# Web security: hardening HTTP cookies

14 September 2018

This post is part of the "**WASEC: Web Application SECurity**" series, which is a portion of the content of WASEC, an e-book on web application security I've written.

Here is a list of all the articles in this series:

1. Web security demystified: WASEC
2. Introduction
3. Understanding the browser
4. Security at the HTTP level
5. HTTP headers to secure your application
6. Hardening HTTP cookies
7. Situationals

If you've enjoyed the content of this article, consider buying the complete ebook on either the Kindle store or Leanpub.

Imagine being a backend developer who needs to implement *sessions* in an application: the first thing that comes to your mind is to issue a *token* to clients and ask them to send this

token with their subsequent requests. From there onwards you are going to be able to identify clients based on the token included in their request.

HTTP cookies were born to standardize this sort of mechanism across browsers: they're nothing more than a way to store data sent by the server and send it along with future requests. The server sends a cookie, which contains small bits of data, the browsers stores it and sends it along with future requests to the same server.

Why would we bother about cookies from a security perspective? Because the data they contain is, more often than not, extremely sensitive — cookies are generally used to store session IDs or access tokens, an attacker's holy grail. Once they are exposed or compromised, attackers can impersonate users, or escalate their privileges on your application.

Securing cookies is one of the most important aspects when implementing sessions on the web: this chapter will, therefore, give you a better understanding of cookies, how to secure them and what alternatives can be used.

## What's behind a cookie?

A server can send a cookie using the `Set-Cookie` header:

```
HTTP/1.1 200 Ok
Set-Cookie: access_token=1234
...
```

A client will then store this data and send it in subsequent requests through the `Cookie` header:

```
GET / HTTP/1.1
Host: example.com
Cookie: access_token=1234
...
```

Note that servers can send multiple cookies at once:

```
HTTP/1.1 200 Ok
Set-Cookie: access_token=1234
Set-Cookie: user_id=10
...
```

and clients can do the same in their request:

```
GET / HTTP/1.1
Host: example.com
Cookie: access_token=1234; user_id=10
...
```

In addition to the plain *key* and *value*, cookies can carry additional directives that limit their time-to-live and scope:

## Expires

Specifies when a cookie should expire, so that browsers do not store and transmit it indefinitely. A clear example is a session ID, which usually expires after some time. This directive is expressed as a date in the form of `Date: <day-name>, <day> <month> <year> <hour>:<minute>:<second> GMT`, like `Date: Fri, 24 Aug 2018 04:33:00 GMT`. Here's a full example of a cookie that expires on the 1st of January 2018:

```
access_token=1234;Expires=Mon, 1st Jan 201
```

## Max-Age

Similar to the `Expires` directive, `Max-Age` specifies the number of seconds until the cookie should expire. A cookie that should last 1 hour would look like the following:

```
access_token=1234;Max-Age=3600
```

### Domain

This directive defines which hosts the cookie should be sent to. Remember, cookies generally contain sensitive data, so it's important for browsers not to leak them to untrusted hosts. A cookie with the directive `Domain=trusted.example.com` will not be sent along with requests to any domain other than `trusted.example.com`, not even the root domain (`example.com`). Here's a valid example of a cookie limited to a particular subdomain:

```
access_token=1234;Domain=trusted.example.c
```

### Path

Similar to the `Domain` directive, but applies to the URL path (`/some/path`). This directive prevents a cookie from being shared with untrusted paths, such as in the following example:

```
access_token=1234;Path=/trusted/path
```

## Session and persistent cookies

When a server sends a cookie without setting its `Expires` or `Max-Age`, browsers treat it as a *session cookie*: rather than guessing its time-to-live or apply funny heuristics, the browser deletes it when it shuts down.

A *persistent cookie*, on the contrary, is stored on the client until the deadline set by its `Expires` or `Max-Age` directives.

It is worth to note that browsers might employ a mechanism known as *session restoring*, where session cookies can be recovered after the client shuts down: browsers have implemented this kind of mechanism to conveniently let users resume a session after, for example, a crash. Session restoring could lead to unexpected issues if we're expecting session cookies to expire within a certain timeframe (eg. we're absolutely positive a session would not last more than X time). From a browser's perspective, session restoring is a perfectly valid feature, as those cookies are left in the hands of the client, without an expiration date. What the client does with those cookies does not affect the server, who is unable to detect whether the client shut down at any point in time. If the client wishes to keep session cookies alive forever that's no concern for the server — it would definitely be a questionable implementation, but there's nothing the server could do about it.

I don't think there is a clear-cut winner between session and persistent cookies, as both might serve different purposes very well: what I've observed, though, is that Facebook, Google, and similar services will use persistent cookies. From personal experience, I've generally always used persistent cookies — but never had to tie critical information, such as a social security number or a bank account's balance, to a session. In some contexts you might be required to use session cookies due to compliance requirements: I've seen auditors asking to convert

all persistent cookies to session ones. When people ask me "*should I use X or Y?*" my answer is "it depends on the context": building a guestbook for your blog carries different security ramifications than building a banking system. As we will see later in this series, I would recommend to understand your context and try to build a system that's *secure enough*: absolute security is utopia, just like a 100% SLA.

## Host-only

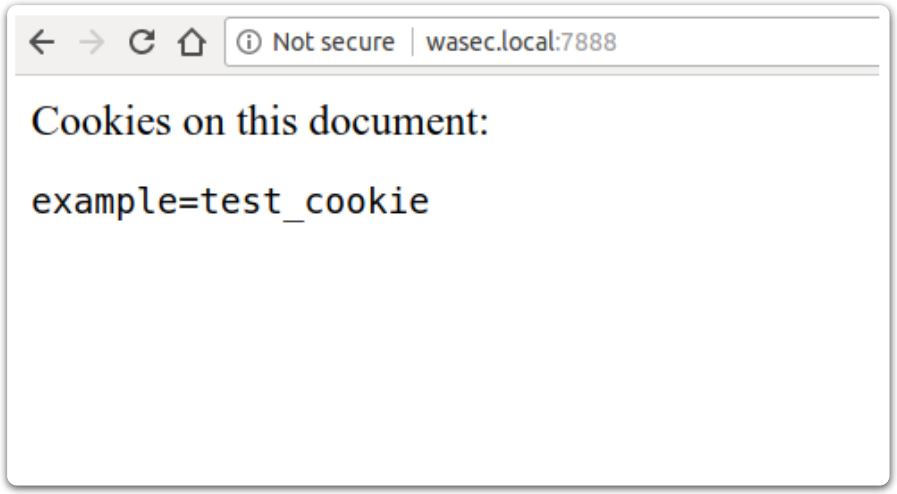
When a server does not include a `Domain` directive the cookie is to be considered a `host-only` one, meaning that its validity is restricted to the current domain only.

This is a sort of "default" behavior from browsers when they receive a cookie that does not have a `Domain` set. You can find a small example I wrote at github.com/odino/wasec/tree/master/cookies: it's a simple web app that sets cookies based on URL parameters, and prints cookies on the page, through some JavaScript code:

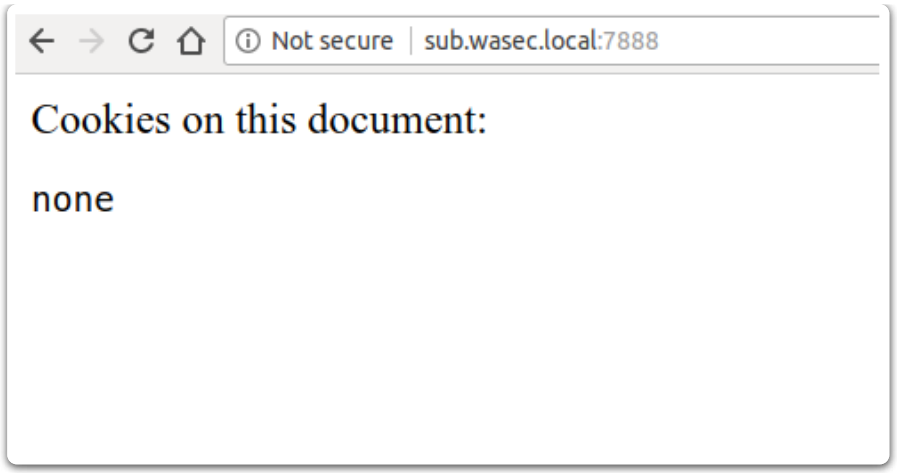
```
<html>
  <div id="output"/ >
  <script>
    let content = "none";

    if (document.cookie) {
      let cookies = document.cookie.split(
      content = ''
```

```
      cookies.forEach(c => {
        content += "<p><code>" + c + "</co
      })
    }

    document.getElementById('output').inne
  </script>
</html>
```

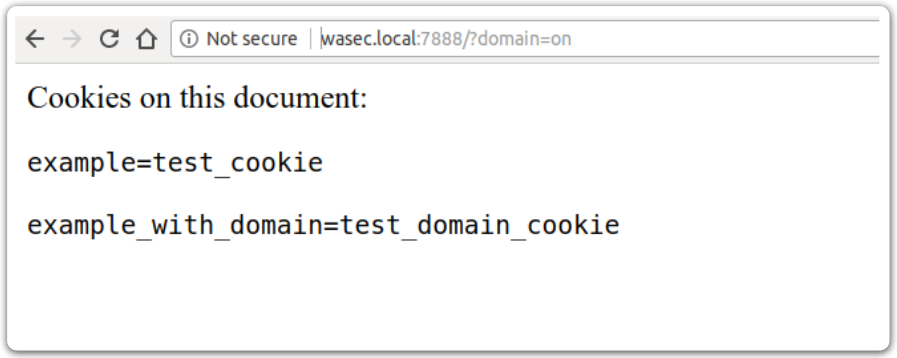
If you follow the instructions in the `README` you will be able to access a webserver at wasec.local:7888, which illustrates how `host-only` cookies work:



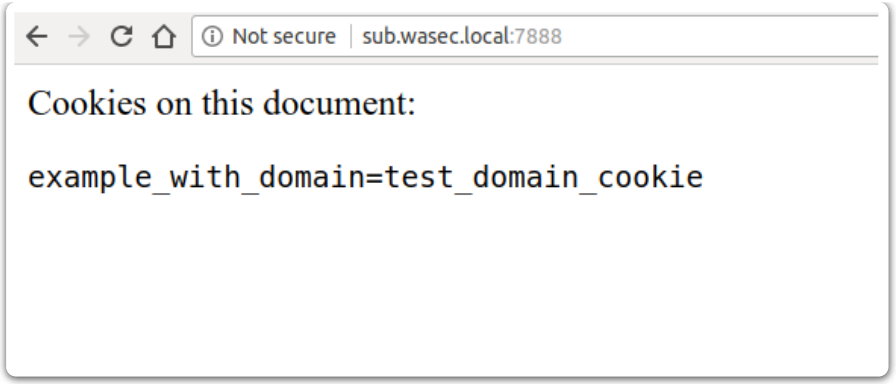

If we then try to visit a subdomain, the cookies we set on the main domain are not going to be visible — try navigating to sub.wasec.local:7888:




A way to circumvent this limitation is, as we've seen earlier, to specify the `Domain` directive of the cookie, something that we can do by visiting wasec.local:7888/?domain=on:




If we have a look at the application running on the subdomain, we will now be able to see cookies set on the parent domain, as they use `Domain=wasec.local`, which allows any domain "under" `wasec.local` to access the cookies:




In HTTP terms, this is how the responses sent from the server look like:

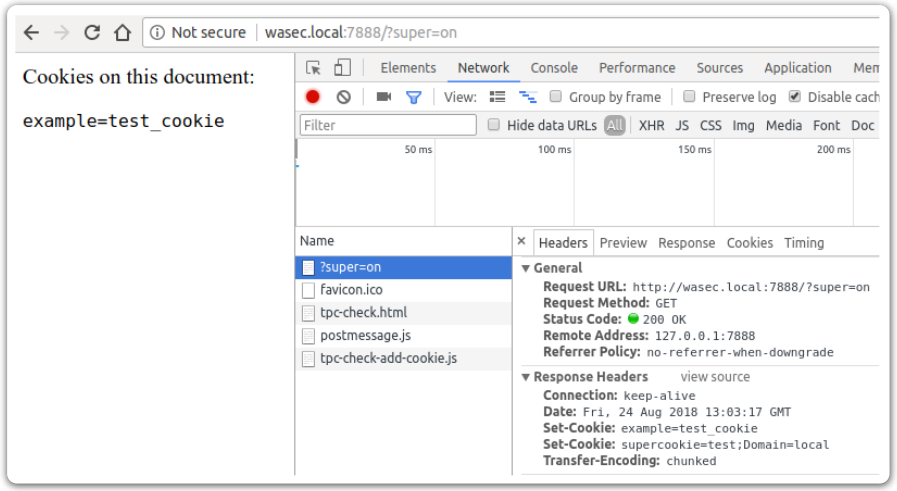
```
~ ▷ curl -I http://wasec.local:7888
HTTP/1.1 200 OK
Set-Cookie: example=test_cookie
Date: Fri, 24 Aug 2018 09:34:08 GMT
Connection: keep-alive

~ ▷ curl -I "http://wasec.local:7888/?dom
HTTP/1.1 200 OK
Set-Cookie: example=test_cookie
Set-Cookie: example_with_domain=test_doma
Date: Fri, 24 Aug 2018 09:34:11 GMT
Connection: keep-alive
```

## Supercookies

What if we were able to set a cookie on a top-level domain (abbr. TLD) such as `.com` or `.org`? That would definitely be a huge security concern, for two main reasons:

- user privacy: every website running on that specific TLD would be able to track information about the user in a shared storage
- information leakage: a server could mistakenly store a sensitive piece of data in a cookie available to other sites

Luckily, TLD-cookies, otherwise known as supercookies, are disabled by web browsers for the reasons I mentioned above: if you try to set a supercookie, the browser will simply refuse to do so. If we append the parameter `super=on` in our example, we will see the server trying to set a supercookie, while the browser ignores it:




In today's web, though, there are other ways to keep track of users — ETag tracking being an example of this. Since cookies are usually associated with tracking, these techniques are often referred to as supercookies as well, even though they do not rely on HTTP cookies. Other terms that may refer to the same set of

technologies and practices are permacookies (permanent cookies) or zombiecookies (cookies that never die).

> ### *Unwanted Verizon ads*
>
> *Companies love to make money out of ads, that's no news. But when ISPs start to aggressively track their customers in order to serve unwanted ads — well, that's a different story.*
>
> *In 2016, Verizon was found guilty of tracking users without their consent, and sharing their information with advertisers. This resulted in a fine of $1.35 million and the inability, for the company, to continue with their questionable tracking policy.*
>
> *Another interesting example was Comcast, who used to include custom JavaScript code in web pages served through its network.*
>
> *Needless to say, if all web traffic would be served through HTTPS we wouldn't have this problem, as ISPs wouldn't be able to decrypt and manipulate traffic on-the-fly.*

## Cookie flags that matter

Until now we've barely scratched the surface of HTTP cookies: it's now time for us to taste the real juice.

There are 3 very important directives (`Secure`, `HttpOnly`, and `SameSite`) that should be understood before using cookies, as they heavily impact how cookies are stored and secured.

## Encrypt it or forget it

Cookies contain very sensitive information: if attackers can get a hold of a session ID, they can impersonate users by hijacking their sessions.

Most *session hijacking* attacks usually happen through a *man-in-the-middle* who can listen to the unencrypted traffic between the client and server, and steal any information that's been exchanged. If a cookie is exchanged via HTTP, then it's vulnerable to MITM attacks and session hijacking.

To overcome the issue, we can use HTTPS when issuing the cookie and add the `Secure` flag to it: this instruct browsers to never send this cookie in plain HTTP requests.

Going back to our practical example, we can test this out by navigating to https://wasec.local:7889/?secure=on. The server sets 2 additional cookies, one with the `Secure` flag and one without:


When we go back and navigate to the HTTP version of the site, we can clearly see that the `Secure` cookie is not available in the page — try navigating to wasec.local:7888:


We can clearly see that the HTTPS version of our app set a cookie that's available to the HTTP one (the `not_secure` one), but the other cookie, flagged as `Secure`, is nowhere to be seen.

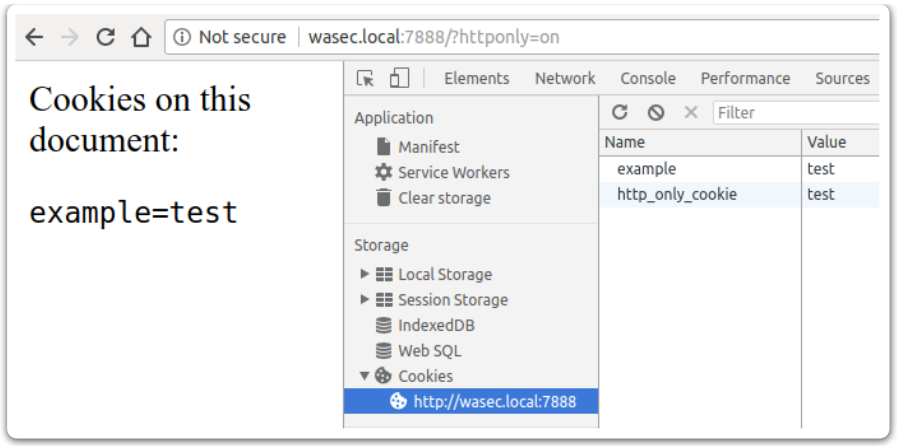
Marking sensitive cookies as `Secure` is an incredibly important aspect of cookie security: even if you serve all of your traffic to HTTPS, attackers could find a way to set up a plain old HTTP page under your domain and redirect users there. Unless your cookies are `Secure`, they will then have access to a very delicious meal.

## JavaScript can't touch this

As we've seen earlier in this series, XSS attacks allow a malicious user to execute arbitrary JavaScript on a page: considering that you could read the contents of the cookie jar with a simple `document.cookie`, protecting our cookies from untrusted JavaScript access is a very important aspect of hardening cookies from a security standpoint.

Luckily, the HTTP spec took care of this with the `HttpOnly` flag: by using this directive we can instruct the browser not to share the cookie with JavaScript. The browser then removes the cookie from the `window.cookie` variable, making it impossible to access the cookie via JS.

If we look at the example at wasec.local:7888/?httponly=on we can clearly see how this works. The browser has stored the cookie (as seen on the DevTools) but won't share it with JavaScript:




The browser will then keep sending the cookie to the server in subsequent requests, so the server can still keep track of the client through the cookie: the trick, in this case, is that the cookie is never exposed to the end-user, and remains "private" between the browser and the server.

The `HttpOnly` flag helps mitigate XSS attacks by denying access to critical information stored in a cookie: using it makes it harder for an attacker to hijack a session.

> ## *Circumventing HttpOnly*
>
> *In 2003, researchers found an interesting vulnerability around the* `HttpOnly` *flag: Cross-Site Tracing (abbr. XST).*
>
> *In a nutshell, browsers wouldn't prevent access to* `HttpOnly` *cookies when using the* `TRACE` *request method. While most browsers have now disabled this method, my recommendation would be to disable* `TRACE` *at your*

> *webserver's level, returning the* `405 Not allowed` *status code.*

## SameSite: the CSRF killer

Last but not least, 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* (abbr. 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 (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 3rd party website sets up an HTML form as such:

```
1 <form action="https://bank.com/transfer" m
2 <input type="hidden" name="destination" va
3 <input type="hidden" name="amount" value="
```

```
<input type="submit" value="CLICK HERE TO
</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.

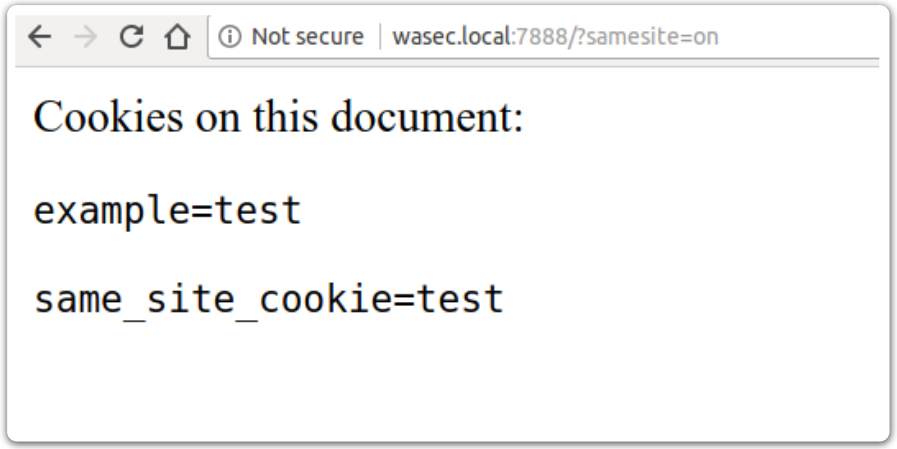
Traditionally, there have been 2 ways to get rid of CSRF:

- `Origin` and `Referer` headers: the server could verify that these headers come from trusted sources (ie. `https://bank.com`). The downside of this approach is that, as we've seen earlier in this series, neither the `Origin` nor the `Referer` 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 a complicated task, but a simpler solution would be more than welcome.

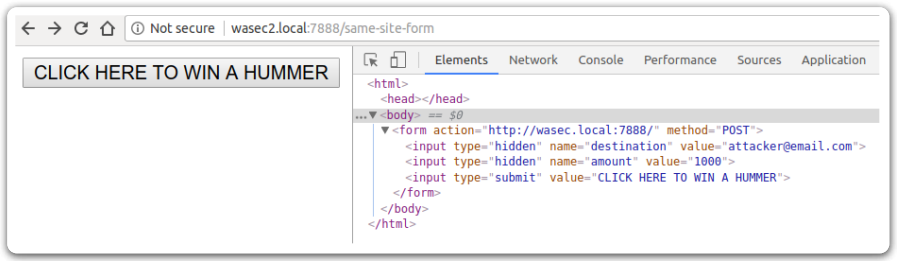
`SameSite` cookies aim to supersede the solutions mentioned above once and for all: when you tag a cookie with this flag, you tell the browser not to include the cookie in requests that were generated by different origins. When the browser initiates a request to your server and a cookie is tagged as `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.

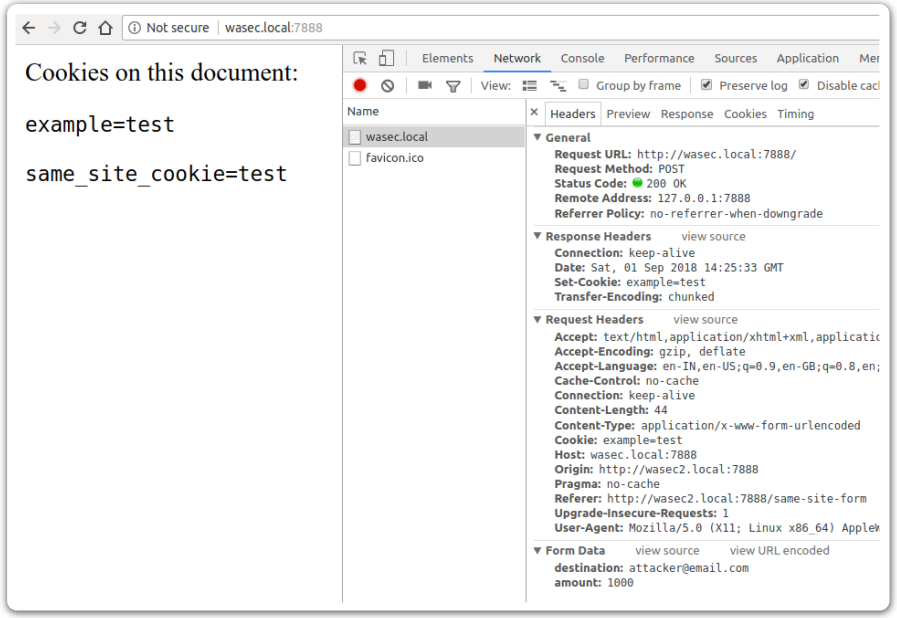



If we then visit wasec2.local:7888/same-site-form we will see an example HTML form that will trigger a cross-site request:

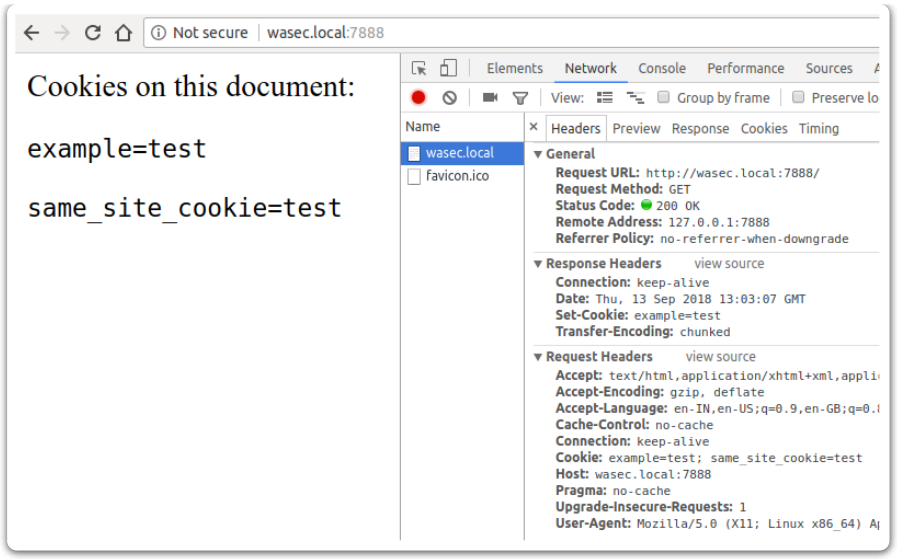



If we click on the submit button of the form, we will then be able to understand the true power of this flag — the form will

redirect us to wasec.local:7888, but there is no trace of the `SameSite` cookie in the request made by the browser:




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:




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 2 variants: `Lax` and `Strict`. Our example uses the former, as it allows top-level navigation to a website to include the cookie; when you tag a cookies 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.

> ## *Cookie flags are important*

> *Let's recap what we've learned about cookies 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 (eg. 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*

## Alternatives

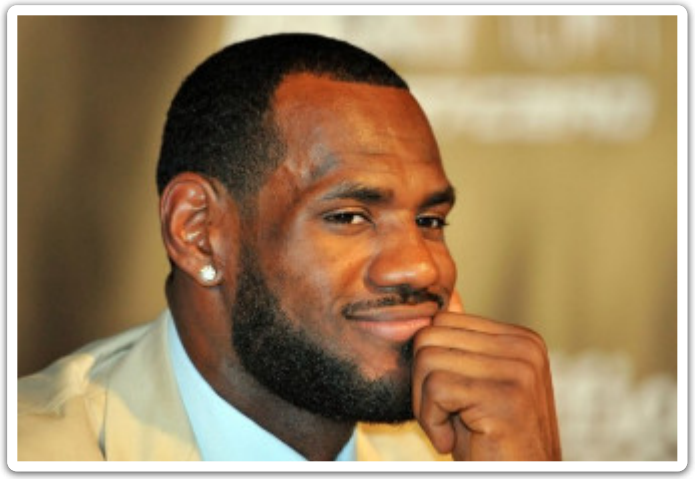
Reading all of this material about cookies and security you might be tempted to say "I really want to stay away from cookies!": the reality is that, as of now, cookies are your best bet if you want to implement some sort of session mechanism over HTTP. Every now and then I'm asked to evaluate alternatives to cookies, so I'm going to try and summarize a couple things that get mentioned very often:

- localStorage: especially in the context of single-page applications (SPA), localStorage gets sometimes mentioned when discussing where to store sensitive tokens: the problem with this approach, though, is that localStorage does not offer any kind of protection against XSS attacks. If an

attacker is able to execute a simple `localStorage.getItem('token')` on a victim's browser, it's game over. `HttpOnly` cookies easily overcome this issue.

- JWT: JSON Web Tokens define a way to securely create access tokens for a client. JWT is a specification that defines how an access token would look like and does not define where is the token going to be stored. In other words, you could store a JWT in a cookie, the localStorage or even in memory — so it doesn't make sense to consider JWTs an "alternative" to cookies.

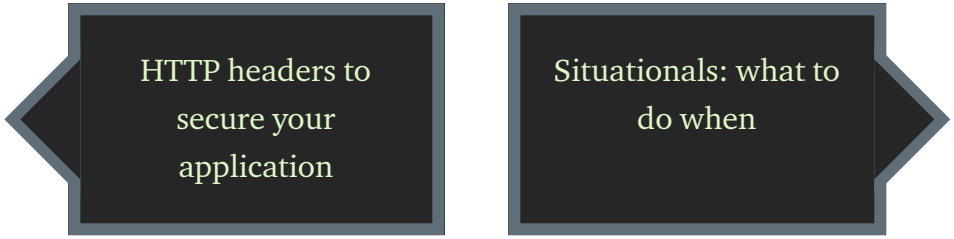
## What would *LeBron* do?



It's time to move on from the HTTP protocol and its features, such as cookies: we've been on a long journey, dissecting why cookies were born, how they're structured and how you can protect them by applying some restrictions on their `Domain`, `Expires`, `Max-Age` and `Path` attributes, and how other flags such as `Secure`, `HttpOnly` and `SameSite` are vital in hardening cookies.

Let's move forward and try to understand what we should do, from a security perspective, when we encounter a particular situation: the next article will try to provide advice based on best practices and past experience.

It's time to introduce the *situationals*.




~

## In the mood for some more reading?

- fwupd is the best thing that ever happened to Linux (15 April 2021)
- Avoid battery draining on your Linux-flavored Dell XPS (31 January 2021)
- Combining two numbers into a unique one: pairing functions (05 December 2020)
- Running CI tests in Kubernetes through Github Actions (20 March 2020)
- I've decided to make the WASEC ebook free during these trying times (20 March 2020)
- Local k8s development in 2020 (31 December 2019)

- WASEC, a book about Web Application Security, is now available for sale (25 November 2019)

...or check the archives.

---