicious code since it comes from a trusted web server
5. The code steals sensitive info

Note: This XSS is “not persistent”, i.e. a new request must be made to attack again

Reflected XSS Attack: Example

1. Click on a **crafted URL**

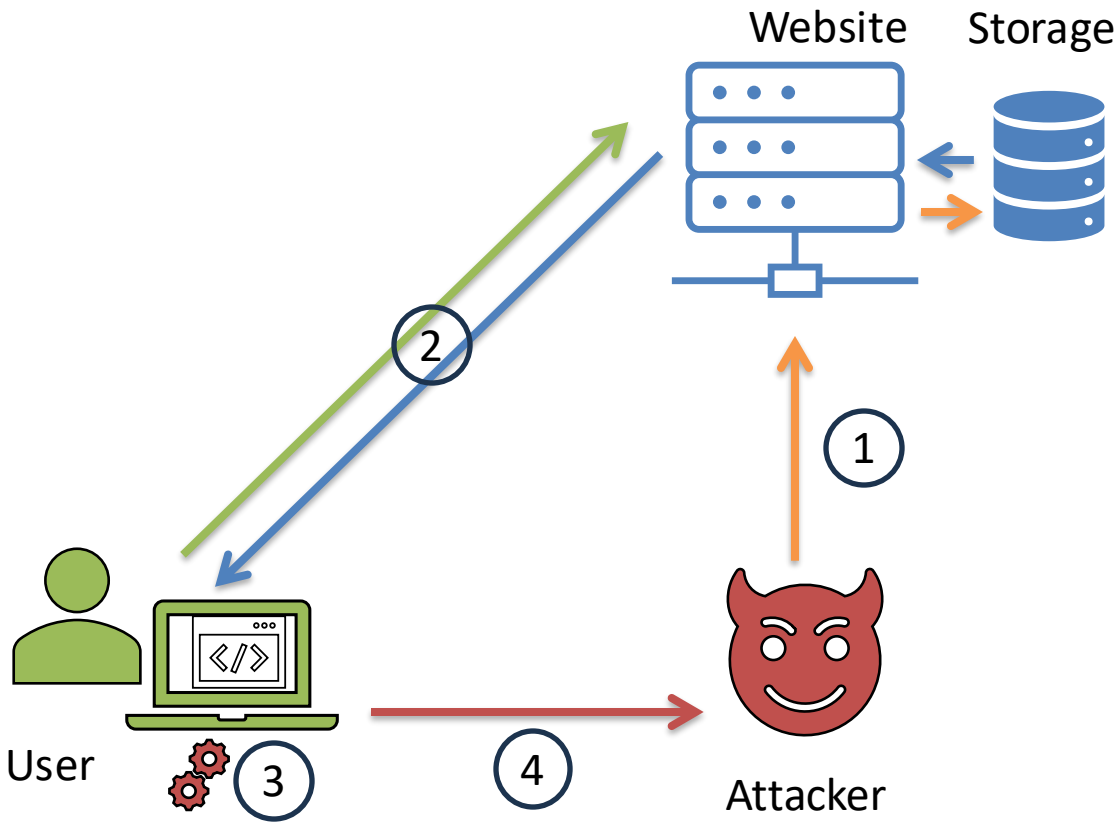
```
http://myapp.com/search?page=<script>alert('xss')</script>
```

2. Server response reflects back the content of an URL parameter

```
<html>
<body>
Page <script>alert('xss')</script> was not found!
</body>
</html>
```

Stored XSS Attack

https://owasp.org/www-community/attacks/xss/



1. The attacker injects the vulnerable website with a malicious script which is stored by the website
2. The user visits the website and unknowingly retrieves the malicious code
3. The browser executes the malicious code since it comes from the same trusted origin (“web server”)
4. The code steals sensitive info

Note: This XSS is “persistent” since the attack runs each time the server is accessed

Stored XSS Attack: Example

1. Attacker save this content in an article in a forum

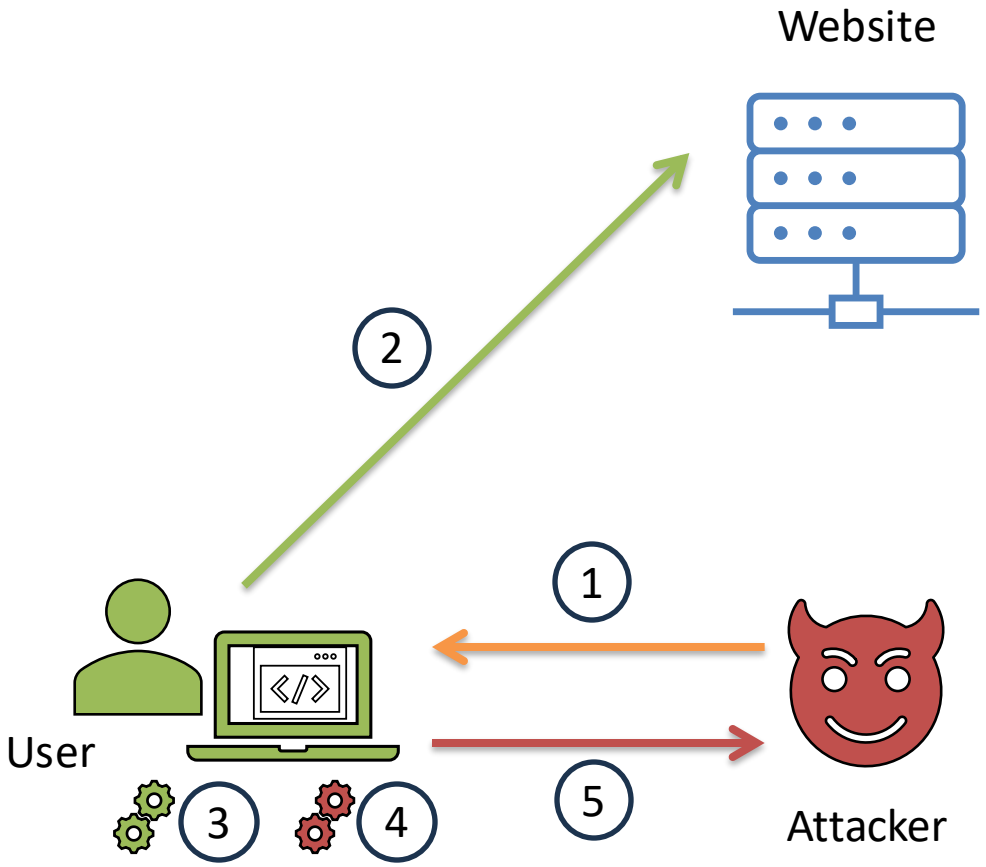
```
<h1>Cybersecurity</h1>
<p>Attend good courses to understand it!</p>
<script>alert('xss')</script>
```

2. Server sends such HTML code to any user that requests the article

```
<html><body>
<h1>Cybersecurity</h1>
<p>Attend good courses to understand it!</p>
<script>alert('xss')</script>
</body></html>
```

DOM-based XSS Attack

https://owasp.org/www-community/attacks/xss/



1. Malicious string is sent to the victim (typically a crafted link)
2. The user opens the link which becomes active in the browser
3. The JS code in the browser executes an unsafe action that modifies the browser DOM (e.g., using part of the link to create new local HTML content)
4. The DOM triggers the execution of the malicious code
5. The code steals sensitive info

This XSS works because the original JS code of the website has a security flaw

DOM-based XSS Attack: Example

- The malicious code is taken by Javascript from a source that can be controlled by an attacker, e.g., URL: window.location
- **AND** the data is passed to an element / function ("**sink**") that supports dynamic code execution
  - innerHTML, eval(), document.write, … that can alter the DOM in the page
  - list of common "sinks" **to avoid**
- The browser executes code that yield unexpected behavior, since the sink was expecting code but not that one

```
// Malicious URL fragments
#<script>alert('xss')</script>
#<img src=x onerror=alert('xss')>

// Example of crafted URL
http://myapp.com/list#<script>alert('xss')</script>
```

```
// DOM-based XSS vulnerable code
const userText = window.location.hash.substring(1);
document.write(userText);
```

Example from: https://qwiet.ai/dom-based-xss-attacks-how-to-identify-and-fix-vulnerabilities/

https://portswigger.net/web-security/cross-site-scripting/dom-based

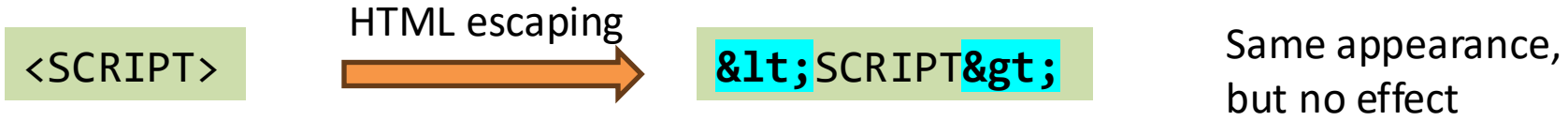
Looking at XSS from a different perspective

- Various types of XSS can overlap (stored and reflected)
- XSS can also be explained in terms of where it happens: **server** or **client**
- **Server XSS**
  - Untrusted data is coming from the server, either pre-stored or reflected
  - The browser is simply executing scripts that it deems valid
- **Client XSS**
  - Untrusted data is used to update the browser DOM in an unsafe JS call, i.e., that can introduce JS in the DOM
  - Source of untrusted data can be the DOM itself (DOM-based XSS) or the server

https://owasp.org/www-community/Types_of_Cross-Site_Scripting

XSS Attack Mitigation: Client

- ONLY use SAFE Javascript methods/functions, see guidelines
  - Untrusted data to be treated **only as TEXT** to be displayed (never as code, of any kind). Convert sensitive characters (<,>,&, ' , ",/) to **HTML entities** using **escaping**



  - Use only DOM methods to create dynamic DOM elements
    - Examples: `.addElement('div')` `.appendChild(…)` etc.
  - No `innerHTML`, `document.write()`, …, avoid methods that internally run `eval()`
  - …

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Escaping

- Use validator.js **`escape()`** function

```
const validator = require('validator');

const clean = validator.escape('<b>hello there</b>');
// clean: &lt;b&gt;hello there&lt;&#x2F;b&gt;
```

- Available also as a method on `check()` by `express-validator`

```
app.post('/answer', [
  check('explanation').escape() //  explanation can safely display any HTML code
], (req, res) => {
. . . Process request: body.explanation will be already escaped (ready to be stored in db)
});
```

Sanitization

- Escaping is not always possible: for instance, a visual HTML editor that must immediately show the result, or some HTML formatting is allowed
- If input is HTML that needs to be rendered, it must be **sanitized** before allowing the browser to process it
- Sanitize with good libraries such as DOMPurify which removes potentially dangerous content when it is processed by the browser

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

DOMPurify

- See instructions at https://www.npmjs.com/package/dompurify
- Typically use it on the server side, before storing and/or returning data

```
const createDOMPurify = require('dompurify');  // npm install dompurify jsdom
const { JSDOM } = require('jsdom');

const window = new JSDOM('').window;
const DOMPurify = createDOMPurify(window);

const clean1 = DOMPurify.sanitize('<b>hello there</b>');
// clean1: <b>hello there</b>

const clean2 = DOMPurify.sanitize(`<img src="/" onerror="alert('hacked xss');">`);
// clean2: <img src="/">
```

Escaping vs Sanitizing

Also available with express-validator.js

- Escaping makes content harmless when interpreted, does not modify how content is shown (use `.escape()` from validator.js)

```
<p>How to include scripts: <SCRIPT src='file.js' /></p>
```

```
&lt;p&gt;How to include scripts: &lt;SCRIPT
src=&#x27;file.js&#x27; &#x2F;&gt;&lt;&#x2F;p&gt;
```

Not interpreted, just shown as

```
<p>How to include scripts:
<SCRIPT src='file.js' /></p>
```

- Sanitization REMOVES unsafe content (DOMPurify)

```
<p>How to include scripts: <SCRIPT src='file.js' /></p>
```

```
<p>How to include scripts: </p>
```

Interpreted and shown as

```
How to include scripts:
```

XSS Attack Mitigation: Server

- Context-sensitive server-side output encoding
  - Modify server output so that it cannot harm in the place where it will be used
- Some examples
  - HTML context: every content should be **escaped or DOM-purified**
    - `<script>` → `&lt;script&gt;` or `<script>` → `''`
  - HTML attribute context: set them as strings. Attributes that can be used with code are not safe even if used with strings (e.g., `onClick` etc.)
    - `<div attr="$unsafeVar">`
  - CSS context: only set property values. The rest is unsafe
    - `<style> selector { property: "$unsafeVar"; }</style>`

Most frameworks (**including React**) help to mitigate this problem when using variables coming from their environment

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

XSS Attack Mitigation: Server

- Other examples
  - JS context: everything should be quoted (with " "). Try to avoid JS context since this is not always enough
  - URL context: use the URL encoding format %HH (hex value) to encode special characters of the URL
    - `site.com/search?<script>` → `site.com/search?%3Cscript%3E`

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

Other Advices

- **Input validation** for API: **reject anything suspicious** (characters, code etc.) as soon as possible
- Example #1: the name of a person should not contain HTML tags
  - beware of “ ’ “, spaces and other Unicode characters if appropriate
- Example #2: text field with an HTML snippet, as in a programming forum. Ok but:
  - Either escape it before storing (best)
  - Or escape it before returning it to client

```
//express-validator example

app.post('/api/articles',
  […  check(text).escape(), … ],
async (req, res) => { …
// body.text is already HTML-escaped here
// before storage in the DB
}
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Controlling What the Browser is allowed to Load / Process / Execute

CONTENT SECURITY POLICY

Content Security Policy (CSP)

- Loading page resources (images, CSS, JS) from places different from the original website (different “origins”) without restrictions is a **huge security risk**
- Modern browsers **can be told what is safe to load / execute** directly when loading the first page of the website
  - Via the Content-Security-Policy HTTP header
  - Example: images from any origin, audio/video (media) from two origins, and scripts only from one specific origin

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

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Content Security Policy (CSP)

- ☺ **Enforced** by the browser since the first load of the page
- ☺ Does **not** require configuration of **other web servers**
- ☺ Can **mitigate** cross-site scripting (**XSS**) attacks which exploit the browser's trust in the content received from the server
  - **Ignores** all scripts not coming from allowed domains, including **inline scripts** and event-handling HTML attributes
- ☹ May be **complex** to configure in presence of many servers and data sources

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

CSP with Express

https://www.npmjs.com/package/express-csp-header

- Use a good middleware such as `express-csp-header`
  - `npm install express-csp-header`

```
const { expressCspHeader, INLINE, NONE, SELF } = require('express-csp-header');
…
app.use(expressCspHeader({
    directives: {
      'default-src': [SELF],
      'script-src': [SELF, 'unpkg.com', 'cdn.jsdelivr.net'],  // No INLINE !
      'style-src': [SELF, 'cdn.jsdelivr.net'],
      'font-src': [SELF, 'cdn.jsdelivr.net'],
    }
}));
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

CSP without Express

- In case you do not control the server, as in files served by a CDN, it can be inserted as a `<meta>` tag in the `<head>` section

```
<meta http-equiv="Content-Security-Policy"
   content="default-src 'self'; img-src https://*; child-src 'none';" />
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Summary

- **General rule**: data and data sources (e.g., window.location) coming from outside the application **must be treated as dangerous**
- Validate / escape / sanitize data before storing whenever possible (i.e., when preserving original data is not needed)
- When data is returned or processed by the browser, data MUST be
  - Either already safe (because of the previous actions)
  - Or escaped / sanitized for the context where it will be used
- Whenever possible, use Content Security Policy (CSP) and similar browser-level HTTP security mechanisms

References

- XSS Attacks
  - https://owasp.org/www-community/attacks/xss/
  - https://owasp.org/www-community/Types_of_Cross-Site_Scripting
- XSS Protection Cheat Sheet
  - https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
  - https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html
  - https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/