# SameSite: The CSRF Killer

In this lesson, we'll look at the SameSite flag.

**We'll cover the following**

## Introduction

Last but not least, let’s look at the `SameSite` flag, one of the latest entries in the cookie world.

Introduced by Google Chrome v51, this flag effectively eliminates *Cross-Site Request Forgery* (CSRF) from the web, `SameSite` is a simple yet groundbreaking innovation as previous solutions to CSRF attacks were either incomplete or too much of a burden to site owners.

In order to understand `SameSite`, we first need to have a look at the vulnerability it neutralizes. A CSRF is an unwanted request made by site A to site B while the user is authenticated on site B.

Sounds complicated? Let me rephrase, suppose that you are logged in on your banking website, which has a mechanism to transfer money based on an HTML `<form>` and a few additional parameters like destination account and amount. When the website receives a `POST` request with those parameters and your session cookie, it will process the transfer. Now, suppose a malicious third party website sets up an HTML form as such.

```
<form action="https://bank.com/transfer" method="POST">
<input type="hidden" name="destination" value="attacker@email.com" />
<input type="hidden" name="amount" value="1000" />
<input type="submit" value="CLICK HERE TO WIN A HUMMER" />
```

```
</form>
```

See where this is getting? If you click on the submit button, cleverly disguised as an attractive prize, $1000 is going to be transferred from your account. This is a cross-site request forgery, nothing more, nothing less.

Traditionally, there have been two ways to get rid of CSRF:

## Origin and Referrer headers

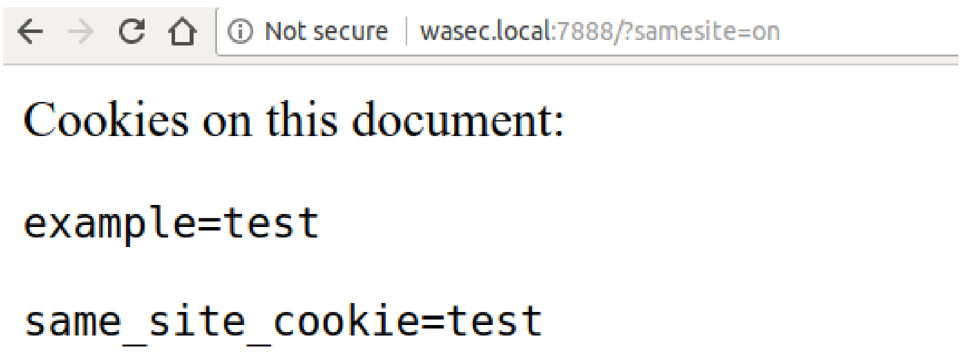
The server could verify that these headers come from trusted sources (ie. `https://bank.com`). The downside to this approach is that, as we've seen in previous chapters, neither the `Origin` nor `Referrer` are very reliable and could be turned off by the client in order to protect the user's privacy.

## CSRF tokens

The server could include a signed token in the form and verify its validity once the form is submitted. This is a generally solid approach and it's been the recommended best practice for years. The drawback of CSRF tokens is that they're a technical burden for the backend, as you'd have to integrate token generation and validation in your web application: this might not seem like a complicated task, but a simpler solution would be more than welcome.

`SameSite` cookies aim to supersede the solutions mentioned above once and for all. When you tag a cookie with this flag, you tell the browser not to include the cookie in requests that were generated by different origins. When the browser initiates a request to your server and a cookie is tagged as `SameSite`, the browser will first check whether the origin of the request is the same origin that issued the cookie. If it's not, the browser will not include the cookie in the request.

We can have a practical look at `SameSite` with the example at github.com/odino/wasec/tree/master/cookies. When you browse to wasec.local:7888/?samesite=on the server will set a `SameSite` cookie and a regular one.



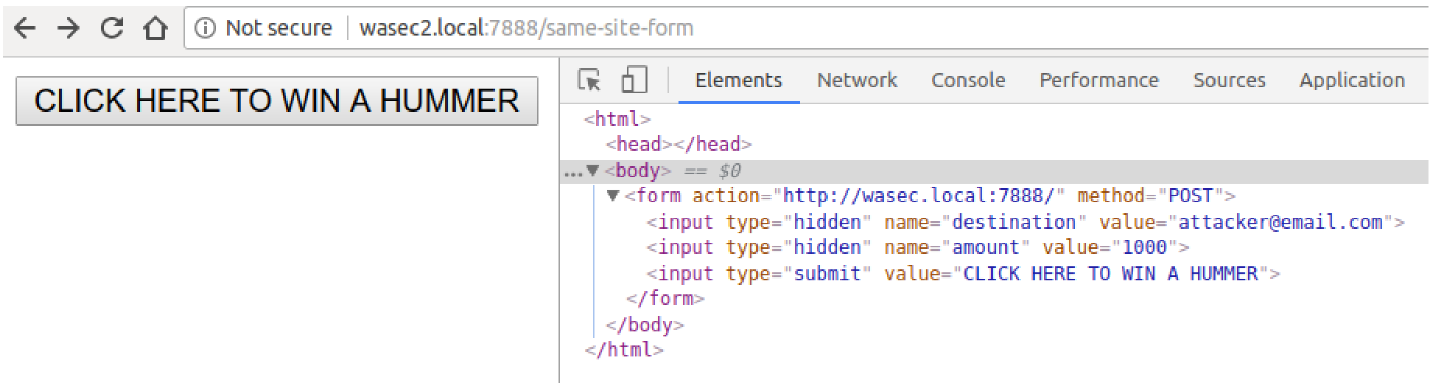
Cookies on this document:

example=test

same_site_cookie=test

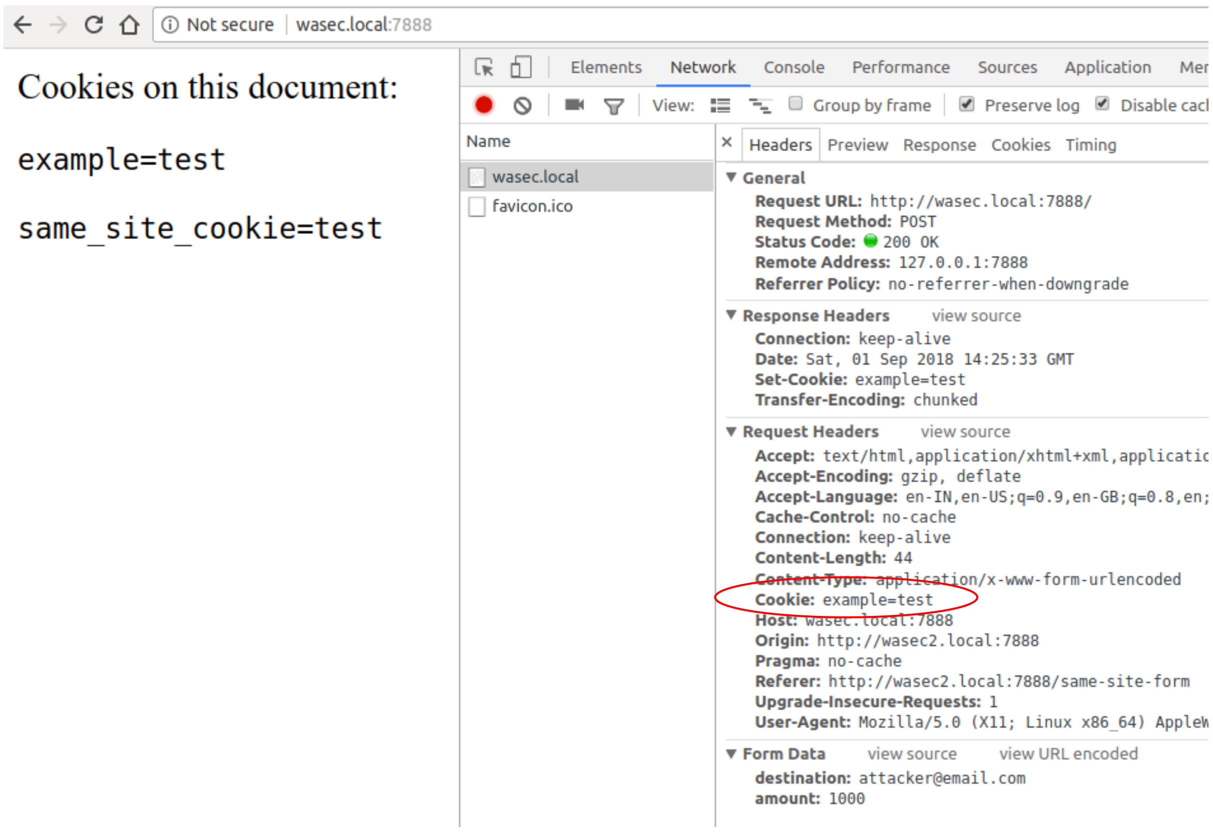
Server sets a "SameSiite" cookie and a "regular" one

If we then visit wasec2.local:7888/same-site-form we will see an example HTML form that will trigger a cross-site request.



```
<html>
  <head></head>
…▼<body> == $0
  ▼<form action="http://wasec.local:7888/" method="POST">
      <input type="hidden" name="destination" value="attacker@email.com">
      <input type="hidden" name="amount" value="1000">
      <input type="submit" value="CLICK HERE TO WIN A HUMMER">
    </form>
  </body>
</html>
```

If we click on the submit button of the form, we will then be able to understand the true power of this flag, the form will redirect us to wasec.local:7888, but there is no trace of the `SameSite` cookie in the request made by the browser.



Cookies on this document:

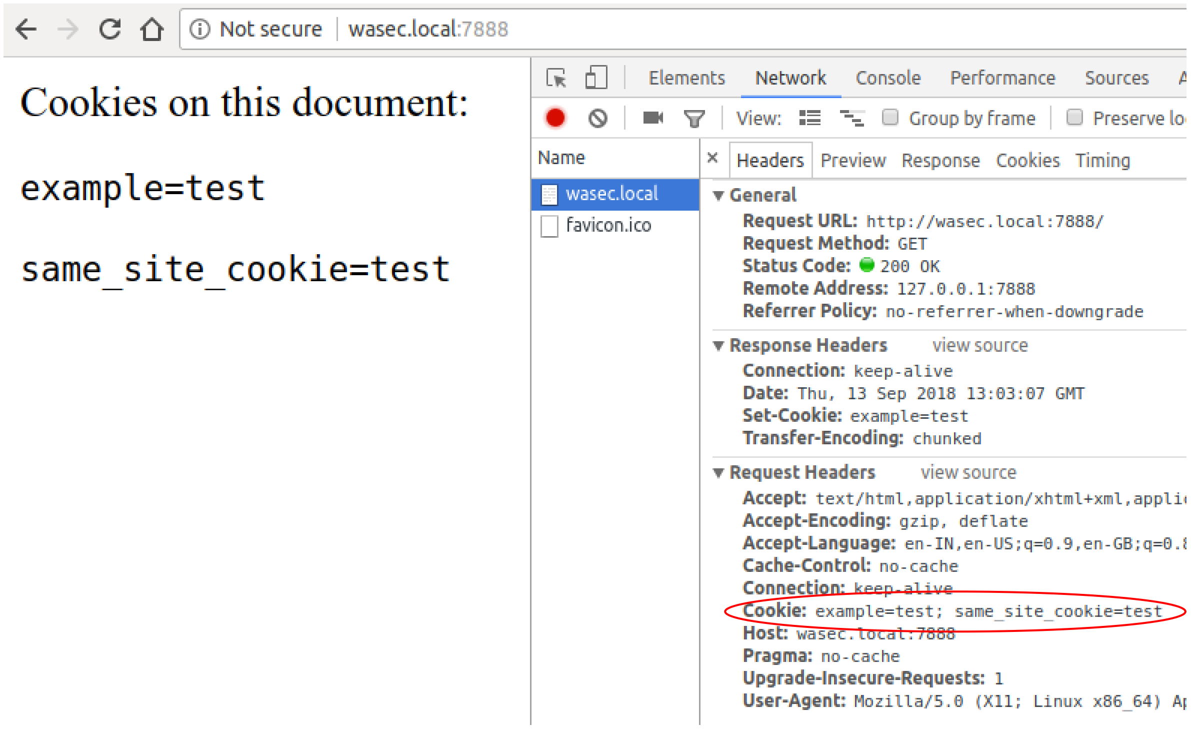
example=test

same_site_cookie=test

```
▼ General
    Request URL: http://wasec.local:7888/
    Request Method: POST
    Status Code: 200 OK
    Remote Address: 127.0.0.1:7888
    Referrer Policy: no-referrer-when-downgrade
▼ Response Headers      view source
    Connection: keep-alive
    Date: Sat, 01 Sep 2018 14:25:33 GMT
    Set-Cookie: example=test
    Transfer-Encoding: chunked
▼ Request Headers      view source
    Accept: text/html,application/xhtml+xml,applicatic
    Accept-Encoding: gzip, deflate
    Accept-Language: en-IN,en-US;q=0.9,en-GB;q=0.8,en;
    Cache-Control: no-cache
    Connection: keep-alive
    Content-Length: 44
    Content-Type: application/x-www-form-urlencoded
    Cookie: example=test
    Host: wasec.local:7888
    Origin: http://wasec2.local:7888
    Pragma: no-cache
    Referer: http://wasec2.local:7888/same-site-form
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleW
▼ Form Data      view source      view URL encoded
    destination: attacker@email.com
    amount: 1000
```

SameSite cookie not requested

Don’t get confused by seeing `same_site_cookie=test` on your screen, the cookie is made available by the browser, but it wasn’t sent in the request itself. We can verify this by simply typing `http://wasec.local:7888/` in the address bar:



Cookies on this document:

example=test

same_site_cookie=test

```
▼ General
    Request URL: http://wasec.local:7888/
    Request Method: GET
    Status Code: 200 OK
    Remote Address: 127.0.0.1:7888
    Referrer Policy: no-referrer-when-downgrade
▼ Response Headers      view source
    Connection: keep-alive
    Date: Thu, 13 Sep 2018 13:03:07 GMT
    Set-Cookie: example=test
    Transfer-Encoding: chunked
▼ Request Headers      view source
    Accept: text/html,application/xhtml+xml,appli
    Accept-Encoding: gzip, deflate
    Accept-Language: en-IN,en-US;q=0.9,en-GB;q=0.
    Cache-Control: no-cache
    Connection: keep-alive
    Cookie: example=test; same_site_cookie=test
    Host: wasec.local:7888
    Pragma: no-cache
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) Ap
```

Cookie requested

Since the originator of the request is safe (no origin, `GET` method) the browser

Since the originator of the request is safe (no origin, `GET` method) the browser sends the `SameSite` cookie with the request.

This ingenious flag has two main variants, `Lax` and `Strict`. Our example uses the former variant, as it allows top-level navigation to a website to include the cookie; when you tag a cookie as `SameSite=Strict` instead, the browser will not send the cookie across any cross-origin request, including top-level navigation: this means that if you click a link to a website that uses `strict` cookies you won't be logged in at all – an extremely high amount of protection that, on the other hand, might surprise users. The `Lax` mode allows these cookies to be sent across requests using safe methods (such as `GET`), creating a very useful mix between security and user experience.

The last variant for this flag, `None`, can be used to opt-out of this feature altogether. You might think that by not specifying the `SameSite` policy for a cookie, browsers would treat it the same way they did for years while, in reality, vendors are preparing to step up their security game. Chrome 80, set to be released in Q1 2020, is going to apply a default `SameSite=Lax` attribute if a cookie doesn't have a value set for this flag. Firefox developers have already stated they'd like to follow suit, so using `SameSite=None` will be the only way to ask the browser to ignore its default `SameSite` policy. It's worth noting that, in order to push for the adoption of stricter security policies, browsers will reject cookies opting out of SameSite unless they are declared `Secure`. To quote Scott Helme, "CSRF is (really) dead"

### Cookie flags are important

Let's recap what we've learned about cookie flags as they are crucial when you're storing or allowing access to sensitive data through them, which is a very standard practice:

- Marking cookies as `Secure` will make sure that they won't be sent across unencrypted requests, rendering man-in-the-middle attacks fairly useless.
- With the `HttpOnly` flag we tell the browser not to share the cookie with the client (e.g., allowing JavaScript access to the cookie), limiting the blast radius of an XSS attack.
- Tagging the cookie as `SameSite=Lax|Strict` will prevent the browser from sending it in cross-origin requests, rendering any kind of CSRF attack ineffective. It's important to note that there are still low-risk CSRF

> vulnerabilities that your application can be targeted with, like login
>
> CSRF. As I already mentioned, these vulnerabilities have a limited impact and risk associated.

---

In the next lesson, we’ll look at alternatives to cookies.