

# Web Application Security

Security, XSS, SQL Injection, CSRF, Parameter Tampering, CORS



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

[sli.do](https://sli.do)

**#csharp-web**

# Table of Contents

1. Common security problems
2. Cross-Site Scripting (XSS)
3. SQL Injection
4. Cross-Site Request Forgery (CSRF)
5. Parameter Tampering
6. CORS





# **Common Web App Security Problems**

XSS, SQL Injection, CSRF, Parameter Tampering

# Most Common Web Security Problems

- **SQL** Injection
- Cross-site Scripting (**XSS**)
- URL / HTTP manipulation attacks (**Parameter Tampering**)
- Cross-site Request Forgery (**CSRF**)
- **DoS, DDoS** and **Brute Force** attacks
- Too much **information** in Errors

```
Fatal error: Uncaught exception 'Exception' with message 'Lost connection to
MySQL server during query' in /home/www/bdz.bg/www/m/db/database.inc.php:44
Stack trace: #0 /home/www/bdz.bg/www/m/db/mysql_database.inc.php(31): Database-
>ThrowException('Lost connection...') #1 /home/www/bdz.bg/www/m/commit.php(26):
MySqlDatabase->Connect('213.222.56.138', 'new', 'mobile_guide', 'mobile$BDZ')
#2 {main} thrown in /home/www/bdz.bg/www/m/db/database.inc.php on line 44
```

<https://owasp.org/Top10>  
<https://www.exploit-db.com>

- Security flows in other software we use

- Semantic **URL/HTTP attacks** (URL / HTTP manipulation)
  - Always validate the data on the server-side
- **Man in the Middle** (Always use SSL)
- **Insufficient Access Control**
- Other types of **data injection** (Always sanitize data)
- **Phishing** and **Social Engineering** (Educate your users)
- **Security flows** in other software we use (Use latest versions)

- There is a wide range of known types of threats and attacks

Category	Threats / Attacks
<b>Input Validation</b>	Buffer overflow, cross-site scripting, SQL injection, canonicalization
<b>Parameter Tampering</b>	Query string manipulation, form field manipulation, cookie manipulation, HTTP header manipulation
<b>Session Management</b>	Session hijacking, session replay, man-in-the-middle
<b>Cryptography</b>	Poor key generation or key management, weak or custom encryption
<b>Sensitive Information</b>	Access sensitive code or data in storage, network eavesdropping, code/data tampering, <b>Admin password in exceptions</b>
<b>Exception Management</b>	Information disclosure, denial of service

- There is an even wider range of unknown threats and attacks...



# Cross Site Scripting (XSS)

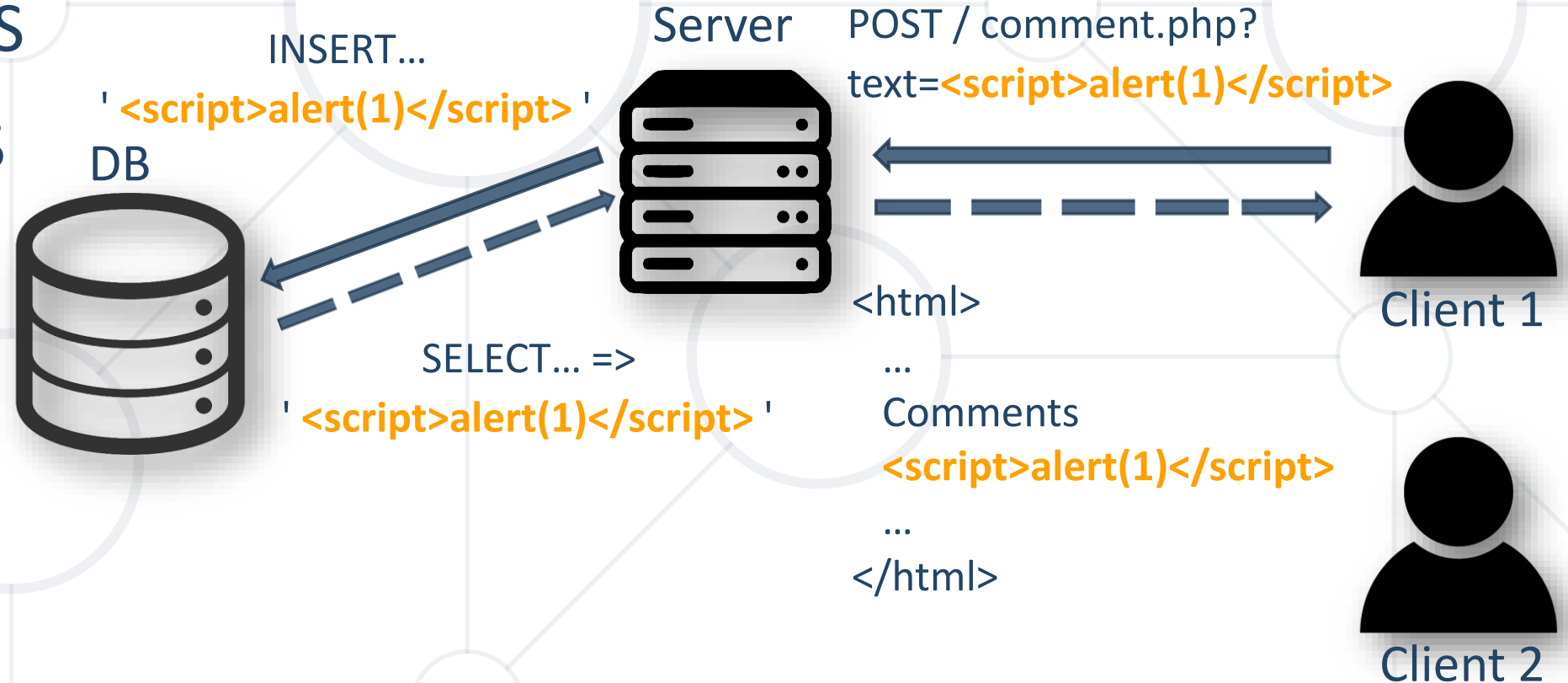
Injecting Unsafe HTML Code (with Scripts)



# What is Cross Site Scripting (XSS)?

- **XSS** attacks enable attackers to inject **client-side scripts** into **web pages** viewed by users

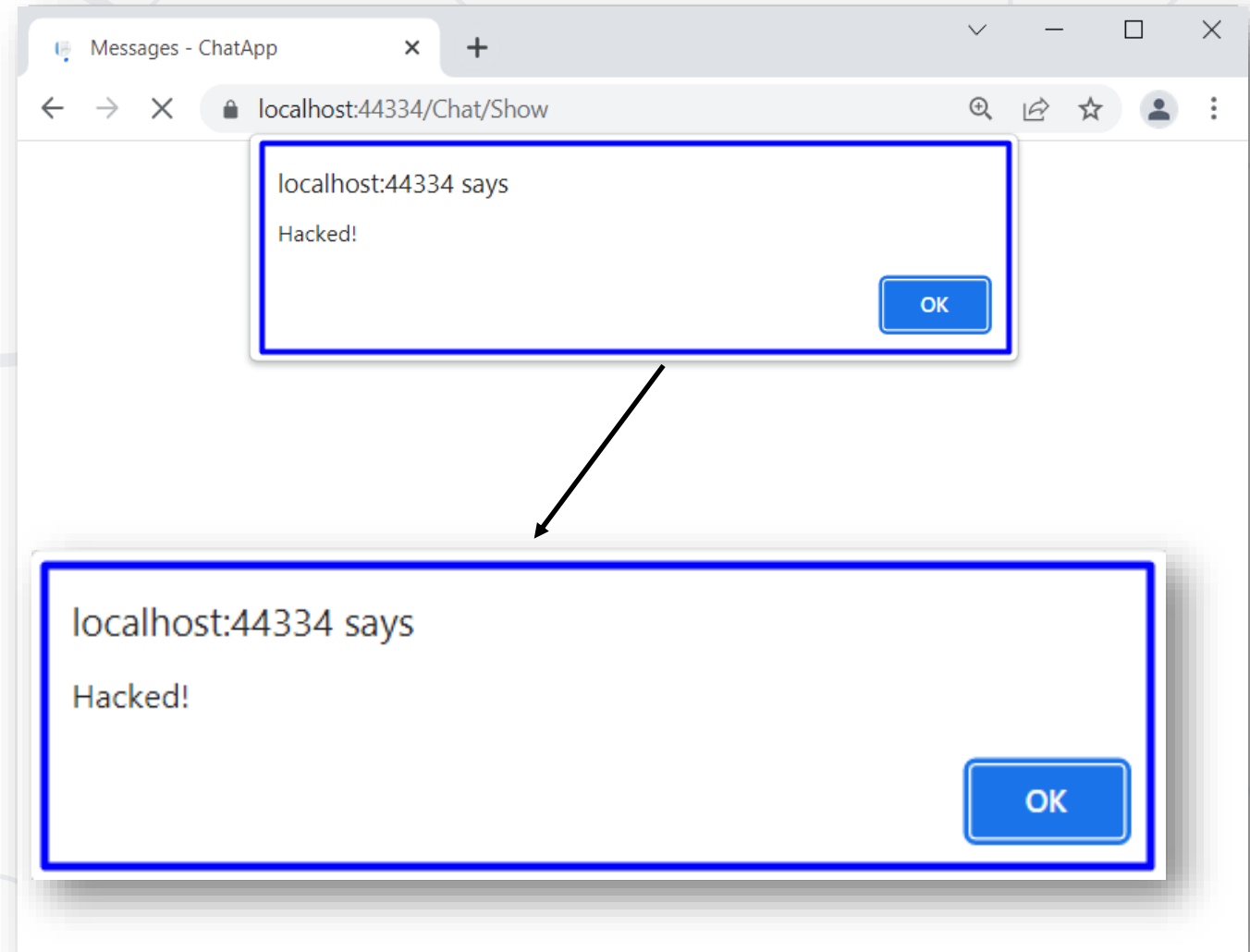
- Server XSS
- Client XSS



# Cross Site Scripting (XSS) – Demo

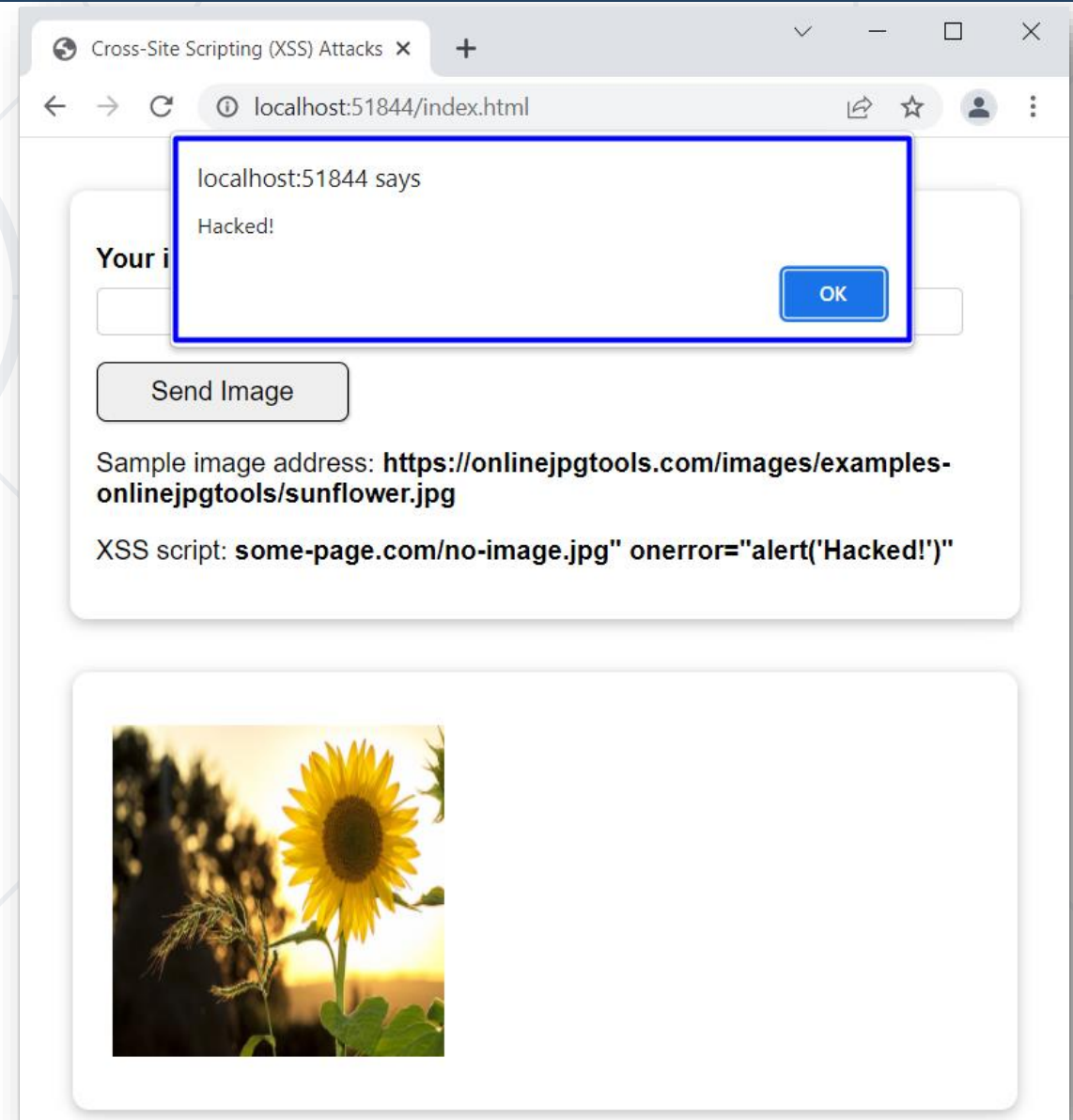
- We have a **vulnerable** ASP.NET Core chat app
  - User adds

```
"<script>  
  alert( 'Hacked!' )  
</script>"
```
  - Then, a JS popup appears
- Demo code: see the resources



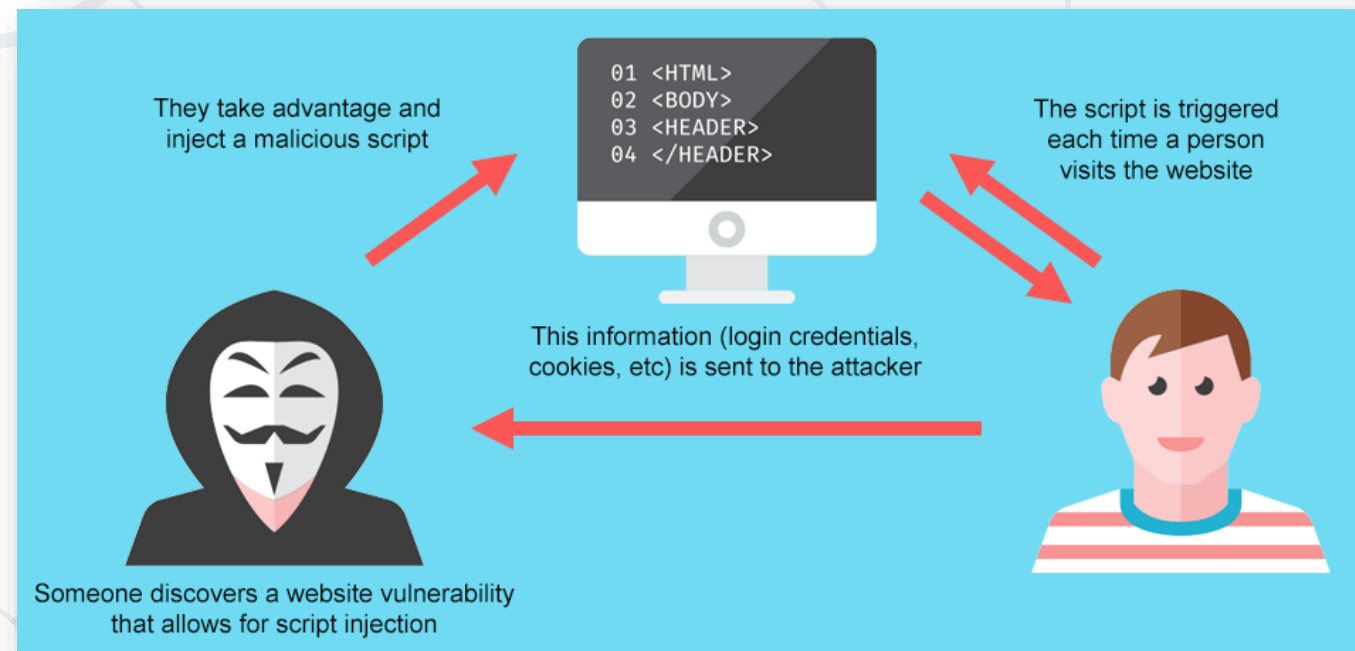
# Cross Site Scripting (XSS) with Image – Demo

- We have a vulnerable JS images app
  - User adds an invalid image URL and an **onclick event with a JS script**
  - Then, a JS popup appears
- Demo code: see the resources

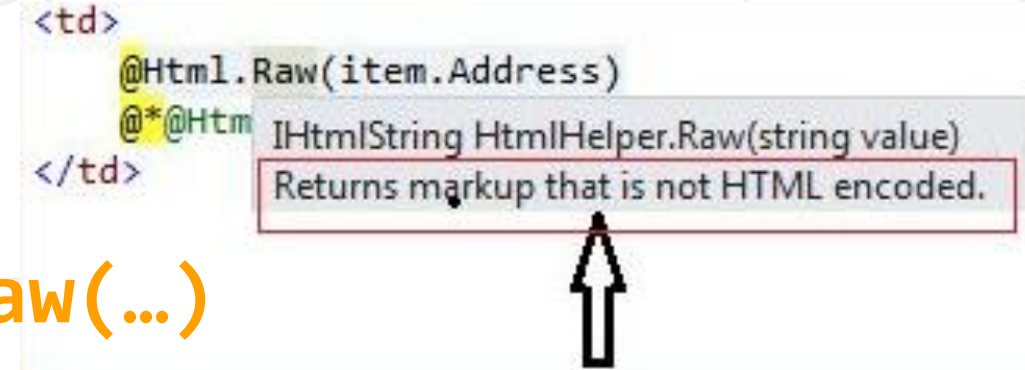


# Why is XSS a Big Security Problem?

- Attackers can
  - **Steal cookies, session storage**, local storage, etc.
  - **Impersonate you**, e. g. "create a new admin user"
  - Perform **actions** on behalf of the user
  - Gain access to the **user's sensitive data**
  - Etc.



- **The Razor view engine** secures you against **XSS** by default
  - If you decide to break it – **@Html.Raw(...)**
- There are several **rules** you must follow to be secured
  - Never put **untrusted data** into your HTML output
  - Before putting untrusted data somewhere, ensure it is **secured**
    - Encoded, parsed, validated, checked for malicious contents
  - Untrusted data can be inputted anywhere in the application
    - URLs, HTML Elements, HTML Attributes, JavaScript code etc.



- **ASP.NET Core** provides you with anything to secure your app
  - **Razor** automatically encodes all output sourced from variables

```
@{ var untrustedInput = "<\\"script\\">"; }  
@untrustedInput
```

&lt;t;&quot;script&quot;&gt;

- You can inject **Encoders** directly to your Views and use them

```
@using System.Text.Encodings.Web;  
@inject JavaScriptEncoder encoder;  
  
@{ var untrustedInput = "<\\"123\\">"; }  
  
<script> document.write("@encoder.Encode(untrustedInput)"); </script>  
  
<script> document.write("\u003C\u0022123\u0022\u003E"); </script>
```

- You can also use ASP.NET Core **Encoder Services**

- **HtmlEncoder**

<"123">



&#x3C;&#x22;123&#x22;&#x3E;

- **JavaScriptEncoder**

<"123">



u003C\u0022123\u0022\u003E

- **UrlEncoder**

<"123">



%3C%22123%22%3E

- Alternatively, you can use the static methods

- **WebUtility.HtmlEncode** and **WebUtility.HtmlDecode**
- **WebUtility.UrlEncode** and **WebUtility.UrlDecode**

- **HtmlSanitizer** is a .NET library for cleaning HTML fragments and documents from constructs that can lead to XSS attacks
  - <https://github.com/mganss/HtmlSanitizer>
- Install the HtmlSanitizer NuGet package, then

```
var sanitizer = new HtmlSanitizer();
var html = @"<script>alert('xss')</script><div onload=""alert('xss')""
+ @"style=""background-color: test"">Test<img src=""test.gif""
+ @"style=""background-image:
    url(javascript:alert('xss')); margin: 10px""></div>";
var sanitized = sanitizer.Sanitize(html, "http://www.example.com");
Assert.That(sanitized, Is.EqualTo(@"<div style=""background-color: test"">
+ @"Test<img style=""margin: 10px""
    src=""http://www.example.com/test.gif""></div>"));
```





# SQL Injection

Inject SQL Code in Unsafe Database Query

- The following SQL commands are executed

- Usual search (**no SQL injection**)

```
SELECT * FROM Messages WHERE MessageText LIKE '%JohnSnow%'
```

- SQL-injected search (matches **all records**)

```
SELECT * FROM Messages WHERE MessageText LIKE '%%%'
```

```
SELECT * FROM Messages WHERE MessageText LIKE '%' or 1=1 --%'
```

- SQL-injected **INSERT** command

```
SELECT * FROM Messages WHERE MessageText  
LIKE '%'; INSERT INTO Messages(MessageText, MessageDate) VALUES  
('Hacked!!!', '1.1.1980') --%'
```

- The following SQL commands are executed

- Usual search (**no SQL injection**)

```
SELECT * FROM Messages WHERE MessageText LIKE '%JohnSnow%'
```

- SQL-injected search (matches **all records**)

```
SELECT * FROM Messages WHERE MessageText LIKE '%%%'
```

```
SELECT * FROM Messages WHERE MessageText LIKE '%" or 1=1 --%'
```

- SQL-injected **INSERT** command

```
SELECT * FROM Messages WHERE MessageText  
LIKE '%'; INSERT INTO Messages(MessageText, MessageDate)  
VALUES ('Hacked!!!', '1.1.1980') --%'
```

- Original **SQL Query**

```
String sqlQuery = "SELECT * FROM user WHERE name = '" + username + "'  
AND pass='" + password + "'";
```

- Setting username to **Admin** & password to '**OR '1'='1**' produces

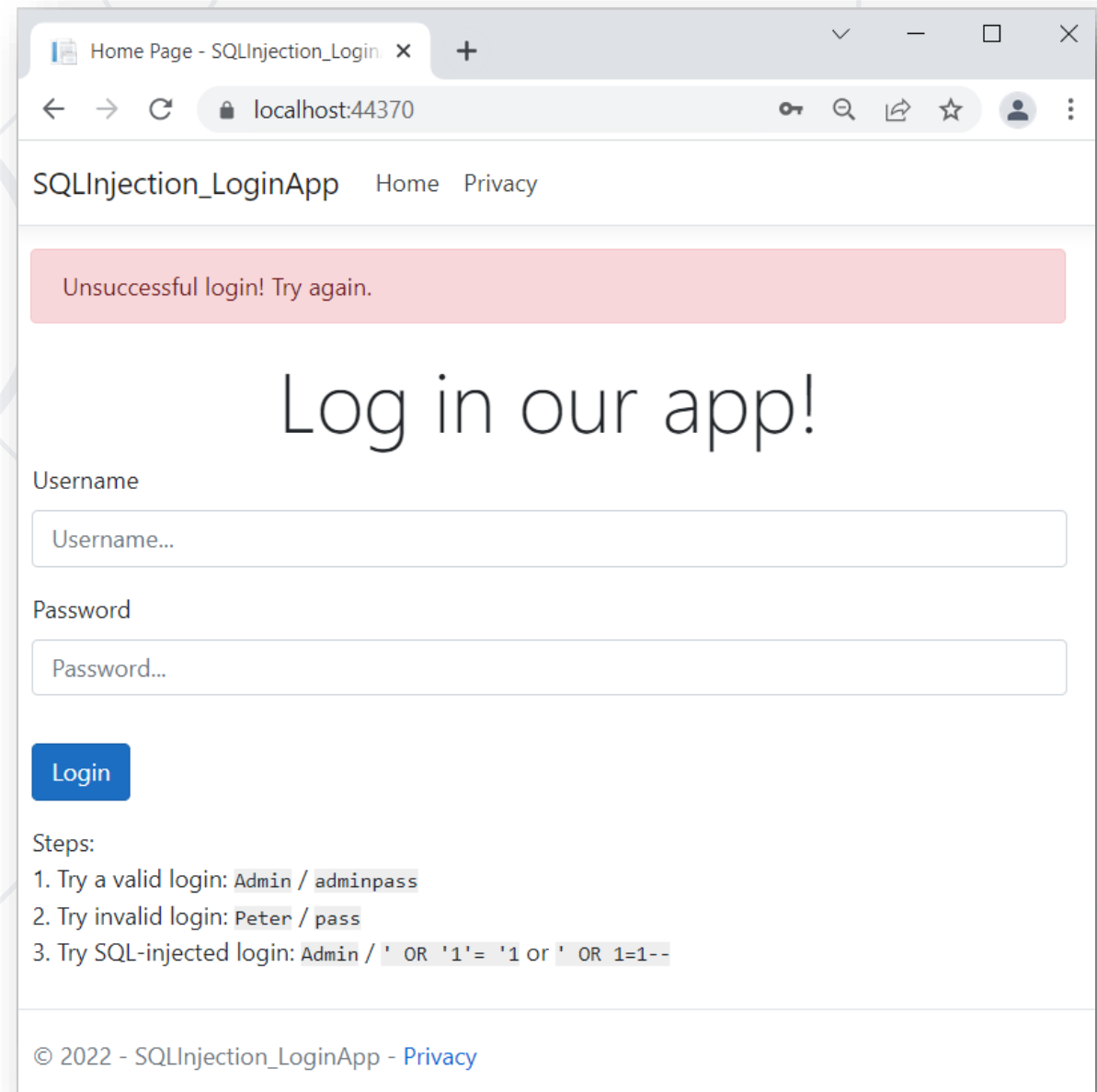
```
String sqlQuery = SELECT * FROM user WHERE name = 'Admin' AND  
pass=' ' OR '1'='1'
```

- The result

- The user with **username** – "**Admin**" will login **WITHOUT** password
- The **pass query** will turn into a **bool** expression which is **always true**

# SQL Injection – Demo (Normal Workflow)

- We have a simple app with an SQL-injectable login form
  - You can log in with "Admin" / "adminpass"
    - A success message appears
- Other credentials are invalid
  - An error message appears



The screenshot shows a web browser window with the title "Home Page - SQLInjection\_LoginApp". The address bar shows "localhost:44370". The page has a navigation bar with "SQLInjection\_LoginApp", "Home", and "Privacy". A red error message box at the top says "Unsuccessful login! Try again." Below this is a large heading "Log in our app!". There are two input fields: "Username" and "Password". Below the inputs is a blue "Login" button. At the bottom, there is a "Steps:" section with three instructions: "1. Try a valid login: Admin / adminpass", "2. Try invalid login: Peter / pass", and "3. Try SQL-injected login: Admin / ' OR '1' = '1 or ' OR 1=1--". The footer shows "© 2022 - SQLInjection\_LoginApp - Privacy".

Home Page - SQLInjection\_LoginApp

localhost:44370

SQLInjection\_LoginApp Home Privacy

Unsuccessful login! Try again.

## Log in our app!

Username

Password

Login

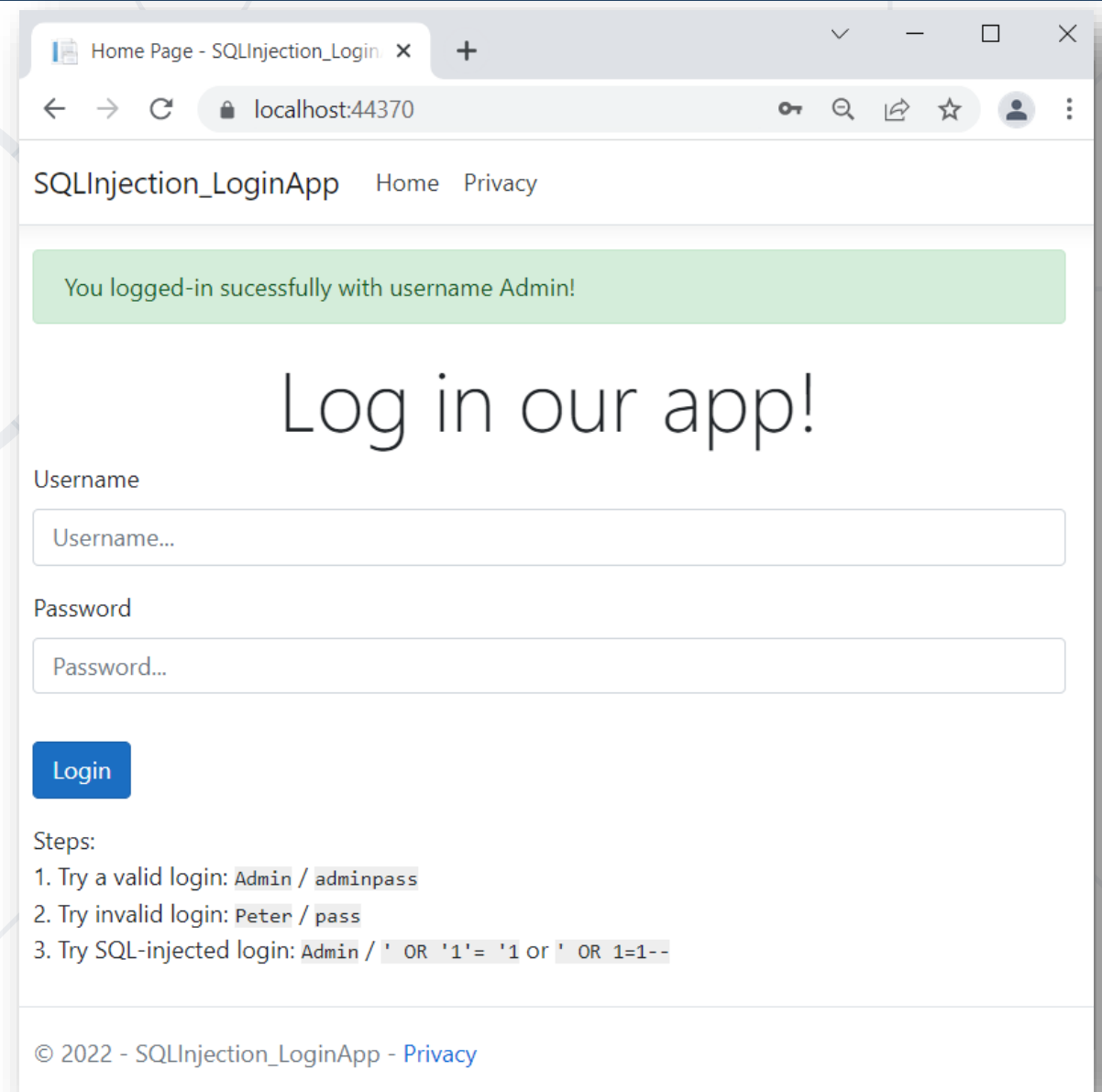
Steps:

1. Try a valid login: Admin / adminpass
2. Try invalid login: Peter / pass
3. Try SQL-injected login: Admin / ' OR '1' = '1 or ' OR 1=1--

© 2022 - SQLInjection\_LoginApp - Privacy

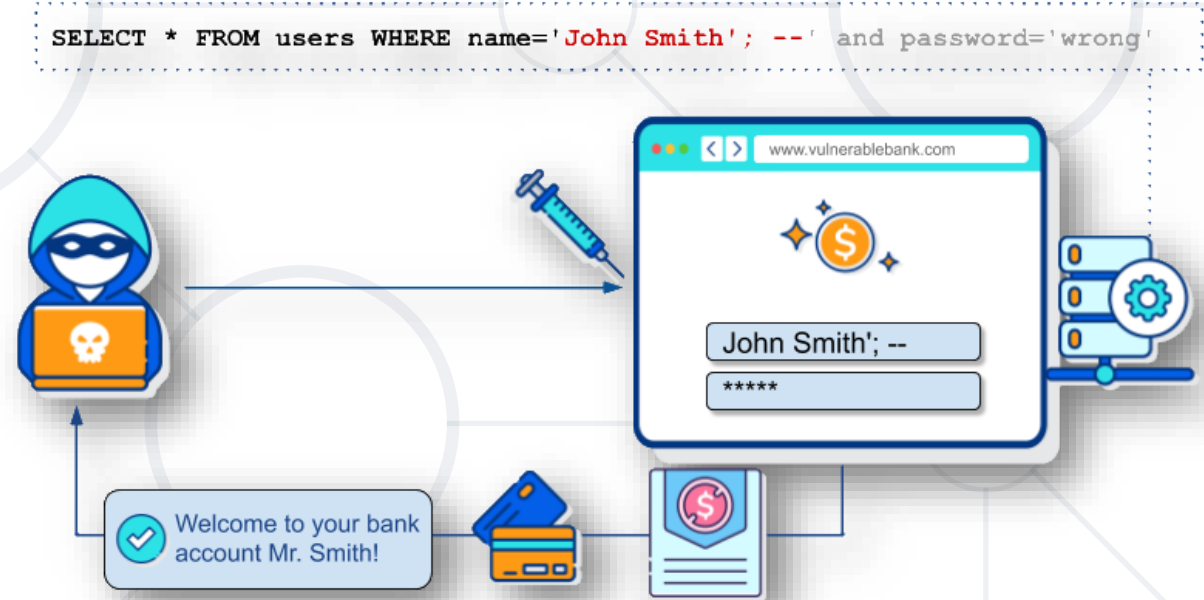
# SQL Injection – Demo

- Use **SQL Injection** to log in with the "Admin" username and **no password**
  - Write '**OR '1'='1**' in the password field
  - Now you are successfully logged-in
- Demo code: see the resources



# SQL Injection – How to Protect?

- Don't concatenate SQL with "+"
  - Use **parameterized SQL** queries or **stored procedures**
- Escape and sanitize all **user input**
- Never connect to a database with an **admin-level account**
- Don't store **secrets** in plain text
- **Exceptions** should reveal minimal information



- SQL-injection **vulnerable code**

```
string sqlQuery = "SELECT * FROM Users WHERE username = '"  
    + user.Username  
    + "' AND Password = '"  
    + user.Password + "'";  
  
var userExists = this.data.Users.FromSqlRaw(sqlQuery).Any();  
  
if(userExists) ...  
else ...
```

- The same code, rewritten correctly, with **parameterized query**

```
var userExists = this.data.Users  
    .Any(u => u.Username == user.Username && u.Password == user.Password);  
  
if(userExists) ...  
else ...
```



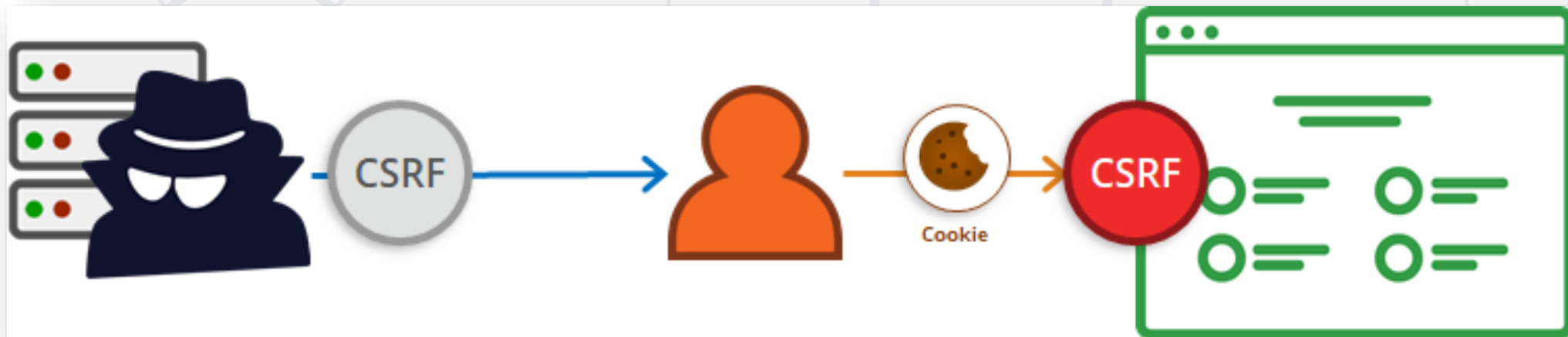


# Cross-Site Request Forgery

Submit a Form on Behalf of Unsuspecting User

# Cross-Site Request Forgery

- **Cross-Site Request Forgery (CSRF / XSRF)** is a web security attack over the HTTP protocol
  - Allows **executing unauthorized commands** on behalf of some user
    - By using his cookies stored in the browser
  - The user has valid permissions to execute the requested command
  - The attacker uses these permissions maliciously, unbeknownst to the user



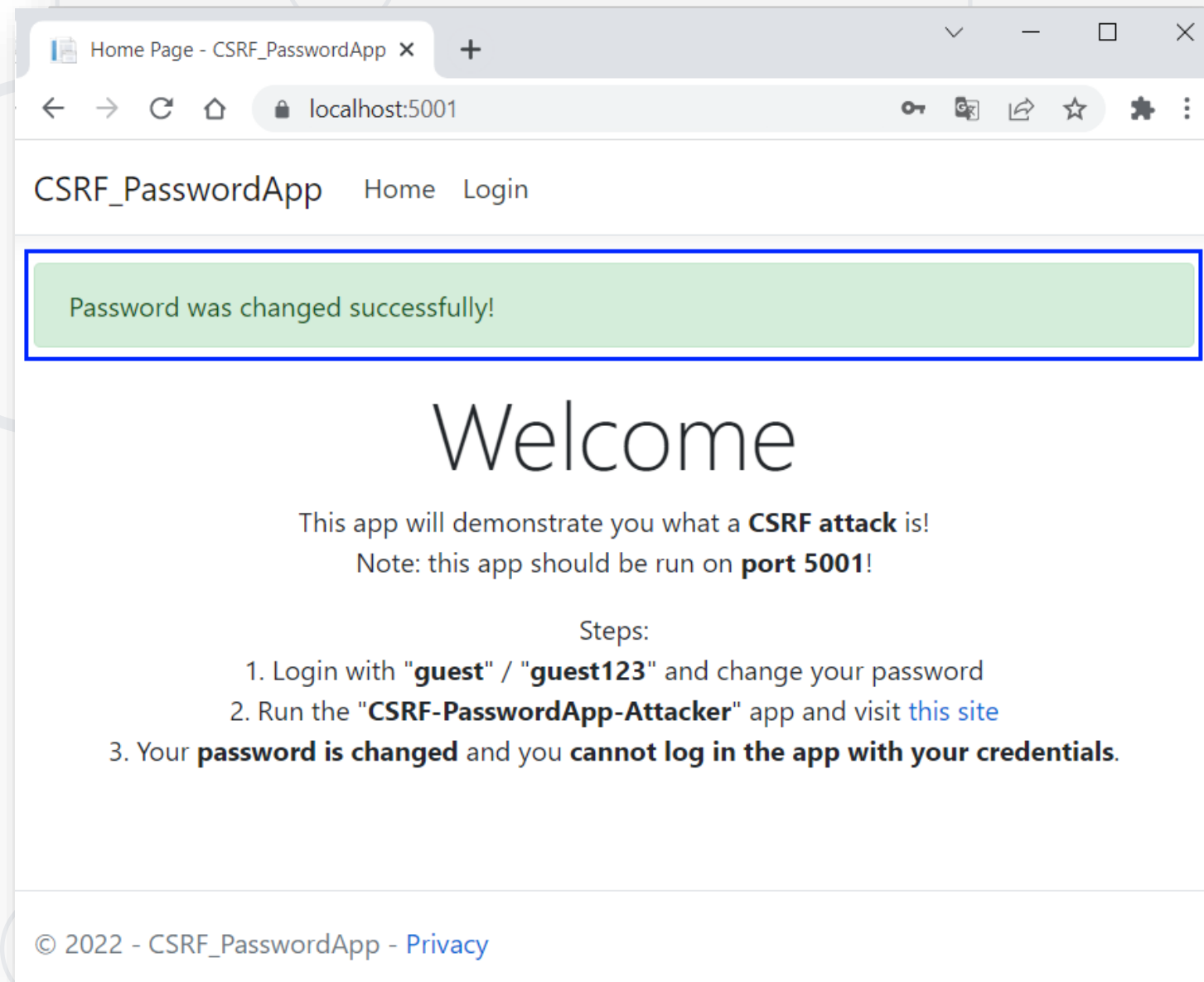
- What **Cross-Site Request Forgery** actually is

```
<!-- SOME MULTI-COLOR USELESS CLICKBAIT CONTENT -->  
  
<form action="http://good-banking-site.com/api/account" method="post">  
  <input type="hidden" name="Transaction" value="withdraw">  
  <input type="hidden" name="Amount" value="1000000">  
  <input type="submit" value="Click to collect your prize!">  
</form>
```

- The user can even **misclick** the button accidentally
  - This will still trigger the attack
  - Security against such attacks is necessary
    - It protects both **your app** and **your clients**

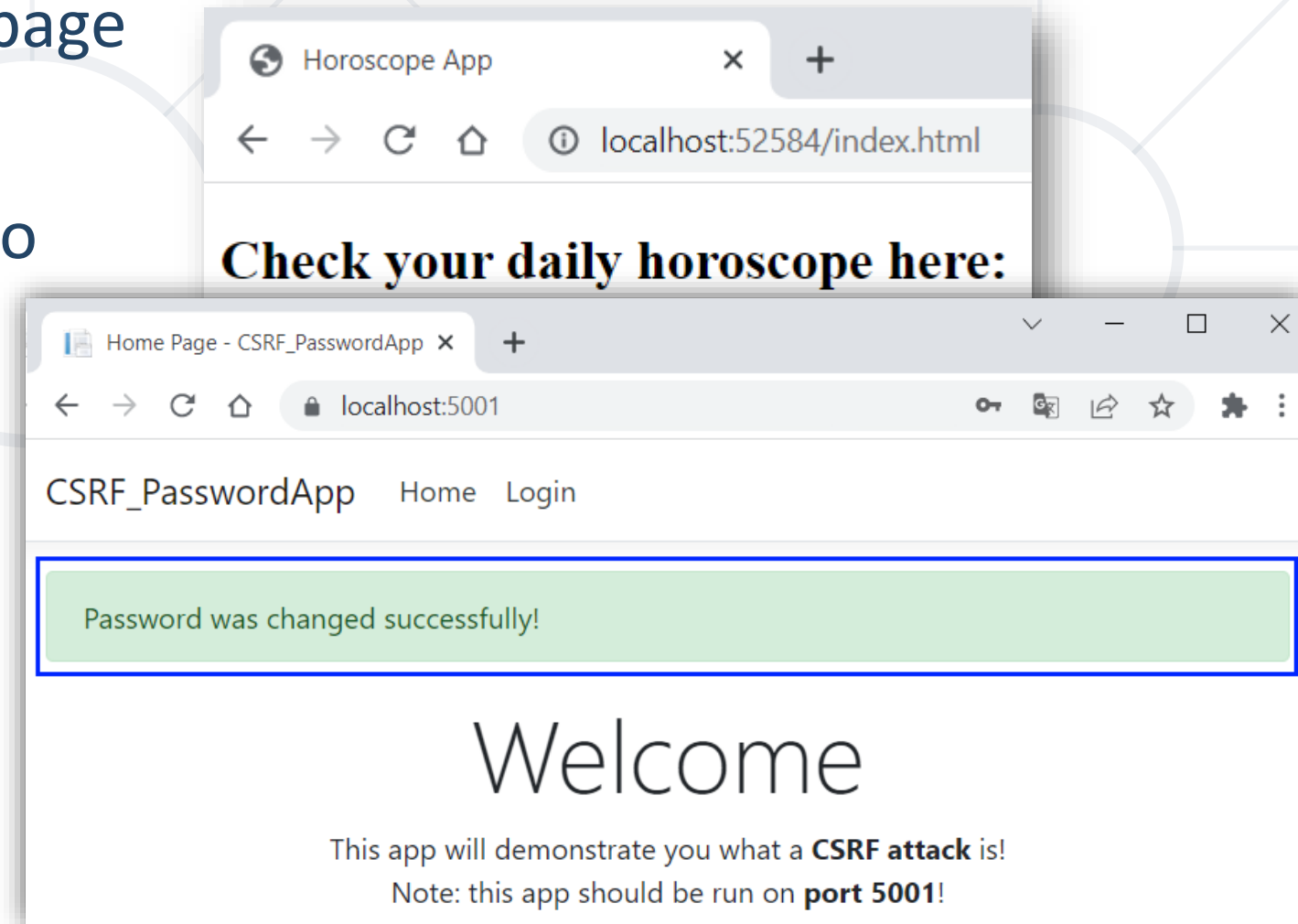
# CSRF – Demo (Normal Workflow)

- We have a very simple app, where users can **login and change their password**
  - Without protection for CSRF
  - Access the app on **port 5001**
- Login with default user "**guest**" / "**guest123**"
- Change the password
- You will see a success message on the "Home" page
- Now you can log in with your new password



# CSRF – Demo (Attack Workflow)

- Visit the **link** on the "Home" page
  - It accesses a **malicious site**
- Click on the [Click] button to **trigger the attack**
- The malicious app **changed your password** through **CSRF**
  - Now you cannot log in with your credentials
  - Your password has changed to "**hacked!**"
- Demo code: see the resources



- The **<form>** tag helper in ASP.NET Core automatically adds a special hidden field to the form
  - It has a random value called **anti-forgery token**
- Then you should **require this token** to be send

- For a specific action

```
[AutoValidateAntiforgeryToken]  
public IActionResult SendMoney(...) { ... }
```

- For all actions in given controller

```
[AutoValidateAntiforgeryToken]  
public class ManageController : Controller
```

- Globally for the whole application

```
services.AddMvc(options =>  
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

# The Anti-Forgery Token in ASP.NET MVC

## Log in

Use a local account to log in.

Email

Password

☐ Remember me?

Log in

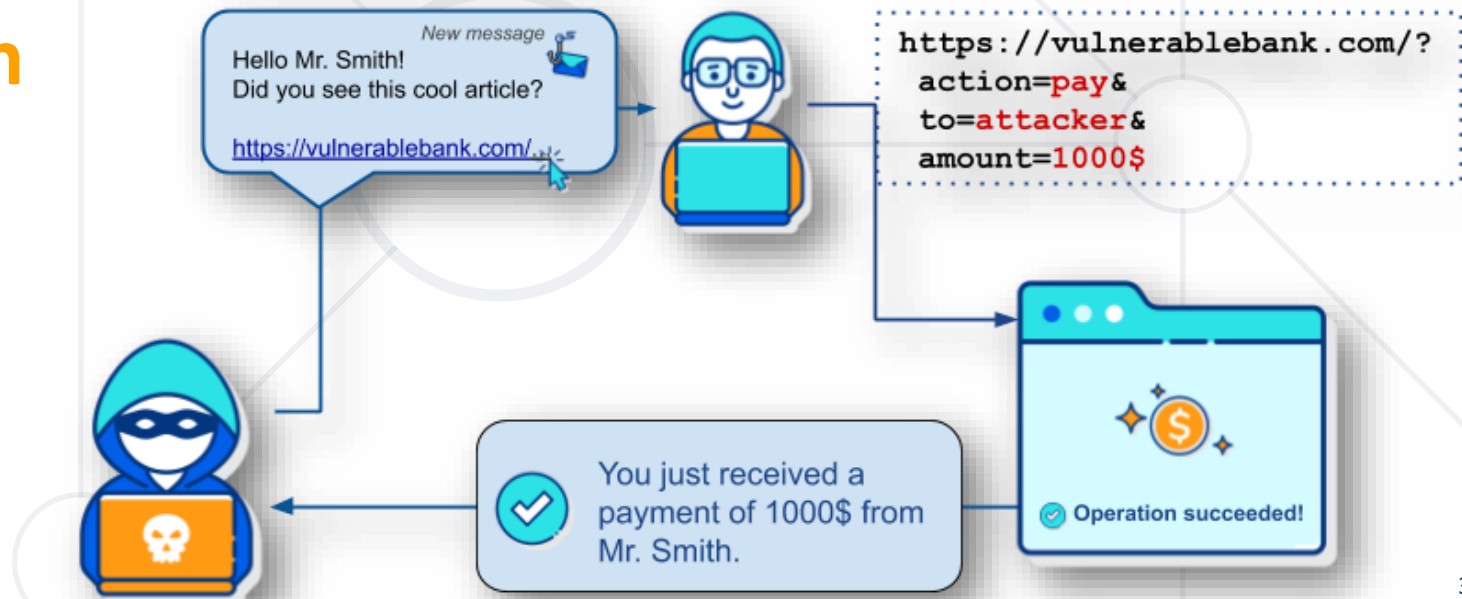
```
<form id="account" method="post" novalidate="novalidate">
  <h4>Use a local account to log in.</h4>
  <hr>
  ▶ <div class="form-group">...</div>
  ▶ <div class="form-group">...</div>
  ▶ <div class="form-group">...</div>
  ▶ <div class="form-group">...</div>
  ▶ <div class="form-group">...</div>
  <input name="__RequestVerificationToken" type="hidden" value=
    "CfDJ8Fksy1R6YXZMqcQ_RDpbjf_9rfrKnZDzbJEUv9iu1gGQE175WG3KLrozo
    BNQiQZgUMaJ6VC7RBC-TkVBim_TXEvWgm72AF-sYJhd2_euEmTYSkNSPqRsr4e
    21BXkLPOrmbW1Fh4hBcUEiR19gP_5JYY"> == $0
  <input name="Input.RememberMe" type="hidden" value="false">
</form>
```

# CSRF: How to Protect?

- Use **anti-forgery token**
- Include **additional authentication** for sensitive actions
- Use the **SameSite** flag in cookies

```
Set-Cookie: CookieName=CookieValue; SameSite=Strict;
```

- Following a **RESTful design**
- Enabling **CORS protection**





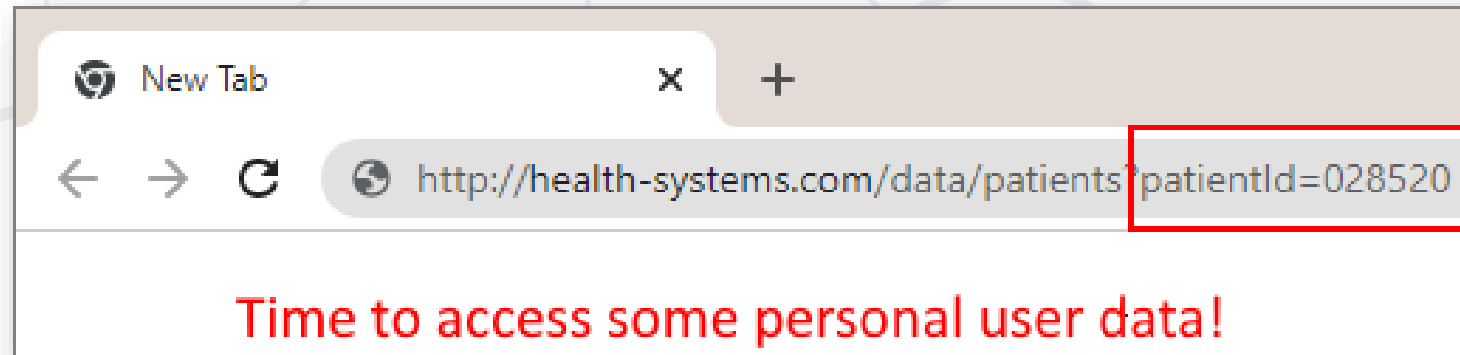


# Parameter Tampering

Changing Input Parameters at the Client Side

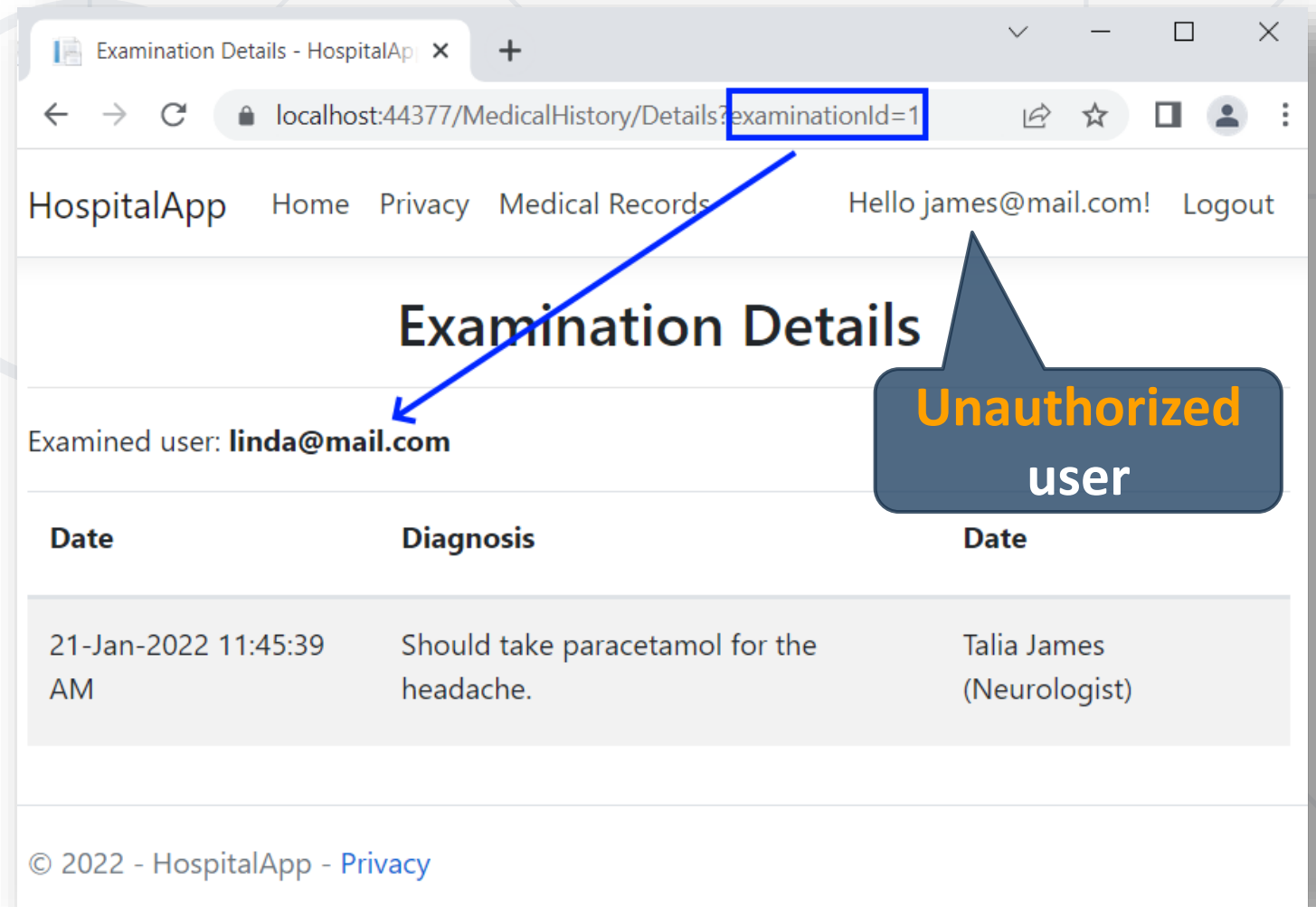
# Parameter Tampering

- **Parameter Tampering** is the manipulation of **parameters** exchanged between **client** and **server**
  - Altered query strings, request bodies, cookies
  - Skipped data validations, injected additional parameters



# Parameter Tampering – Demo

- We have a simple app, which displays data by ID
  - Without checking the permissions
- A hacker can **change the examination ID** in the URL
  - And **access other users' data**
- Demo code: see the resources

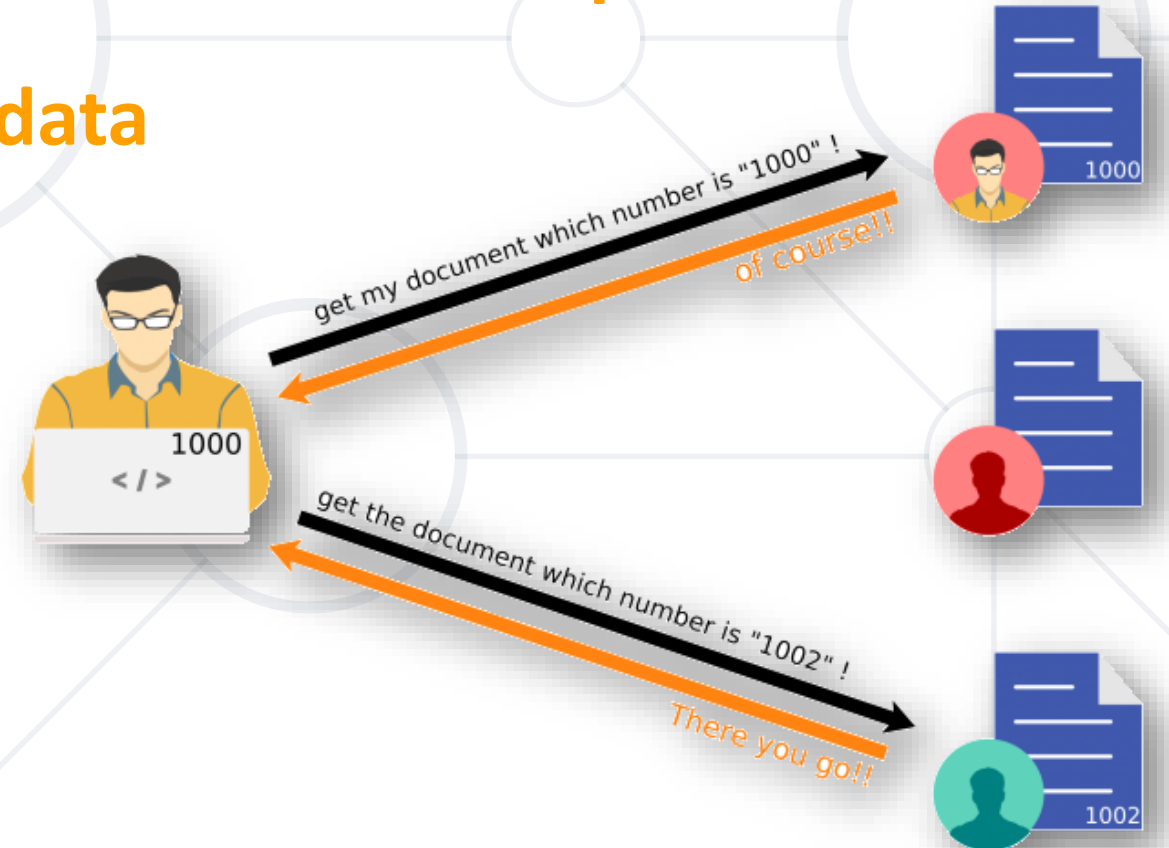


The screenshot shows a web browser window titled "Examination Details - HospitalApp". The address bar displays the URL `localhost:44377/MedicalHistory/Details?examinationId=1`, with `examinationId=1` highlighted by a blue box. A blue arrow points from this box to the "Examination Details" heading on the page. The page header includes "HospitalApp", navigation links "Home", "Privacy", and "Medical Records", and a user greeting "Hello james@mail.com!" with a "Logout" link. The main content area shows "Examination Details" with the text "Examined user: linda@mail.com". Below this is a table with three columns: "Date", "Diagnosis", and "Date". The table contains one row of data: "21-Jan-2022 11:45:39 AM", "Should take paracetamol for the headache.", and "Talía James (Neurologist)". A dark blue callout bubble with the text "Unauthorized user" is positioned to the right of the table. The footer of the page reads "© 2022 - HospitalApp - Privacy".

Date	Diagnosis	Date
21-Jan-2022 11:45:39 AM	Should take paracetamol for the headache.	Talía James (Neurologist)

# Parameter Tampering: How to Protect?

- Check the **input parameters** before accessing the database
- The **forms** on the site should have some **built-in protection**
- Using regex to limit or **validate data**
- Avoid unwanted or hidden data
- **Encrypt** the session cookies



- Code, **vulnerable to parameter tampering**

```
var examination = this.data.Examinations.Include(ex => ex.Patient)
    .FirstOrDefault(ex => ex.Id == examinationId);

if(examination == null) return RedirectToAction("Summary");

var currentLoggedUserId = this.User
    .FindFirstValue(ClaimTypes.NameIdentifier);

if (examination.PatientId != currentLoggedUserId) return Unauthorized();

var model = new ExaminationRecord() { ... };
return View(model);
```

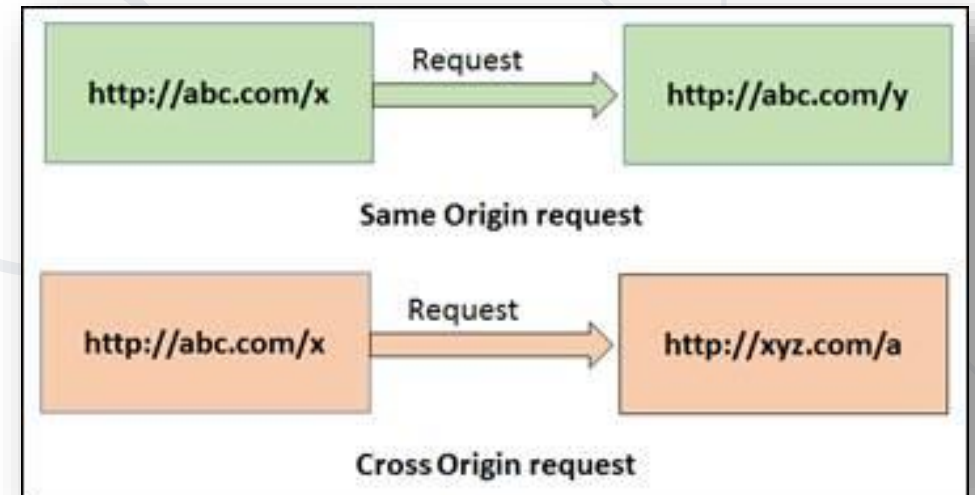
- Add **explicit checks** to secure the code



# Cross Origin Resource Sharing

CORS

- Browser security prevents a web page from making requests to a **domain**, different from the one that served the web page (its **origin**)
  - This restriction is called **Same-Origin Policy (SOP)**
  - This policy also prevents malicious sites from reading data from your site
- Sometimes you might want to allow other sites to bypass this restriction
  - **Cross-origin requests** to your app may become necessary, at some point
  - That's where **Cross Origin Resource Sharing (CORS)** comes to the rescue



- **CORS** is a **W3C** standard that allows a server to "relax" the **SOP**
  - Using **CORS**, a server can **explicitly** allow some cross-origin requests
  - That doesn't mean all cross-origin requests will be allowed
- Two URLs have the **same origin** if they have
  - Identical **Schemes**, **Hosts** and **Ports** (RFC 6454)

Same-origin URLs

`https://example.com/foo.html`

`https://example.com/bar.html`

`https://example.net`

Different **domain**

`https://www.example.com/foo.html`

Different **subdomain**

`http://example.com/foo.html`

Different **scheme**

`https://example.com:9000/foo.html`

Different **port**

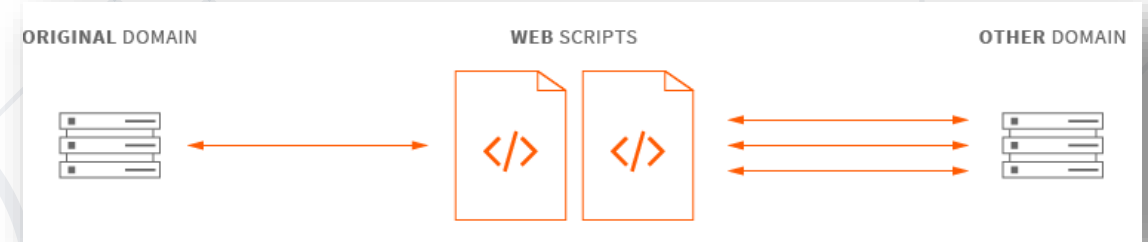


# CORS Example



# CORS in ASP.NET Core

- **CORS**, in ASP.NET Core, is setup
  - **Globally**, via a **middleware**
  - Per **Action** or per **Controller** via an **Attribute**



```
builder.Services.AddCors();
```

```
app.UseCors(builder =>  
    builder.WithOrigins("http://example.com"));
```

```
[HttpGet]  
[EnableCors("AllowSpecificOrigin")]  
public ContentResult GetTest()  
{  
    return Content("test");  
}
```

```
[HttpGet]  
[DisableCors]  
public string Version()  
{  
    return "1.0.0";  
}
```

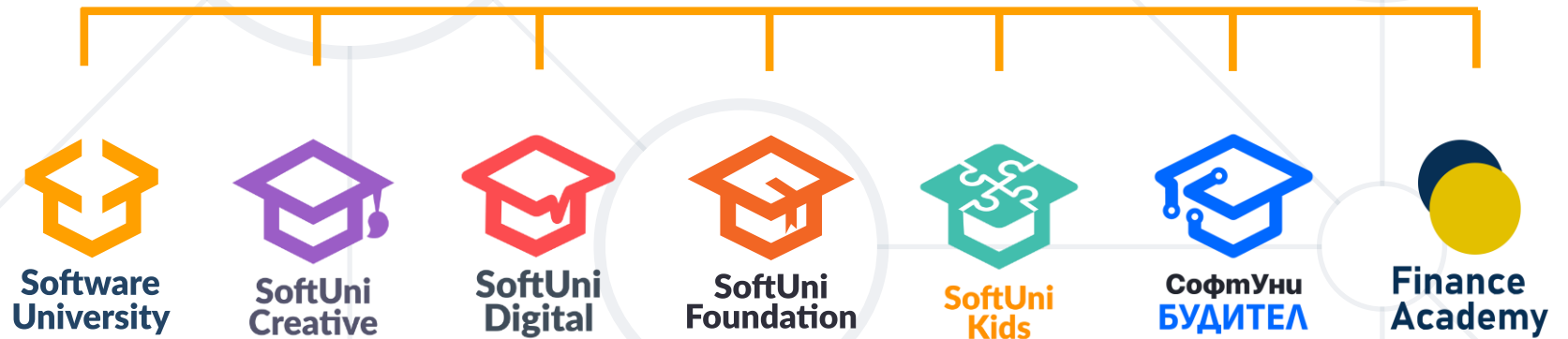
- Common **security problems**
- **Cross-Site Scripting** – attackers inject **malicious scripts**
- **SQL Injection** – attackers interfere with **app queries** to the db
- **Cross-Site Request Forgery** – attackers force users to **execute unwanted actions** on web apps that they are logged-in
- **Parameter Tampering** – attackers manipulate **parameters**, exchanged between client and server
- CORS allows you to bypass the browser's **same-origin policy**



# SoftUni Diamond Partners



# Questions?



- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)
- Software University Foundation
  - [softuni.foundation](http://softuni.foundation)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

