Web Development and API Design

Lesson 09: WebSockets and XSS

Prof. Andrea Arcuri

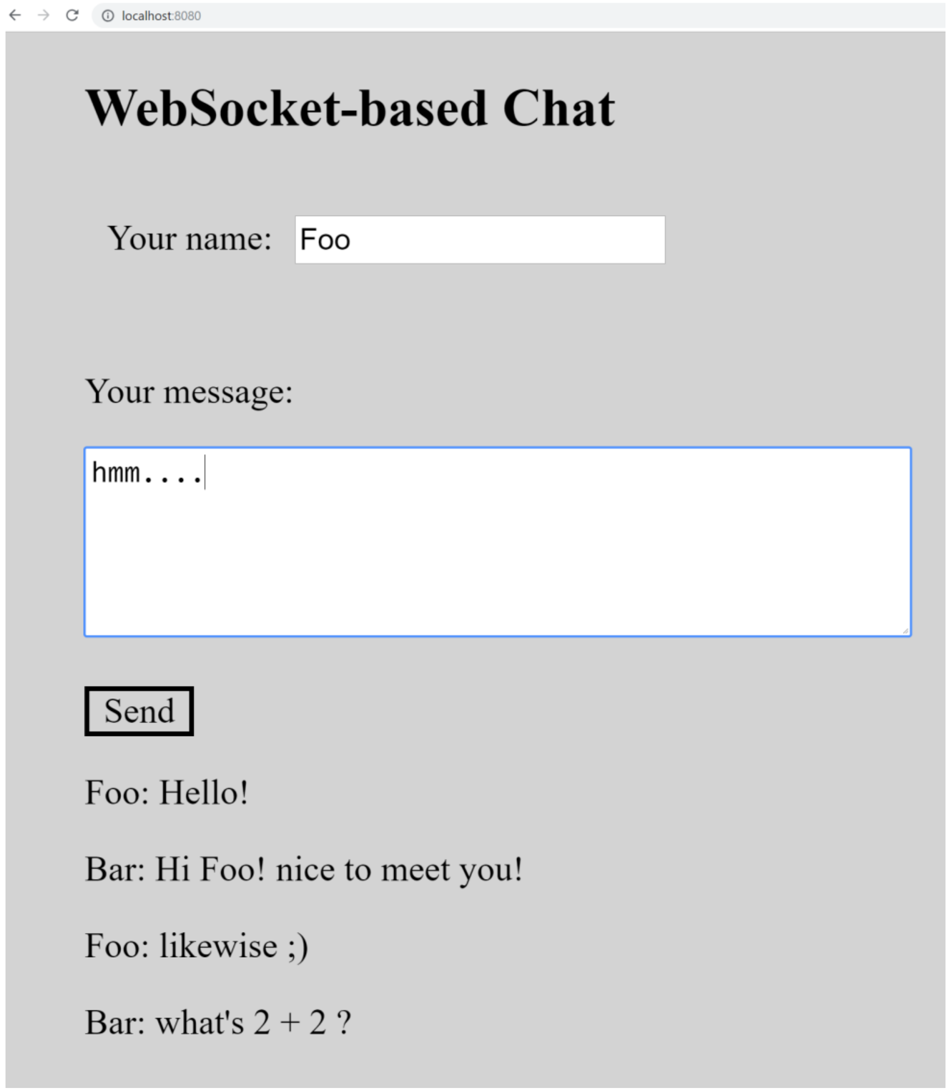
Goals

- 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: Foo

Your message:

hmm....

Send

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

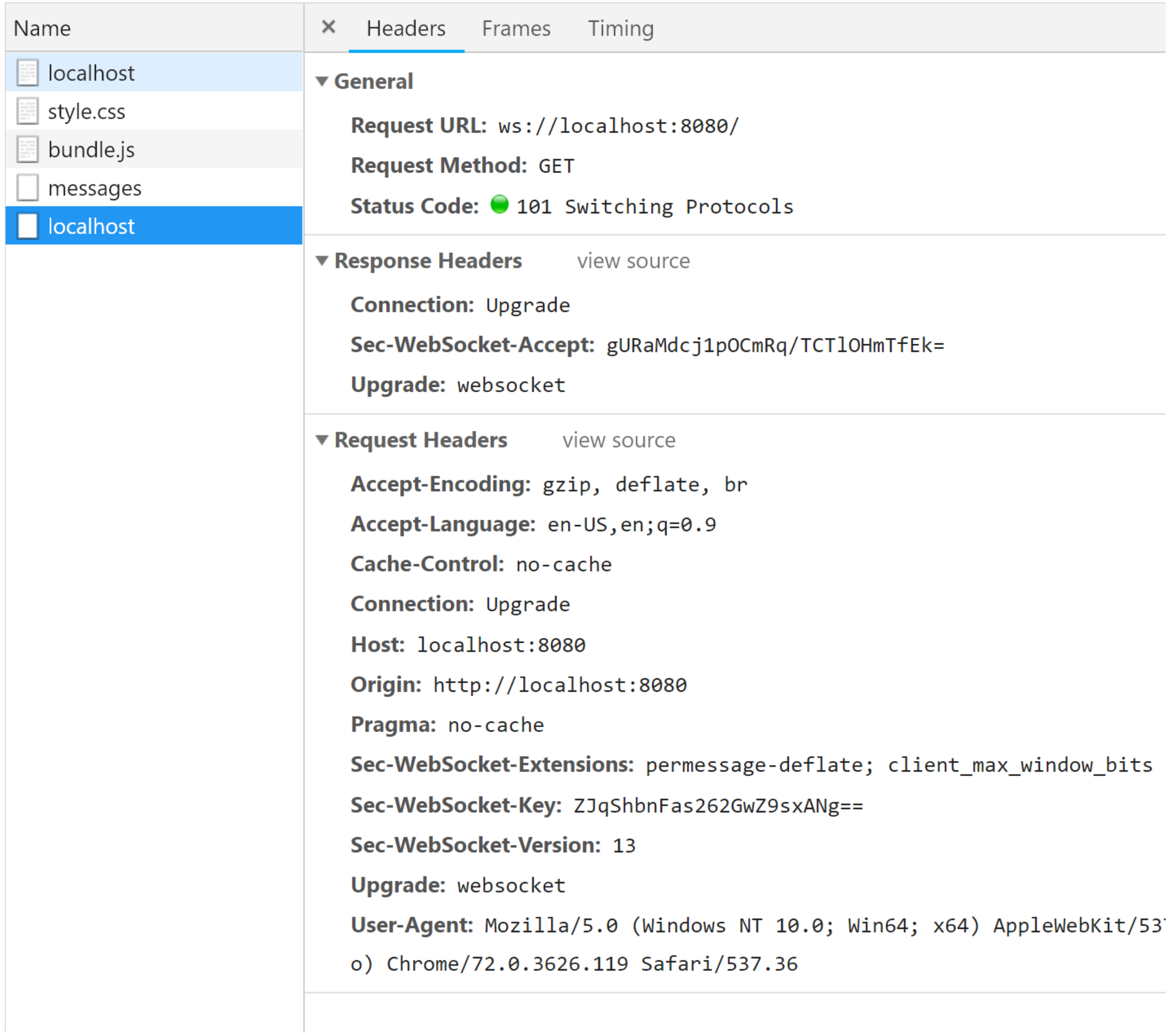
- Use AJAX to fetch list of only the new messages to display
- Repeat AJAX calls in a loop, eg every X milliseconds
- *Issue 0*: might have to wait up to X ms before seeing the new messages from other users
- *Issue 1*: if no new messages, all these AJAX requests are a huge waste of bandwidth
- Choosing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



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

▼ Response Headers     view source
  Connection: Upgrade
  Sec-WebSocket-Accept: gURaMdcj1pOCmRq/TCTlOHmTfEk=
  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/53
  o) 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

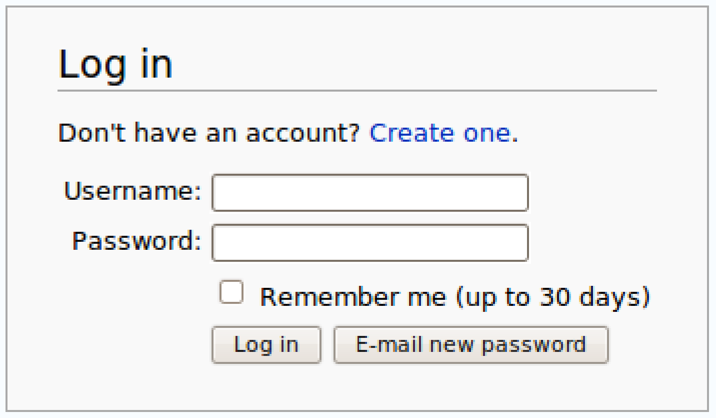
- In JavaScript, can use the *WebSocket* class from global scope
    - Most browsers nowadays support WS
- **WebSocket(url)**
    - create a WS object, trying to connect to the given URL of the server
    - recall to use either "*ws*" or "*wss*" as protocol, and not "*http*"
- **WebSocket.send(payload)**
    - 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)
Log in
E-mail new password

- 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

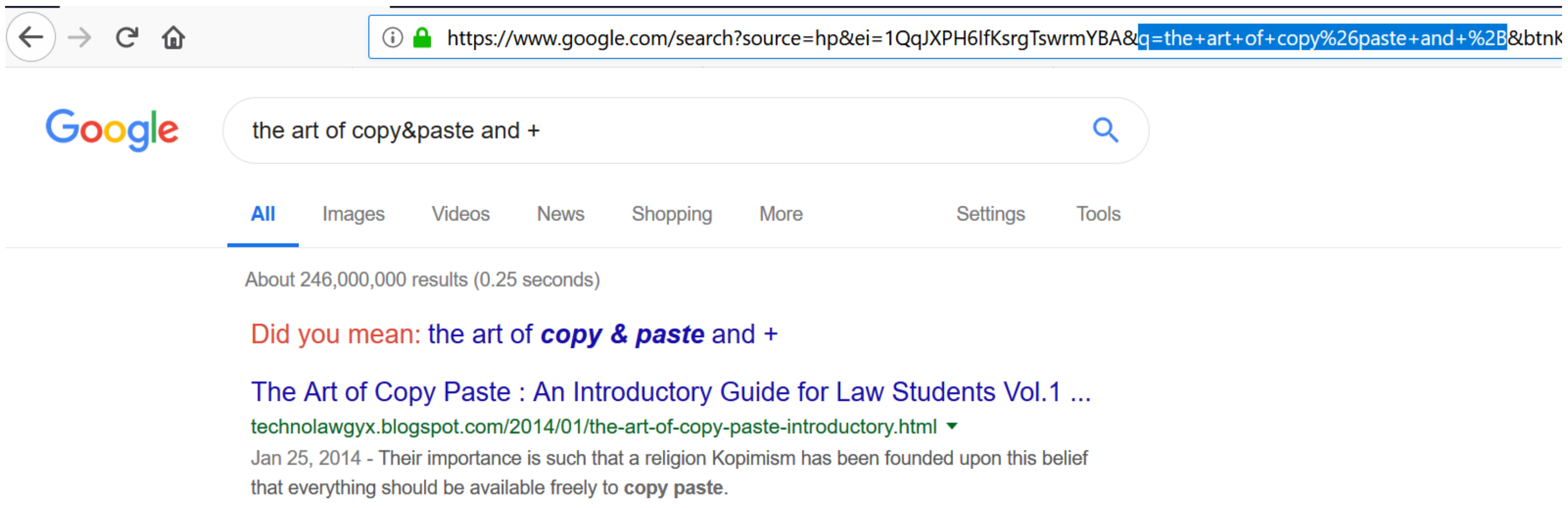
- Stay same: “*”, “-”, “.”, “_”, 0-9, a-z, A-Z
- Space “ ” becomes a “+”
- The rest become “%HH”, a percent sign and two hexadecimal digits representing the code of the character (default UTF-8)
- So, “123&bar=7” becomes “123**%26**bar**%3D**7”
- %26 = (2*16)+6 = 38, which is the code for **&** in ASCII
- %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 “*%25*3D”
  - %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*16) + 11 = 43 , which is the ASCII code for +


https://www.google.com/search?source=hp&ei=1QqJXPH6lfKsrgTswrmYBA&q=the+art+of+copy%26paste+and+%2B&btnk

the art of copy&paste and +

About 246,000,000 results (0.25 seconds)

Did you mean: the art of ***copy & paste*** and +

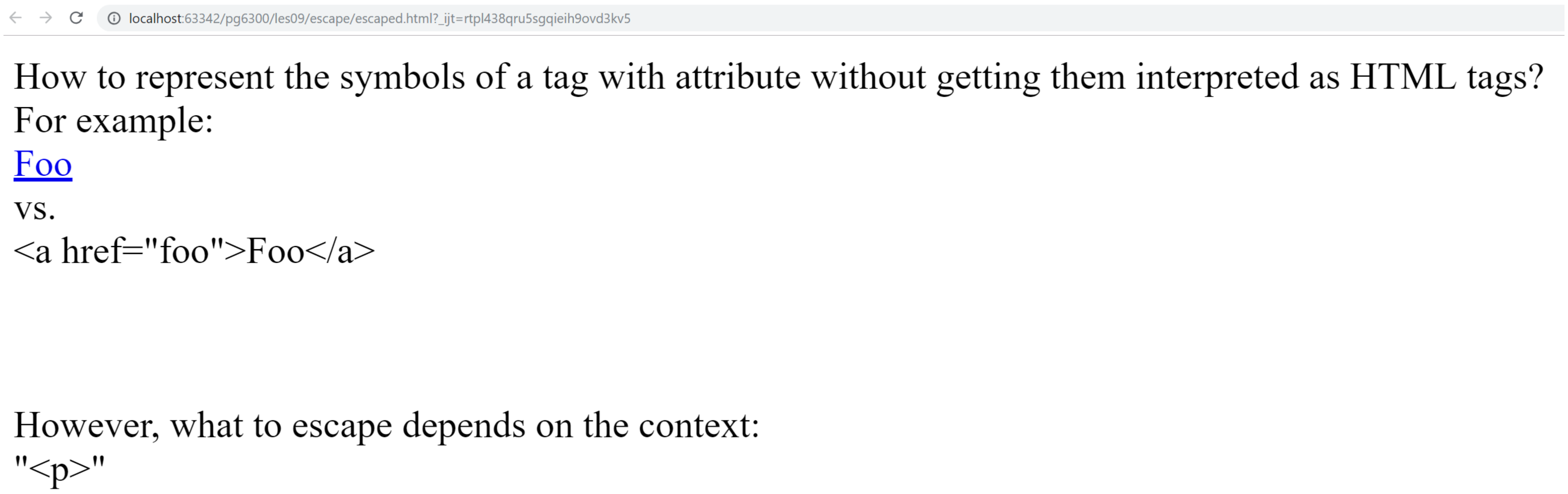
The Art of Copy Paste : An Introductory Guide for Law Students Vol.1 ...
technolawgyx.blogspot.com/2014/01/the-art-of-copy-paste-introductory.html
Jan 25, 2014 - Their importance is such that a religion Kopimism has been founded upon this belief that everything should be available freely to **copy paste**.

- 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**%26**paste**+**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.
<a href="foo">Foo</a>

However, what to escape depends on the context:
"<p>"

HTML/XML Escaping

- "**&**" followed by name (or code), closed by "**;**"
- **&quot;** for **"** (double quotation mark)
- **&amp;** for **&** (ampersand)
- **&apos;** for **'** (apostrophe)
- **&lt;** for **<** (less-than)
- **&gt;** for **>** (greater-than)
- These are most common ones

See “escaped.html” file

```
<a href="foo">Foo</a>
```

vs.

```
&lt;a href=&quot;foo&quot;&gt;Foo&lt;/a&gt;
```

What actually needs to be escaped depends on context

-
  ```
  <div id="&quot;<p>&quot;">
  "&lt;p&gt;"
  </div>
  ```
- Representing **“<p>”** (quotes included)
- In attributes, quotes “ need to be escaped (**&quot;**), 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

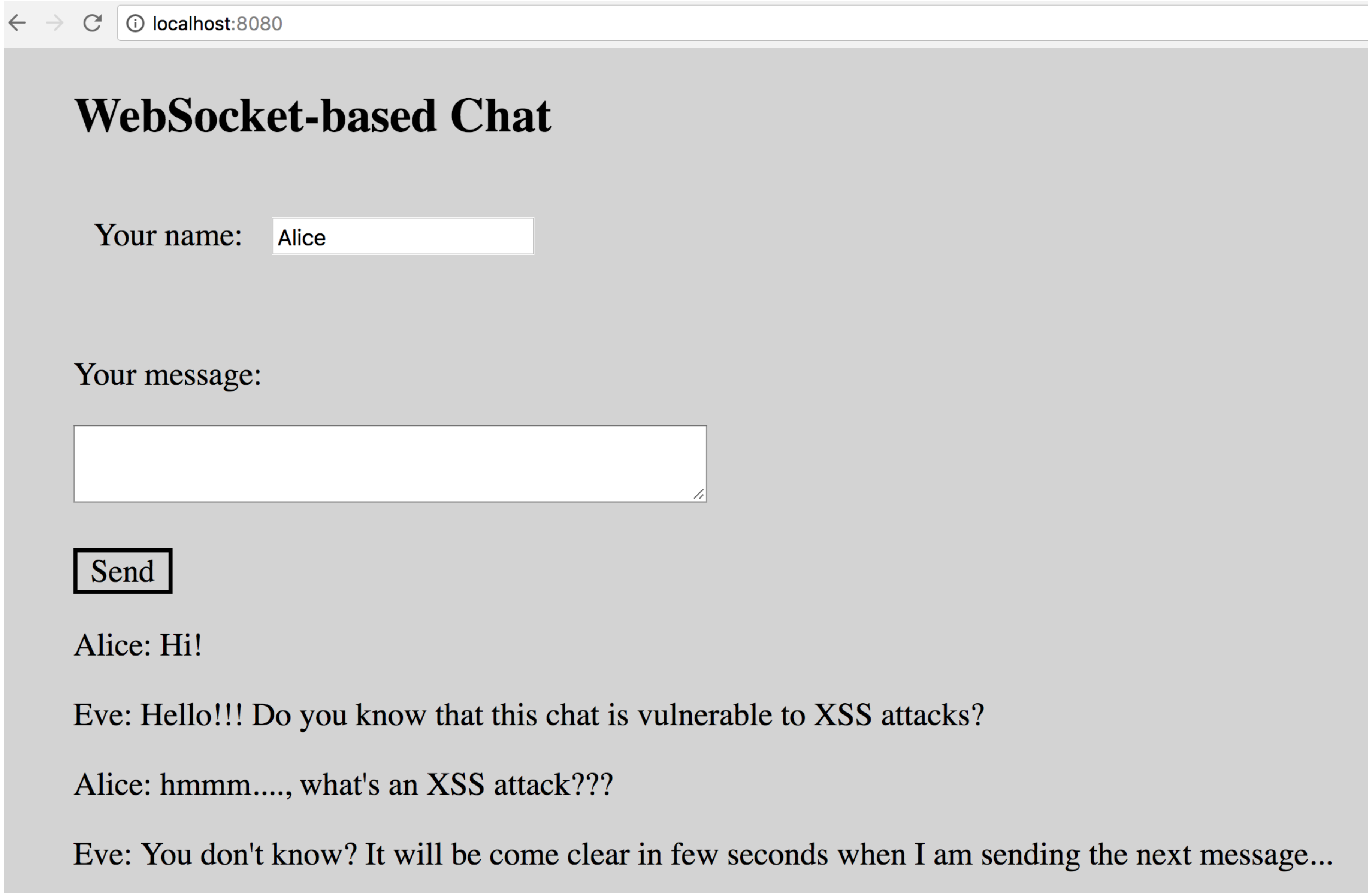
TRUST

USER

INPUTS!!!

NEVER TRUST USER INPUTS!!!

But Why???



localhost:8080

WebSocket-based Chat

Your name: Alice

Your message:

Send

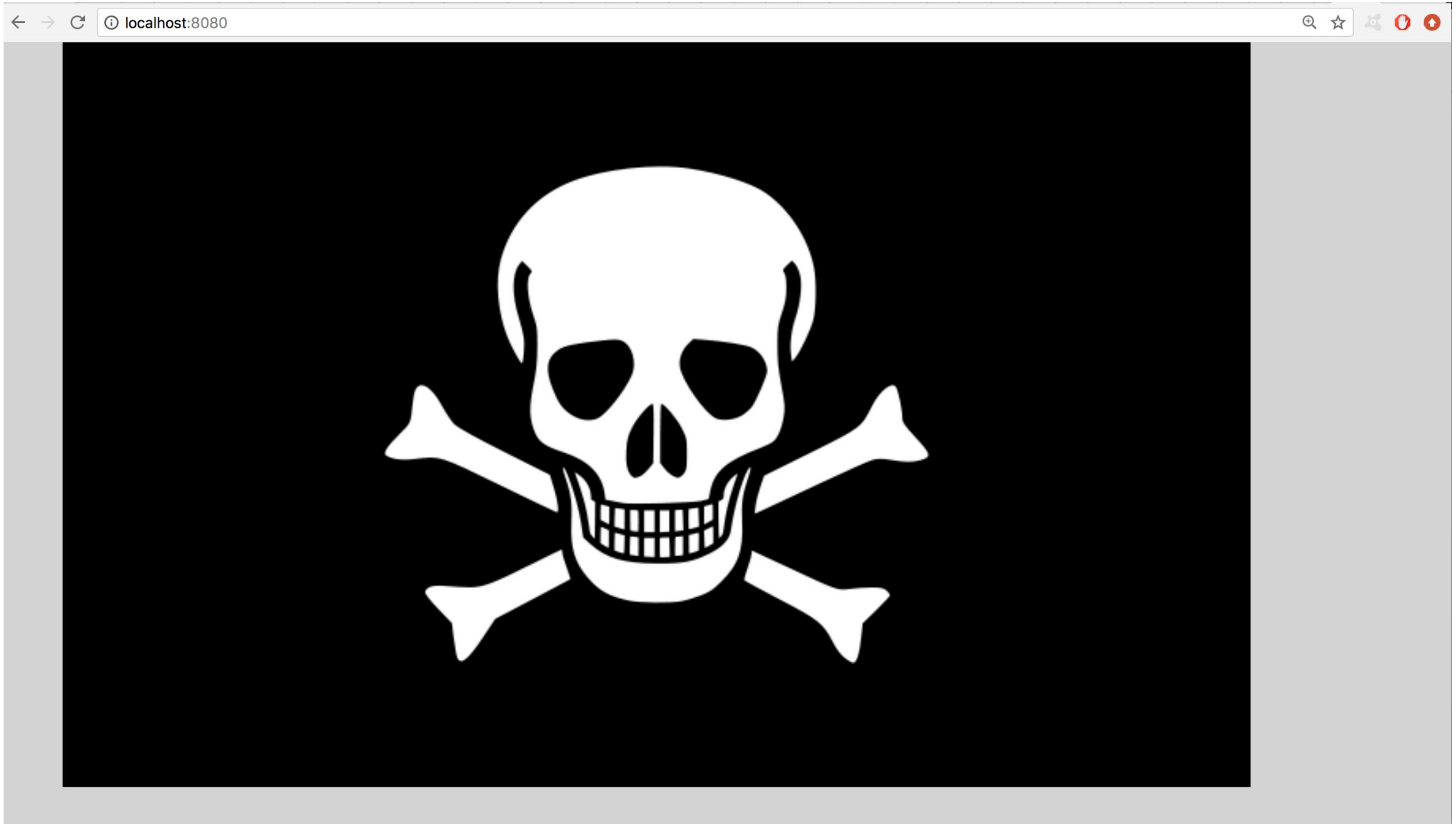
Alice: Hi!

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='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
- *<img src='aURLthatNotExist' **onerror**="… JS here…">*

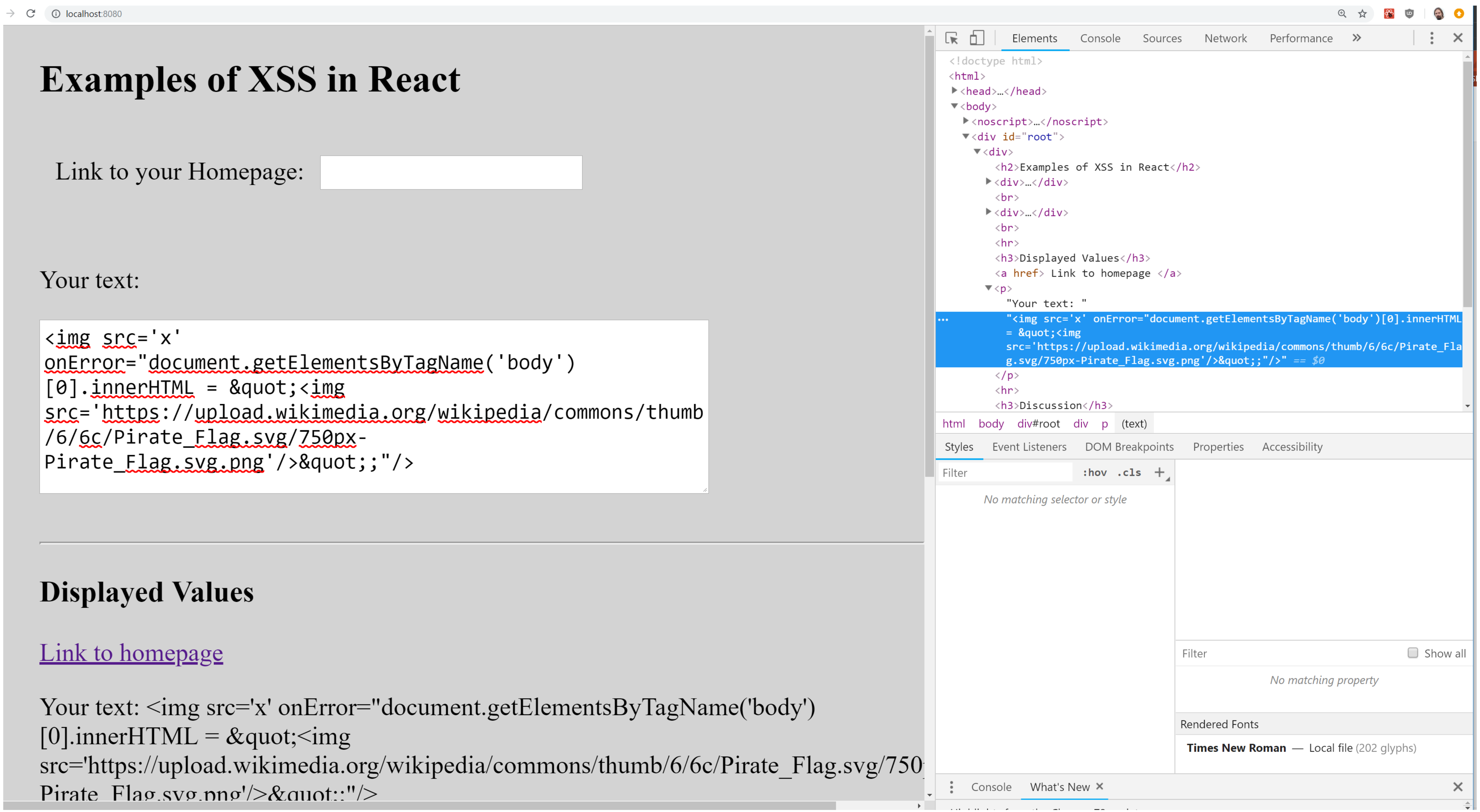
What To Do?

- 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 **&lt;a&gt;** 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

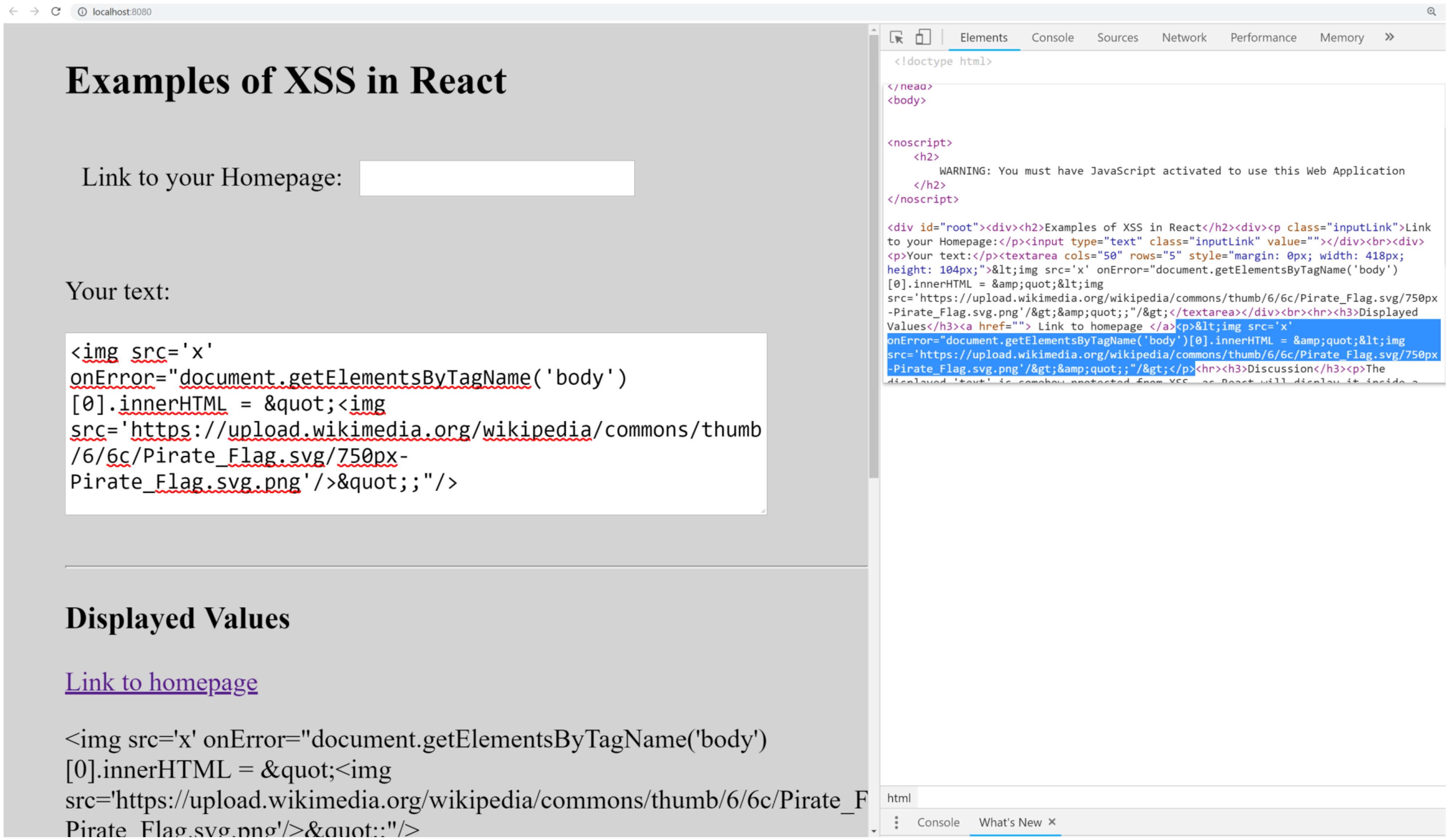
Link to homepage

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

```
<!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_Fla
           g.svg/750px-Pirate_Flag.svg.png'/>&quot;;"/>" == $0
         </p>
         <hr>
         <h3>Discussion</h3>
```

html body div#root div p (text)

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;;"/>

```
<!doctype html>
</head>
<body>


<noscript>
    <h2>
        WARNING: You must have JavaScript activated to use this Web Application
    </h2>
</noscript>

<div id="root"><div><h2>Examples of XSS in React</h2><div><p class="inputLink">Link
to your Homepage:</p><input type="text" class="inputLink" value=""></div><br><div>
<p>Your text:</p><textarea cols="50" rows="5" style="margin: 0px; width: 418px;
height: 104px;">&lt;img src='x' onError="document.getElementsByTagName('body')
[0].innerHTML = &amp;quot;&lt;img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px
-Pirate_Flag.svg.png'/&gt;&amp;quot;;"/&gt;</textarea></div><br><hr><h3>Displayed
Values</h3><a href=""> Link to homepage </a><p>&lt;img src='x'
onError="document.getElementsByTagName('body')[0].innerHTML = &amp;quot;&lt;img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px
-Pirate_Flag.svg.png'/&gt;&amp;quot;;"/&gt;</p><hr><h3>Discussion</h3><p>The
```

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

NO!!!

NO!!!

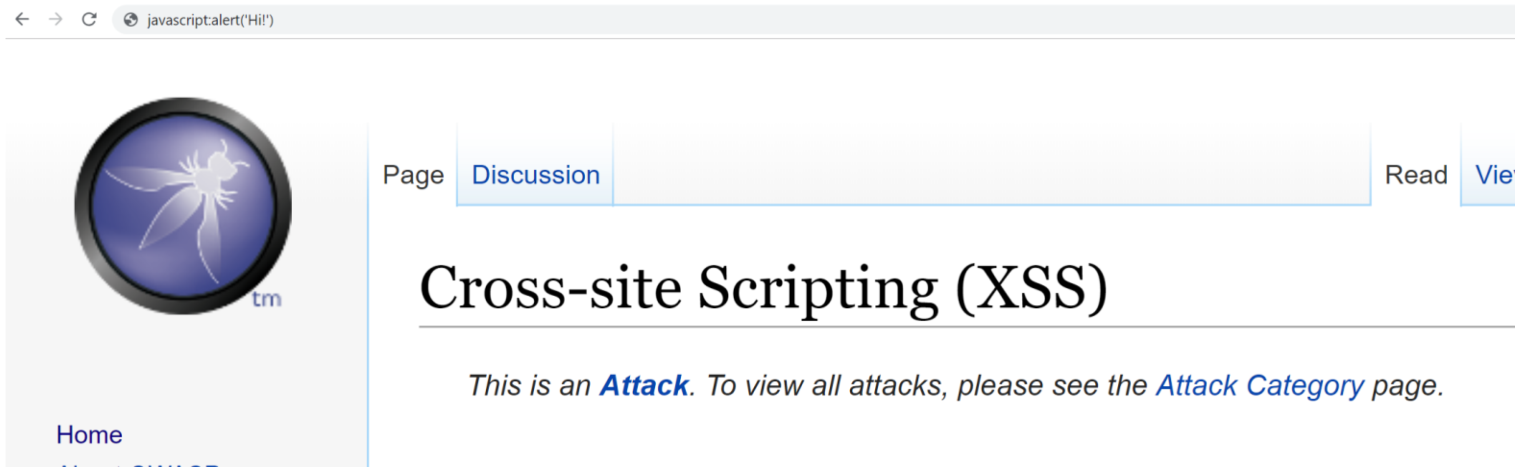
dangerouslySetInnerHTML

- 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:
  - **<a href={user_supplied} / >**
  - **<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…



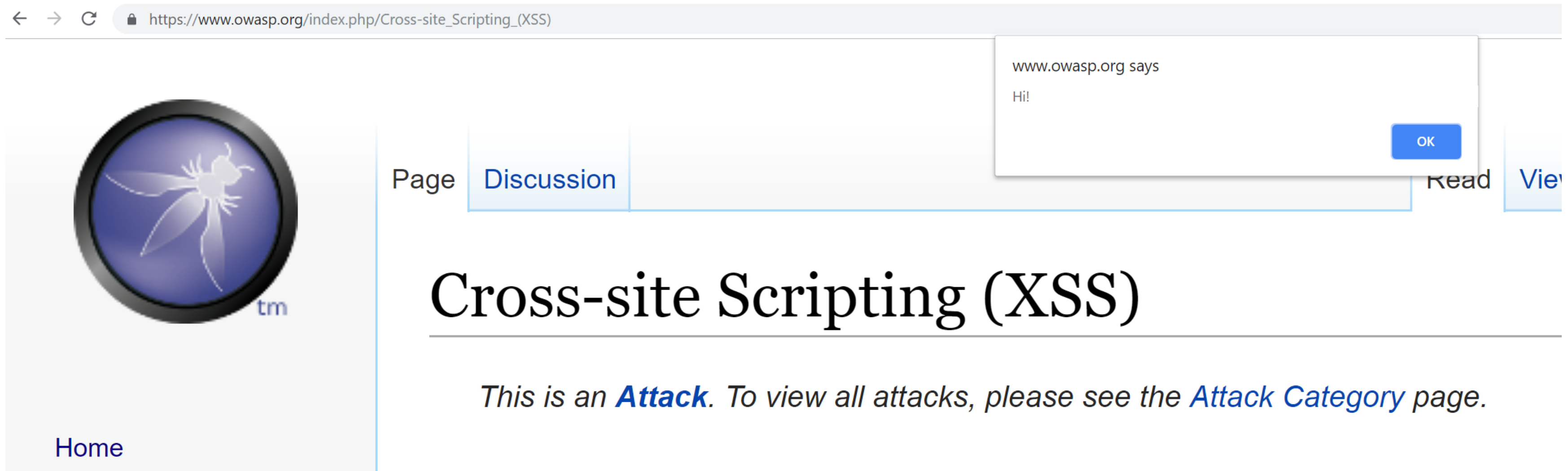
javascript:alert('Hi!')

Page Discussion Read Vie

Cross-site Scripting (XSS)

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Home



https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

www.owasp.org says

Hi!

OK

Page Discussion Read Vie

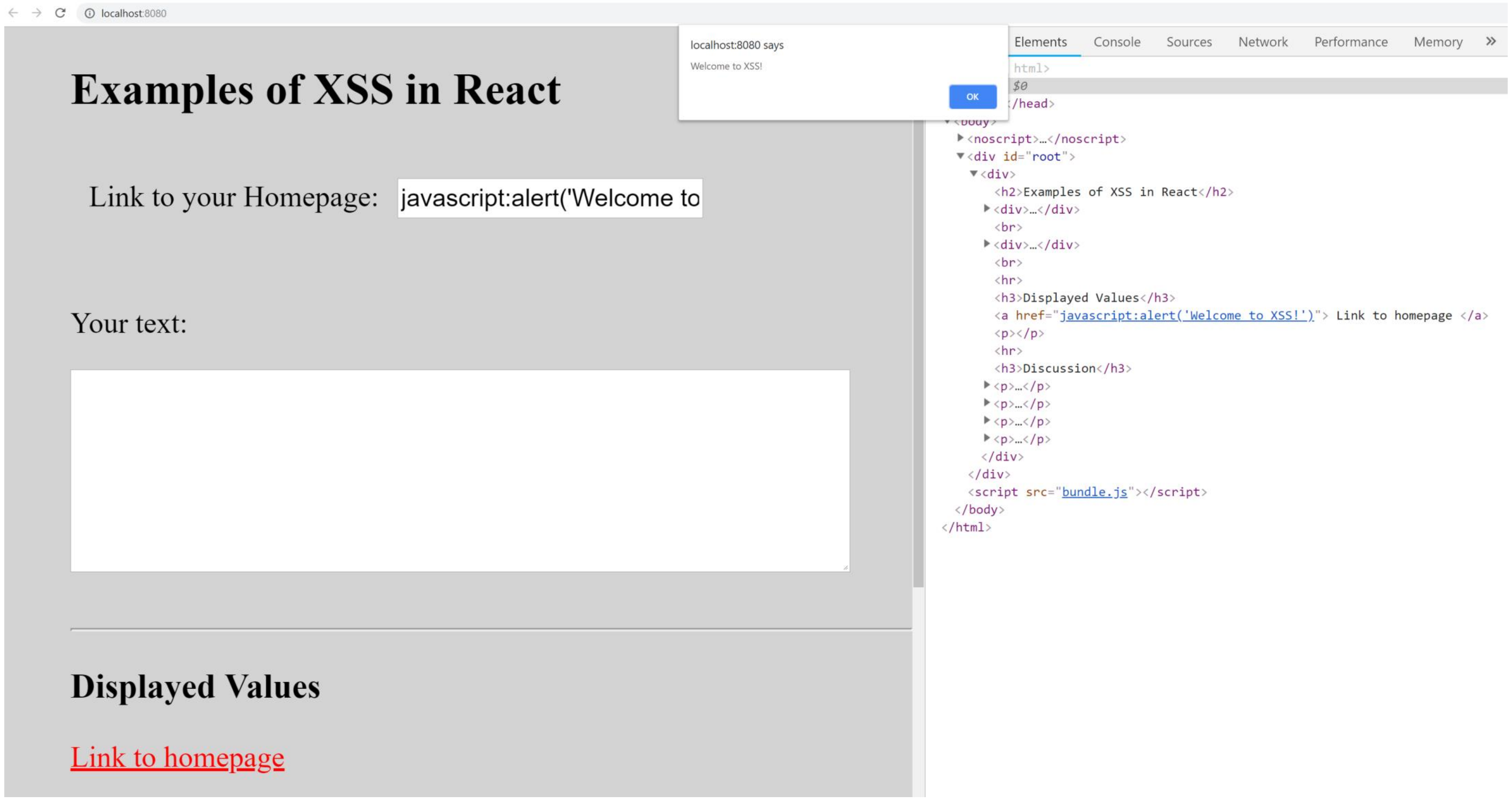
Cross-site Scripting (XSS)

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Home

<a href={this.state.homepageLink} > Link to homepage </a>

That is vulnerable to XSS when clicking the link!!!



localhost:8080 says

Welcome to XSS!

Examples of XSS in React

Link to your Homepage: javascript:alert('Welcome to

Your text:

Displayed Values

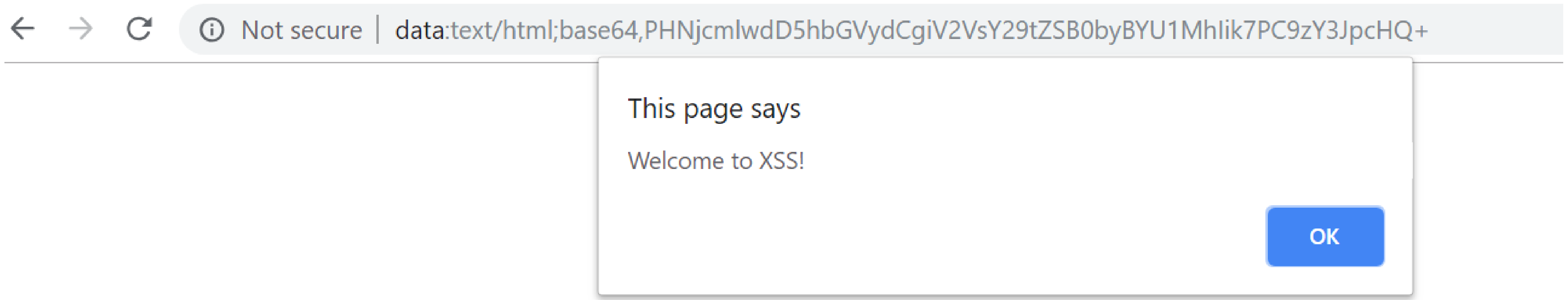
Link to homepage

```
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
  - 2020 note: future versions of *React* will block it
- *As a rule of thumb, shouldn’t write your own sanitization functions, but rather use existing libraries*
  - however, if you do, use *whitelisting*!!! Ie., 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,PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0by BYU1MhIik7PC9zY3JpcHQ+

Try it in the address-bar...

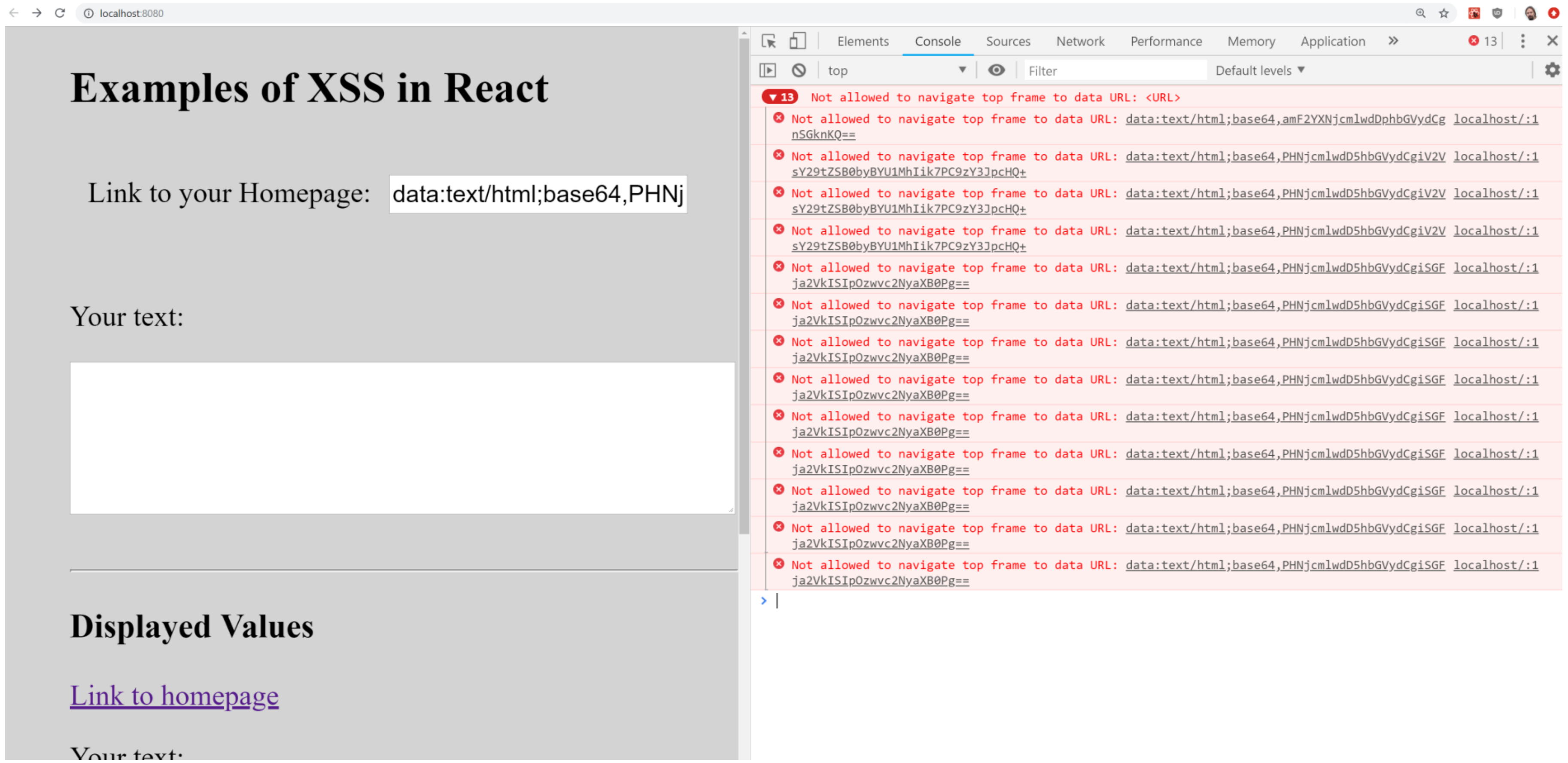

Not secure | data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0byBYU1Mhlik7PC9zY3JpcHQ+

This page says

Welcome to XSS!

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

- But “feature” removed from HTML links in browsers in 2017 in the “*top frame*”, due to security concerns…
- still… good example to see why you should not write your own sanitization functions… so many weird edge cases exist!!!
  - eg, have fun looking at https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet



Examples of XSS in React

Link to your Homepage: data:text/html;base64,PHNj

Your text:

Displayed Values

Link to homepage

Your text:

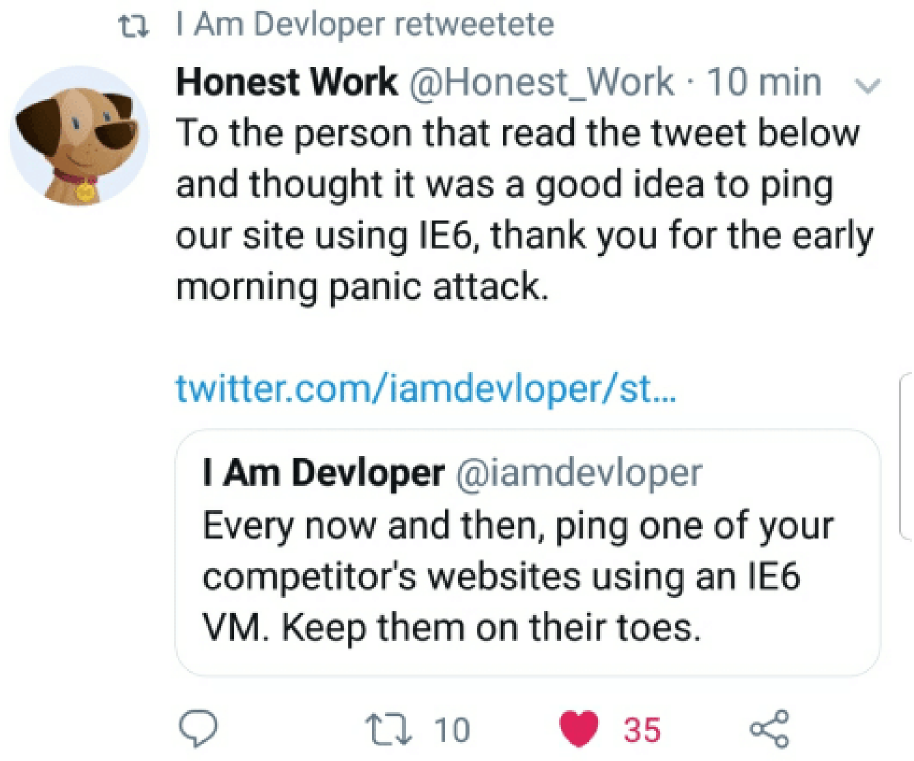
```
▼13 Not allowed to navigate top frame to data URL: <URL>
Not allowed to navigate top frame to data URL: data:text/html;base64,amF2YXNjcmlwdDphbGVydCg localhost/:1
nSGknKQ==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
```

User vs Developer

- *As a user*: **ALWAYS UPDATE TO LATEST BROWSER VERSION**
  - it will protect you from many known attacks

- *As a developer*: many of your clients will still use old browsers…
  - so you might still need to add extra layers of protection in your applications, even for attacks that would not be possible on recent browsers

- 2020: **Internet Explorer** still has a **1.7%** market share
    - 2.1% in Norway
    - In “theory” replaced by **Edge** in 2015…
- 2019: Edge was rebuilt in Chromium
- Legacy Edge in 2020
    - Global: 2.2%
    - Norway: 3.7%
- See https://gs.statcounter.com/


I Am Devloper retweetete

Honest Work @Honest_Work · 10 min
To the person that read the tweet below and thought it was a good idea to ping our site using IE6, thank you for the early morning panic attack.

twitter.com/iamdevloper/st…

I Am Devloper @iamdevloper
Every now and then, ping one of your competitor's websites using an IE6 VM. Keep them on their toes.

10 35