

Web Development and API Design

Lesson 09: WebSockets and XSS

Prof. Andrea Arcuri

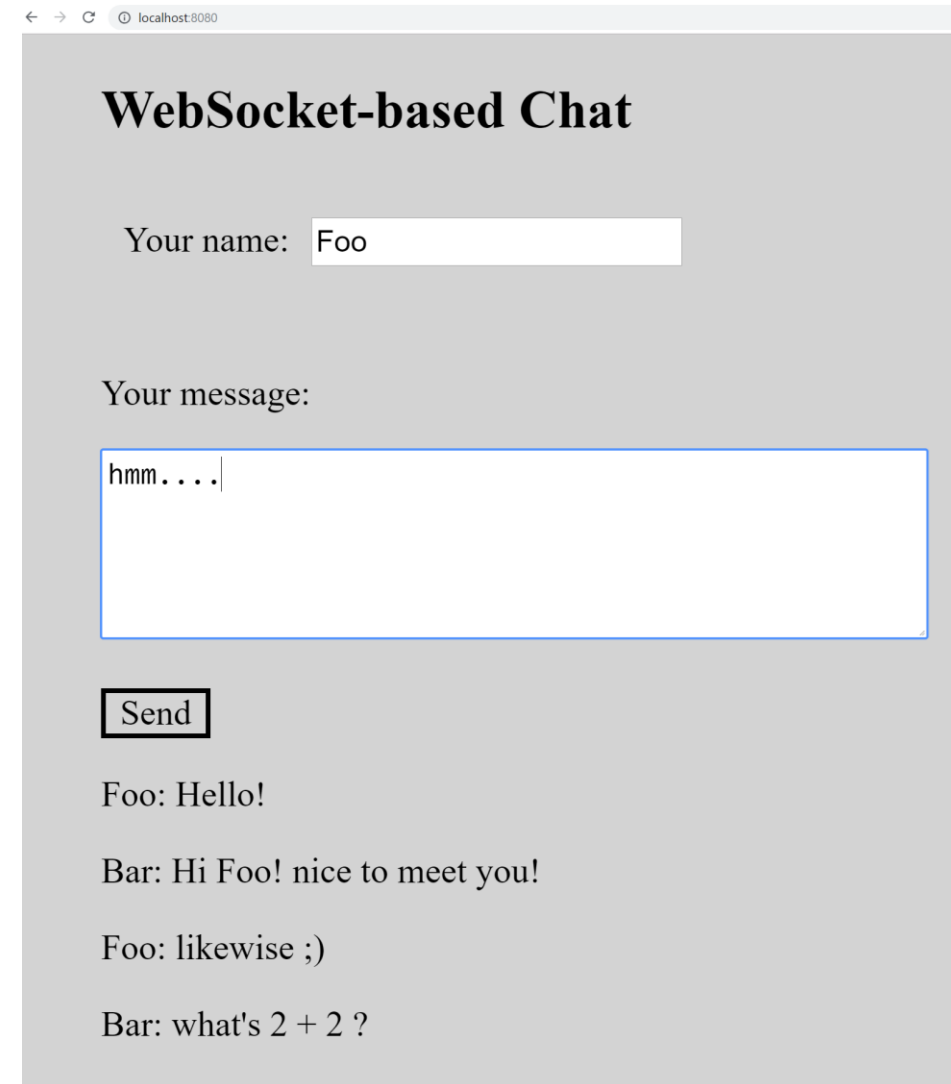
Goals

- Understand what is the problem that WebSockets solve
- Learn how to add WebSocket support to a React/NodeJS app
- Revise knowledge on user-input sanitization and escaping
- Revise knowledge on XSS attacks
- Understand how libraries/frameworks like React help to prevent some XSS, **but not all!!!**

WebSockets

Chat Application

- How would you implement a chat app in a browser?
- It is not as simple as it sounds...



← → ↻ 🌐 localhost:8080

WebSocket-based Chat

Your name:

Your message:

Foo: Hello!

Bar: Hi Foo! nice to meet you!

Foo: likewise ;)

Bar: what's 2 + 2 ?

Option 1: Server-Side Templates

- GET HTML page with current messages
- Create new message with a POST form submission, returning the updated HTML page
- *Issue 0*: download all messages even if only 1 new is created
- *Issue 1*: current user will not see the new messages of other users until s/he interacts with the app
 - eg, reload page or post new message

Option 2: AJAX Polling

- Use AJAX to fetch list of only the new messages to display
- Repeat AJAX calls in a loop, eg every X milliseconds
- *Issue 0*: might have to wait up to X ms before seeing the new messages from other users
- *Issue 1*: if no new messages, all these AJAX requests are a huge waste of bandwidth
- Choosing X is a tradeoff between Issue 0 and 1
 - eg, small X improves usability, but at a huge bandwidth waste cost

Option 3: WebSockets

- Besides HTTP, establish a WS connection
 - most browsers do support WS
- WS enables duplex communications
 - server can decide to send data to browser, which will listens to updates
- Server will keep an active TCP connection for each client
- When new message, server can *broadcast* it to all clients
- Browser just waits for notifications, and update HTML when it receives incoming messages from server
- Server *pushes* data only when available
 - no bandwidth waste

WebSocket Protocol

- Usually over TCP
- It is **NOT** HTTP, but *first message* has same syntax as HTTP
- Note the different protocol in the URL, eg
ws://localhost:8080
 - **wss** is for encrypted, like HTTPS

Name	× Headers Frames Timing
localhost	
style.css	
bundle.js	
messages	
localhost	

▼ General	
Request URL:	ws://localhost:8080/
Request Method:	GET
Status Code:	🟢 101 Switching Protocols

▼ Response Headers	view source
Connection:	Upgrade
Sec-WebSocket-Accept:	gURaMdcj1pOCmRq/TCT1OHmTFEk=
Upgrade:	websocket

▼ Request Headers	view source
Accept-Encoding:	gzip, deflate, br
Accept-Language:	en-US,en;q=0.9
Cache-Control:	no-cache
Connection:	Upgrade
Host:	localhost:8080
Origin:	http://localhost:8080
Pragma:	no-cache
Sec-WebSocket-Extensions:	permessage-deflate; client_max_window_bits
Sec-WebSocket-Key:	ZJqShbnFas262GwZ9sxANG==
Sec-WebSocket-Version:	13
Upgrade:	websocket
User-Agent:	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.0) Chrome/72.0.3626.119 Safari/537.36

Request ws://localhost:8080

- When making a request using WS protocol, browser will craft a message with same syntax as HTTP, with following headers
- **Sec-WebSocket-Extensions**
 - specify some WS extensions to use during the communications, like how to compress the messages, eg, *permessage-deflate* tells to use the “*deflate*” compression algorithm
- **Sec-WebSocket-Key**
 - needed to tell the server that this is indeed a WS connection, and not a HTTP one
 - using a random key
- **Sec-WebSocket-Version**
 - tell the server which version of WS protocol the browser is using
- **Upgrade: websocket**
 - standard HTTP header, telling that, although this request was handled like HTTP, the client (ie browser) wants to switch to a different protocol (WS in this case)

Server Response

- If server supports and accepts the WS connection, it will answer with a HTTP message having the following
 - **Connection: Upgrade**
 - tell browser to update the connection from current HTTP to something else
 - **Upgrade: websocket**
 - the protocol to use for all following requests
 - **Sec-WebSocket-Accept**
 - used to confirm that server is willing to use WS protocol for all following requests
 - it contains the hashed key sent by the browser. Useful to prevent caches to resend previous WS conversations
 - HTTP status code **101**
 - it represents “*Switching Protocols*”

Established WS Connection

- Once WS is established, can send blocks of byte data or strings over TCP
- Can wait for receiving messages
 - duplex communications between browser and server
 - data split and sent as “*frames*” of bytes, with special codes to specify sequences of frames belonging to the same message
- How to structure messages is up to you
 - eg, could use protocols like STOMP
- Typically, *we will just send JSON objects, serialized as strings*

Why First Message in HTTP?

- It allows server to have a single listening TCP socket
 - eg, either 80 or 443, serving both HTTP(S) and WS(S)
- Easy to integrate in current web infrastructures, including **reverse-proxies**
 - often you do not speak directly with a server, but rather with proxies and gateways in front of them... but this is not something we will see in this course
- WS is younger than HTTP
 - first version in Chrome in 2009
- Needed an easy way to integrate the new WS protocol in the existing web infrastructures tailored for HTTP

WebSocket in the Browser

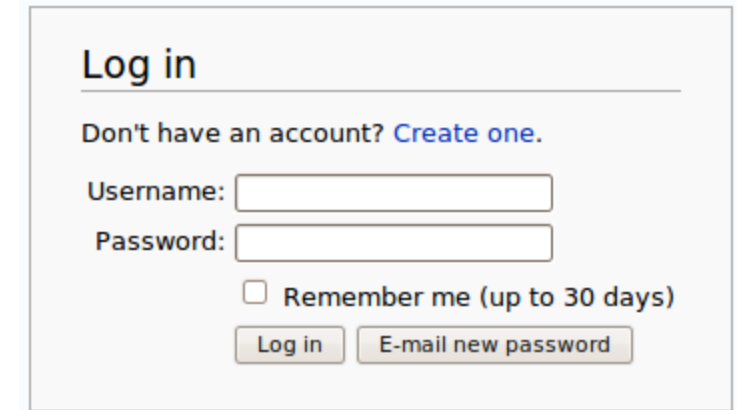
- In JavaScript, can use the *WebSocket* class from global scope
 - Most browsers nowadays support WS
- **WebSocket(url)**
 - create a WS object, trying to connect to the given URL of the server
 - recall to use either “ws” or “wss” as protocol, and not “http”
- **WebSocket.send(payload)**
 - send the given payload (e.g., a string) to the server
- **WebSocket.onmessage**
 - callback used to handle messages from server
- **WebSocket.close()**
 - to close the connection

WebSocket in the Server

- Backend support for WS depends on the programming language and libraries we use
- In this course, we will use the library “*ws*”, and “*express-ws*” to integrate it with Express
- In Express, we will have an endpoint dealing with the “*ws://*” protocol
- When called, a WS object will be created, on which we can register callbacks for incoming messages, open/close events, send messages to browser, broadcast to all users, etc.

Data Escaping/Sanitization

HTML Form Data



Log in

Don't have an account? [Create one.](#)

Username:

Password:

☐ Remember me (up to 30 days)

- How is data sent in a HTML Form?
- What is the structure of payload of the HTTP POST request?
- JSON? eg `{"username": "foo", "password": 123}`
- XML? eg
`<data><username>foo</username><password>123</password></data>`

x-www-form-urlencoded

- For textual data, like inputs in a HTML form
 - For binary data like file uploads, can use *multipart/form-data*
- Old format which is part of the HTML specs
 - <https://www.w3.org/TR/html/sec-forms.html#urlencoded-form-data>
- Each form element is represented with a pair *<name>=<value>*, where each pair is separated by a **&**
- Eg.: *username=foo&password=123*

What if values contain “=” or “&”?

- Eg, password: “123&bar=7”
- (Wrong) result: username=foo&**password=123**&*bar=7*
- The “*bar=7*” would be wrongly treated as a third input variable called “*bar*” with value “7”, and not be part of the “password” value

Solution: Special Encoding

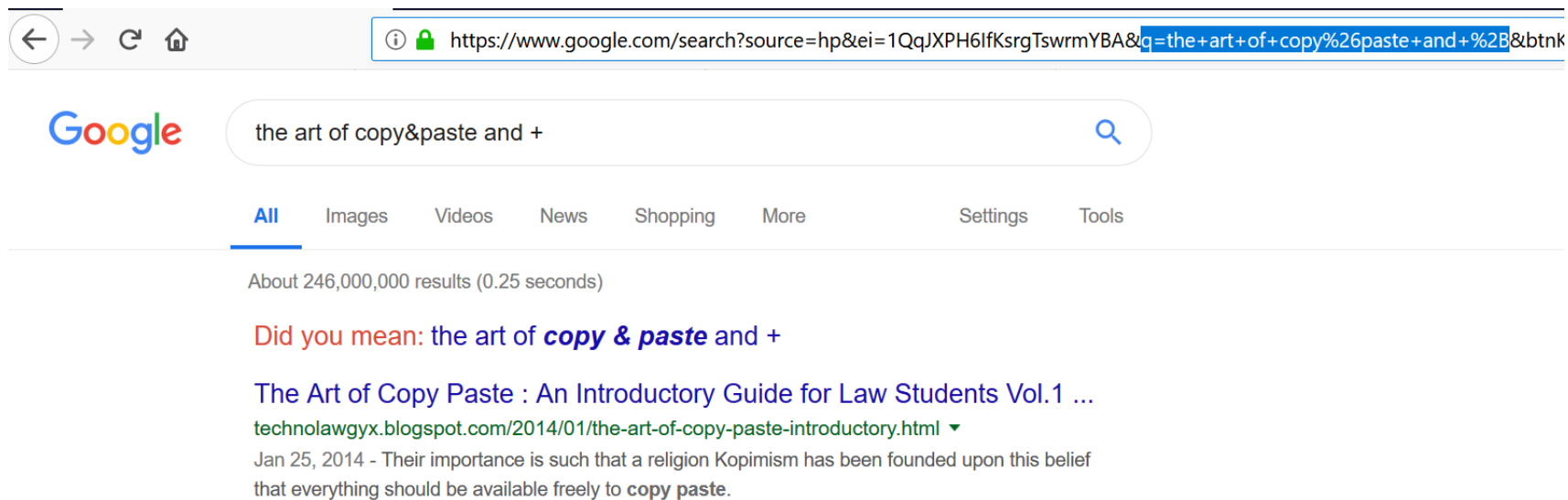
- Stay same: “*”, “-”, “.”, “_”, 0-9, a-z, A-Z
- Space “ ” becomes a “+”
- The rest become “%HH”, a percent sign and two hexadecimal digits representing the code of the character (default UTF-8)
- So, “123&bar=7” becomes “123%**26**bar%**3D**7”
- %26 = $(2 * 16) + 6 = 38$, which is the code for **&** in ASCII
- %3D = $(3 * 16) + 13 = 61$, which is the code for **=** in ASCII
 - Recall, hexadecimal D=13 (A=10,..., F=15)

But...

- What if I have a “%” in my values? Would not that mess up the decoding?
- E.g, password=“%3D”, don’t want to be wrongly treated as a “=”
- Not an issue, as symbol “%” is encoded based on its ASCII code 37, ie “%253D”
 - $\%25 = (2 * 16) + 5 = 37$

URLs and Query Parameters

- Query parameters in a URL are sequences of *<key>=<value>* pairs, separated by the symbol **&**
- What if a key or a value need to use special symbols like = or **&**?
- Those will be escaped as well, using the same kind of *%HH* escaping used in HTML forms
 - one difference though: “ ” empty char will be replaced with a “+”, whereas the symbol “+” is escaped with %2B
 - %2B = $(2 \times 16) + 11 = 43$, which is the ASCII code for +



- Assume in Google you search for “*the art of copy&paste and +*”
- The browser will make a GET request with query parameters, including the pair:
q=the+art+of+copy%26paste+and+%2B
- Notice how empty spaces are replaced with +, & with %26, and + with %2B

Text Transformations

- We can represent text in various formats, eg, HTML, XML, JSON, *x-www-form-urlencoded*
- Such formats use special symbols to define *structures* of the document
 - eg = and & in HTML form data, and <> in HTML/XML documents
- Input text values should NOT use those special structure/syntax symbols
- Need to be *transformed* (aka *escaped*) into non-structure symbols
 - & into %26, and = into %3D in HTML form data

What About HTML???

← → ↻ localhost:63342/pg6300/les09/escape/escaped.html?_ijt=rtpl438qru5sgqieih9ovd3kv5

How to represent the symbols of a tag with attribute without getting them interpreted as HTML tags?
For example:

[Foo](#)

vs.

`Foo`

However, what to escape depends on the context:

`"<p>"`

HTML/XML Escaping

- “&” followed by name (or code), closed by “;”
- **"** for “ (double quotation mark)
- **&** for & (ampersand)
- **'** for ‘ (apostrophe)
- **<** for < (less-than)
- **>** for > (greater-than)
- These are most common ones

See “escaped.html” file

`Foo`

vs.

`Foo`

What actually needs to be escaped depends on context

- `<div id=""<p>";">
 "<p>"
</div>`
- Representing “<p>” (quotes included)
- In attributes, quotes “ need to be escaped (**"**), but no need there for <>, as those latter are no string delimiters
- In node content, it is the other way round

XSS

User Content

- Text written by user which is displayed in the HTML pages when submitted (eg HTML form)
 - eg, Chats and Discussion Forums
 - but also showing back the search query when doing a search
- Also query parameters in URLs are a form of user input if crafted by an attacker
 - eg, `www.foo.com?x=10` if then value of x is displayed in the HTML
 - recall, attacker can use social engineering to trick a user to click on a link
- *What is the most important rule regarding user content given as input to a system???*

NEVER TRUST USER INPUTS!!!

NEVER

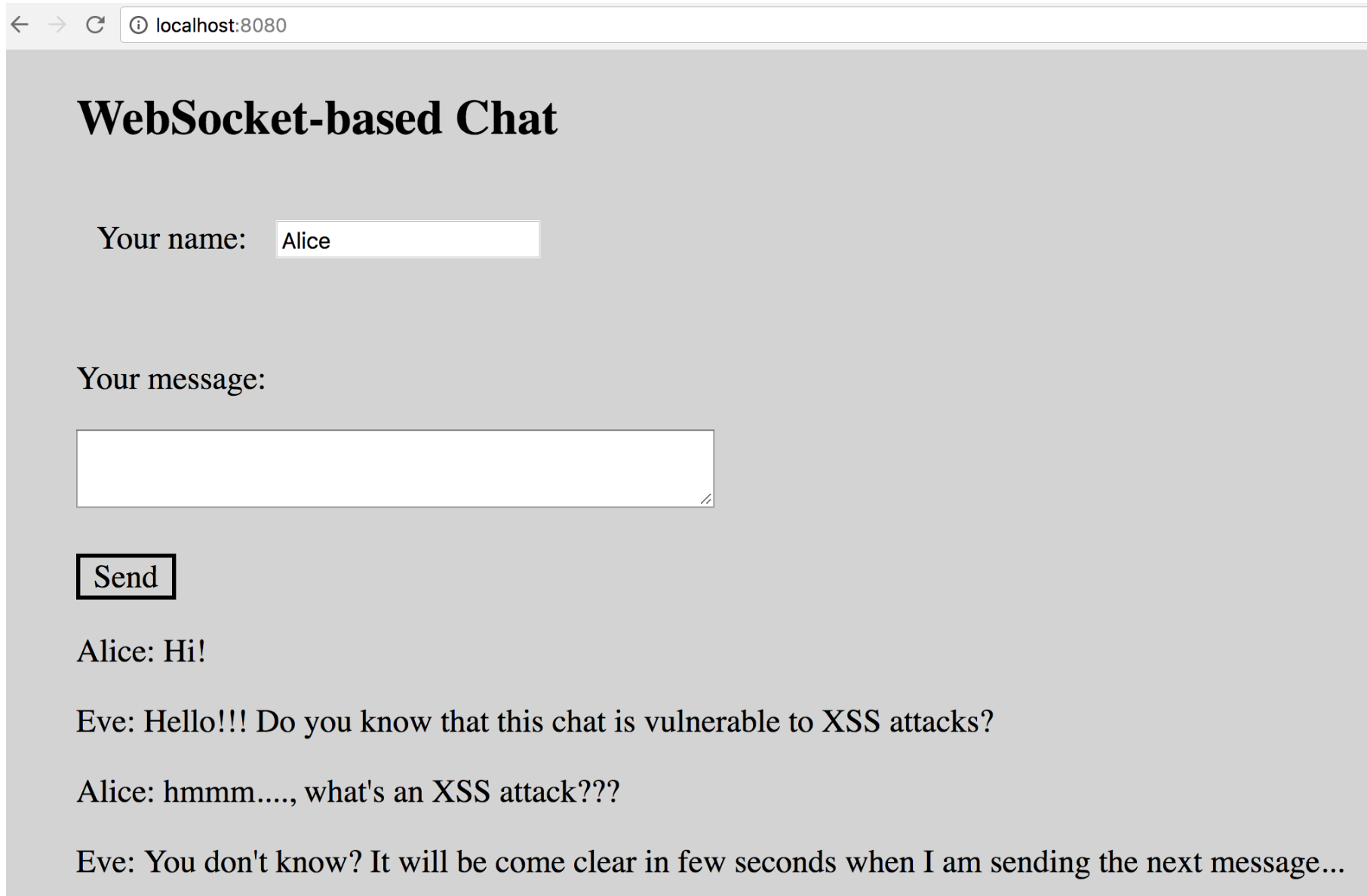
TRUST

USER

INPUTS!!!

NEVER TRUST USER INPUTS!!!

But Why???



← → ↻ ⓘ localhost:8080

WebSocket-based Chat

Your name:

Your message:

Alice: Hi!

Eve: Hello!!! Do you know that this chat is vulnerable to XSS attacks?

Alice: hmmm...., what's an XSS attack???

Eve: You don't know? It will be come clear in few seconds when I am sending the next message...

After Eve's message, chat program is gone on Alice's browser...



What was the problem?

```
let msgDiv = "<div>";

for(let i=0; i<messages.length; i++){
  const m = messages[i];
  //WARNING: this is exploitable by XSS!!!
  msgDiv += "<p>" + m.author + ": " + m.text + "</p>";
}
msgDiv += "</div>";
```

And the message sent was...

```
<img src='x'
```

```
  onerror="document.getElementsByTagName('body')[0].inne  
rHTML = &quot;<img  
src='<a href="https://upload.wikimedia.org/wikipedia/commons/thumb  
/6/6c/Pirate_Flag.svg/750px-Pirate_Flag.svg.png"/>&quot;;" />
```

String Concatenation

- `msgDiv += "<p>" + dto.author + ": " + dto.text + "</p>";`
- Should **NEVER** concatenate strings directly to generate HTML when such data comes from user
 - ie, that is a very, very bad example of handling user inputs
- If data is not escaped, could have HTML <tags> that are interpreted by browser as HTML commands
- Could execute JavaScript!!! And so do whatever you want on a page
- Eg., `dto.text = "<script>...</script>"`

Cross-site Scripting (XSS)

- Type of attack in which malicious JavaScript is injected into a web page
- One of the most common type of security vulnerability on the web
- Typically exploiting lack of escaping/sanitization of user inputs when generating HTML dynamically (both client and server side)
- XSS is particularly nasty, as it adds JavaScript in the current page... so CORS will not help you here

Browser Security

- Most browsers will not execute any `<script>` block that has been dynamically added to the page
 - eg, when changing the HTML by altering “*innerHTML*”
- But that is simply futile... because you can still create HTML tags with JS handlers that are executed immediately
- ``

What To Do?

- When dealing with user inputs, always need to escape/sanitize them before use
- This applies both client-side (JS) and server-side (Java, PHP, C#, etc.)
- There are many edge cases, so must use an *existing* library to sanitize the inputs
 - This will depend on the programming language and framework
 - Do NOT write your own escape/sanitize functions

XSS and React

React Sanitization

- XSS is such a huge problem that many libraries/frameworks for HTML DOM manipulation do some form of input sanitization by default
- E.g., consider in JSX: **<p>Your text: {this.state.userInput}</p>**
- ... and the **userInput** is **<a>**
- ... then, React will *automatically* change it into **<a>** when rendering the HTML
- So, any **<** or **>** in the value will not be interpreted as an HTML tag

Examples of XSS in React

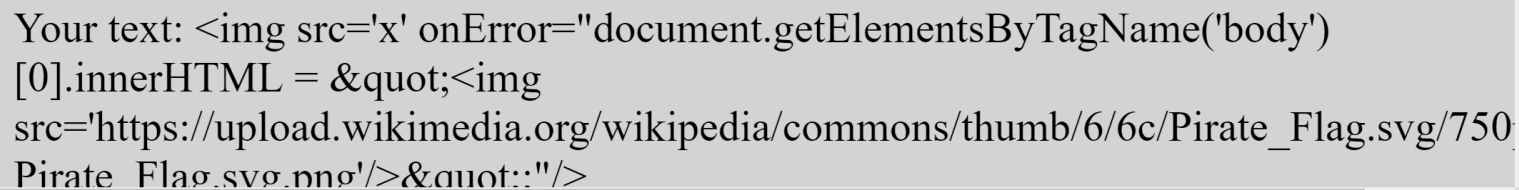
Link to your Homepage:

Your text:

```
<img src='x'
onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px-
Pirate_Flag.svg.png'/>&quot;;"/>
```

Displayed Values

[Link to homepage](#)

Your text: 

Elements

Console

Sources

Network

Performance

```
<!doctype html>
<html>
  <head>...</head>
  <body>
    <noscript>...</noscript>
    <div id="root">
      <div>
        <h2>Examples of XSS in React</h2>
        <div>...</div>
        <br>
        <div>...</div>
        <br>
        <hr>
        <h3>Displayed Values</h3>
        <a href= Link to homepage </a>
        <p>
          "Your text: "
          "...
          "<img src='x' onError="document.getElementsByTagName('body')[0].innerHTML
          = &quot;<img
          src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px-Pirate_Flag.svg.png'/>&quot;;"/>" == $0
          </p>
          <hr>
          <h3>Discussion</h3>
        </div>
      </div>
    </div>
  </body>
</html>
```

html

body

div#root

div

p (text)

Styles

Event Listeners

DOM Breakpoints

Properties

Accessibility

Filter

:hov .cls +

No matching selector or style

Filter

Show all

No matching property

Rendered Fonts

Times New Roman — Local file (202 glyphs)

Console

What's New

Note: CDT does not show you *raw* HTML by default, but you can see it by clicking for example “*Edit as HTML*”

Examples of XSS in React

Link to your Homepage:

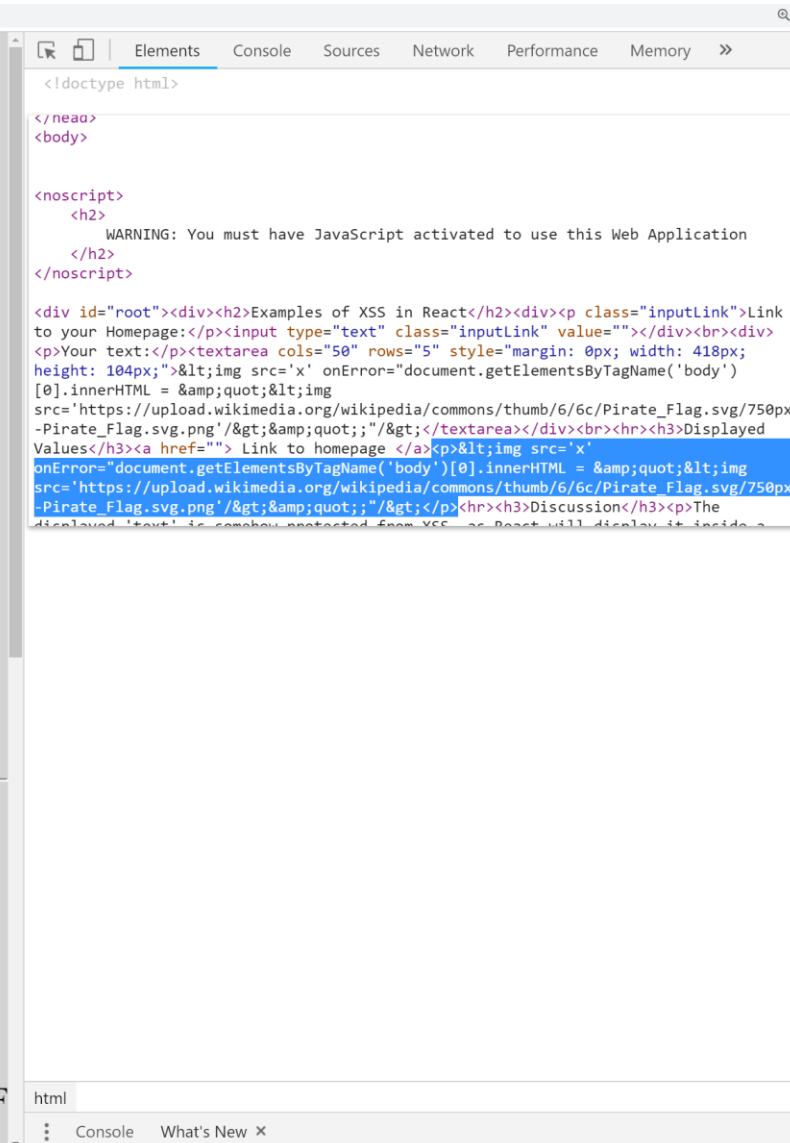
Your text:

```
<img src='x'
onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px-
Pirate_Flag.svg.png' />&quot;;"/>
```

Displayed Values

[Link to homepage](#)

```
<img src='x' onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_F
Pirate_Flag.svg.png' />&quot;;"/>
```



So, are you safe from XSS when
using React???

NO!!!

NO!!!

dangerouslySetInnerHTML

- React components have an attribute called **dangerouslySetInnerHTML** which enables to have raw HTML without escaping
 - note the word **dangerously** in its name...
- Even if you do not use it directly, it is a potential issue if you create attributes based on user inputs
- Eg: `<div {...jsonObjectComingFromUser} />`
- ... as one of those fields could be **dangerouslySetInnerHTML**

Escaping of Attributes

- Issue when you have attributes that are interpreted as URLs:
 - ``
 - `<link rel="import" href={user_supplied}>`
 - `<button formaction={user_supplied}>`
- Why are URLs a potential issue?

For example, type **javascript:alert('Hi!')** in the address-bar of your browser and see what happen...



` Link to homepage `
That is vulnerable to XSS when clicking the link!!!

localhost:8080 says
Welcome to XSS!

Examples of XSS in React

Link to your Homepage:

Your text:

Displayed Values

[Link to homepage](#)

```
localhost:8080  
Welcome to XSS!
```

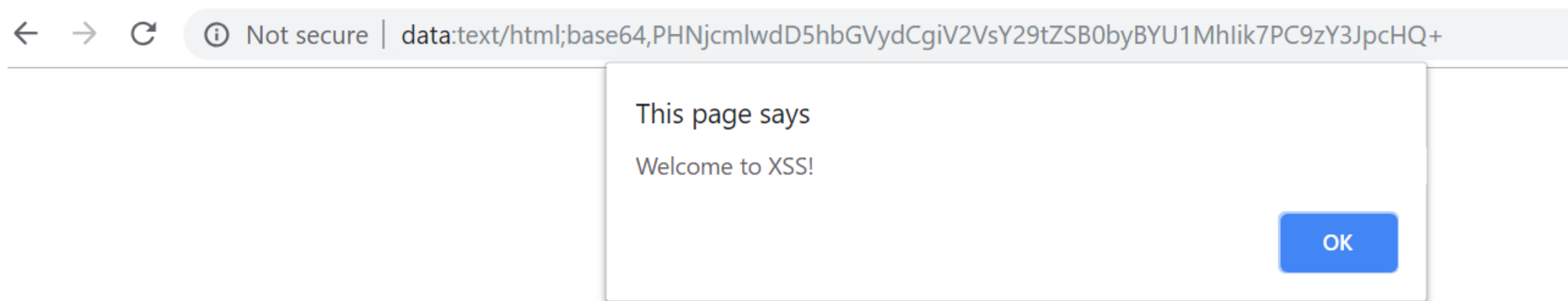
Elements Console Sources Network Performance Memory >>

```
html>  
$0  
/head>  
<body>  
  <noscript>...</noscript>  
  <div id="root">  
    <div>  
      <h2>Examples of XSS in React</h2>  
      <div>...</div>  
      <br>  
      <div>...</div>  
      <br>  
      <hr>  
      <h3>Displayed Values</h3>  
      <a href="javascript:alert('Welcome to XSS!')"> Link to homepage </a>  
      <p></p>  
      <hr>  
      <h3>Discussion</h3>  
      <p>...</p>  
      <p>...</p>  
      <p>...</p>  
      <p>...</p>  
    </div>  
  </div>  
  <script src="bundle.js"></script>  
</body>  
</html>
```

Sanitization

- In case of URLs, you need to manually sanitize the user inputs
 - eg, do not allow the “*javascript:*” protocol in the links
- *As a rule of thumb, shouldn't write your own sanitization functions, but rather use existing libraries*
 - however, if you do, use *whitelisting!!!* i.e., allow “*http:*” and “*https:*”, but block everything else... instead of *blacklisting* of just blocking “*javascript:*”
- For example, what do you think is going to happen if you use this string as URL??
**data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2VsY29tZS
B0byBYU1Mhlik7PC9zY3JpcHQ+**

Try it in the address-bar...



PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0byBYU1Mhlik7PC9zY3JpcHQ+ is the string **<script>alert("Welcome to XSS!");</script>**, encoded in the Base64 format

