The great SameSite confusion

The great SameSite confusion - Julien Cretel, jub0bs.com.

- Published: 2021-01-29
- Original: <https://jub0bs.com/posts/2021-01-29-great-samesite-confusion/>
- Preserved from: https://jub0bs.com/posts/2021-01-29-great-samesite-confusion/ (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

12 minutes

The great SameSite confusion

In this post, I dissect a common misconception about the `SameSite` cookie attribute and I explore its potential impact on Web security.

TL;DR ¶

- The `SameSite` cookie attribute is not well understood.
- Conflating *site* and *origin* is a common but harmful mistake.
- The concept of *site* is more difficult to apprehend than meets the eye.
- Some requests are cross-origin but same-site.
- `SameSite` only has effects on cross-site requests.
- `SameSite` paints a target on your subdomains' back.
- Misguided practitioners may unduly eschew `SameSite=Strict`.

The advent of `SameSite` ¶

You undoubtedly have heard of the `SameSite` cookie attribute. It made headlines when, in February 2020, Chrome started rolling out changes to `SameSite`'s default behaviour. Intended as a defence-in-depth mechanism against cross-site attacks, such as cross-site request forgery (CSRF) and cross-site script inclusion (XSSI), `SameSite` had been lying dormant at the heart of implementing browsers since its inception in 2016.

`SameSite`'s activation in early 2020 required labourious adjustments by some websites to maintain third-party access, but was widely hailed as a welcome addition to browser defences. Both in anticipation and in reaction to `SameSite`'s activation in browsers, posts started sprouting on blogs all over the Web to spread the word about the mechanics of the "new" cookie attribute.

Playing fast and loose with terminology ¶

Some of those posts were of admirable precision, such as Rowan Merewood's web.dev piece entitled "SameSite cookies explained". Unfortunately, relatively few of the posts about `SameSite` went through the effort of clarifying the concept of *site*, from which the technical concepts of *same-site request* and *cross-site request* are of course derived.

Moreover, many posts, including those produced by influential members of the infosec community, appeared to use the terms "origin" and "site" interchangeably or, at least, somewhat loosely.

Back in February 2019, Kristian Bremberg wrote the following on the venerable Detectify blog:

>

The SameSite attribute is rather new and provides excellent protection against CSRF attacks. If a cookie uses the SameSite attribute, the web browser will make sure that the request made with the cookie came from the **origin** that sat [sic] the cookie.

(my emphasis)

Infosec superstar Troy Hunt himself, in a seminal post entitled "Promiscuous Cookies and Their Impending Death via the SameSite Policy" and published in early January 2020, described the effects of the different `SameSite` attribute values as follows:

>

- None: what Chrome defaults to today without a SameSite value set
- Lax: some limits on sending cookies on a **cross-origin** request
- Strict: tight limits on sending cookies on a **cross-origin** request

(my emphasis)

And a few months later, in an otherwise fascinating post analysing how the advent of `SameSite` was affecting a range of vulnerabilities cherished by hackers, the Reconless team wrote the following:

>

After the update, all cookies without an explicit `SameSite` attribute will be treated as having `SameSite=Lax`. This means **cross-origin** requests no longer carry cookies, except for top-level navigations.

(my emphasis)

Domain, *host*, *origin*, *site*... Using those terms loosely in informal communication is natural; such laxity in the use of terminology can be forgiven, and if you've been guilty of it yourself, you're in

good company.

Are “site” and “origin” interchangeable? ¶

However, the advent of the `SameSite` cookie attribute raises some questions…

- Is a careful distinction between “origin” and “site” warranted, here?
- Is it just a distinction without a difference?
- Is a *cross-site* request no different from a *cross-origin* request?
- Could the cookie attribute have as well been named “SameOrigin”, then?
- Or, if there is indeed a real difference between “site” and “origin”, does it matter to practitioners?
- And, if the difference does matter, how so?

You may have already guessed the answers from the title of this post: “site” has a very technical meaning in the context of `SameSite`, yet it is unduly neglected; and the distinction between *site* and *origin does* matter, yet the two concepts are frequently conflated.

This lapse in terminology didn’t escape everyone. That Google’s *Developer Advocate of Web* Eiji Kita mura felt the need to dedicate a whole blog post about the distinction between “origin” and “site”, only a few months after Chrome activated `SameSite`, is revealing.

In order to understand why the distinction matters, you first need to understand the difference between *origin* and *site*.

What do we mean by “origin”? ¶

If you work with Web technologies, you have at least some familiarity with the Same-Origin Policy (SOP), arguably one of the main pillars of Web security. The concept of *origin* of a URI is of course central to the SOP and, as such, it is relatively well-understood. Section 3.2 of RFC 6454 defines an *origin* as a triple:

>

Roughly speaking, two URIs are part of the same origin (i.e., represent the same principal) if they have the same scheme, host, and port.

The port is optional; if not specified, the default port associated to the scheme is implied (e.g. 80 for `http` and 443 for `https`). MDN Web Docs has a series of clarifying examples.

What do we mean by “site”? ¶

Behind the painfully generic term that is “site” hides a concept fundamentally more difficult to grasp than that of *origin*. For one thing, the term “site” was not always a technical one: it predates the SOP and was in in common use when attacks like cross-site scripting came onto the scene. Furthermore, the modern concept of *site* is fraught with technical difficulties. It is intimately linked to that of a host’s *registrable domain*, which the URL Living Standard defines as

>

[…] a domain formed by the most specific public suffix, along with the domain label immediately preceeding it, if any.

(A host's registrable domain is also known as its "eTLD+1", short for "effective top-level domain plus one".)

In the simplest of cases, **the site of an origin simply corresponds to the registrable domain (if any) of the origin's host**.

Two examples, to fix ideas:

- The site of `https://www.example.org` is `example.org`, because `org` is the host's most specific public suffix and, therefore, `example.org` is the host's eTLD+1.
- The site of `https://jub0bs.github.io` is `jub0bs.github.io`, because `github.io` is the host's most specific public suffix and, therefore, `jub0bs.github.io` is the host's eTLD+1.

Yes! Perhaps surprisingly, `github.io` is a public suffix!

It's worth noting, however, that the concept of registrable domain is a fluid one, because it relies on the Public-Suffix List, a list which is not set in stone but subject to change over time. Not to mention that different browsers may not necessarily stay abreast of changes to the Public-Suffix list at the same pace.

The technicalities do not end there! As web.dev warns us, the concept of *site* is still evolving and will soon incorporate the scheme also. The change currently sits behind a flag in Chrome but will soon be rolled out. However, to sidestep this difficulty and prolong the relevance of this post, I'll only consider origins whose scheme is `https` in what follows.

Same-site vs. cross-site requests ¶

Now that we've dealt with the concept of *site*, we can finally discuss the concepts of *same-site* and *cross-site* requests. A given request is either *same-site* or *cross-site*. Whether a request is same-site or cross-site depends on the comparison between the sites of the request's source origin and target origin:

- If the two sites are identical, the request is said to be *same-site*;
- If the two sites are different, the request is said to be *cross-site*.

Here are three examples:

- A request sent from `https://foo.example.org` to `https://bar.example.org` is same-site, because the site of both origins is `example.org`.
- A request sent from `https://foo.github.io` to `https://bar.github.io` is cross-site, because the site of the first origin is `foo.github.io`, whereas the site of the second origin is `bar.github.io`.
- A request sent from `https://foo.bar.example.org` to `https://bar.example.org` is same-site, because the site of both origins is `example.org`.

If you've made it this far down the present post, I thank you for your patience. Take heart: the payoff is near!

Cross-origin, same-site requests ¶

All cross-site requests are necessarily cross-origin; that much is clear. However, as shown by the first and third examples above and as illustrated by the crude Venn diagram below, not all cross-origin requests are cross-site.

[image: Venn-diagram]

And the `SameSite` cookie attribute is only concerned with cross-site requests; it has no effect on cross-origin requests that happen to be same-site. This is why the distinction between *origin* and *site* is important.

Online demo ¶

To prove my point, I've drawn inspiration from Troy Hunt's post and I've deployed a simple Go server to `samesitedemo.jub0bs.com` composed of two endpoints. Endpoint `/setcookie` sets a `SameSite=Strict` cookie, like so:

```
Set-Cookie: StrictCookie=foo; Path=/; Max-Age=3600; SameSite=Strict
```

Endpoint `/readcookie` prints the cookie (if any) attached to the request. I've also set up two "attacking" pages:

- https://jub0bs.github.io/samesitedemo-attacker-foiled
- https://samesitedemo-attacker.jub0bs.com/

Both pages simply consist in a single link to `https://samesitedemo.jub0bs.com/readcookie`.

To fix ideas, here's what you can do:

- Navigate to https://samesitedemo.jub0bs.com/setcookie. Doing so will set the `Strict` cookie in your browser.
- Navigate to https://jub0bs.github.io/samesitedemo-attacker-foiled and follow the link on that page. Because the site of the attacking URI (`jub0bs.github.io`) is different from that of the target URI (`jub0bs.com`), the browser does not attach the cookie to the request resulting from following the link, and no cookie gets printed in the response. `SameSite=Strict` works as expected, and the "attack" is foiled.
- Now navigate to https://samesitedemo-attacker.jub0bs.com/ and follow the link on that page. Because the site of the attacking URI (`jub0bs.com`) is identical to that of the target URI (`jub0bs.com`), the browser *does* attach the cookie to the request resulting from following the link, and the cookie *does* get printed in the response. The `SameSite` cookie attribute simply doesn't apply, in this case, and the "attack" is a success.

The cost of conflating site and origin ¶

False sense of security ¶

Implying that `SameSite` applies to *all* cross-origin requests is harmful, because it may lead practitioners to believe, incorrectly, that `SameSite` protects their users against *all* cross-origin abuse. Such a misconception is particularly dangerous to practictioners who neglect to scrutinise the security level of their subdomains. In particular,

- a subdomain takeover, or
- an instance of cross-site scripting (XSS) on a subdomain of the same site, or
- an instance of HTML injection on a subdomain of the same site

may be sufficient for an attacker to bypass the relative protection that `SameSite` provides.

A subdomain takeover is an attack that was popularised by Detectify as far back as 2014. It consists in exploiting a dangling DNS record on a subdomain in order to take control of some or all of the content served by the subdomain in question. An attacker may leverage a subdomain takeover to various ends: defacement, phishing, etc… but also cross-origin attacks that would otherwise not be possible!

For instance, if the attacker were capable of taking over `https://vulnerable.example.org`, he or she may be in a position to send, from client code running in the context of the vulnerable subdomain, malicious requests to `https://example.org` (or any subdomain thereof); and such requests, being same-site, would carry all the relevant cookies, regardless of the value of their `SameSite` attribute!

A subdomain takeover may not even be required for such cross-origin, same-site attacks. The presence of an XSS or HTML-injection vulnerability on a subdomain of the same site may be all the attacker needs to send malicious cross-origin, same-site requests.

Note: “Cross-site request forgery” is a misnomer in both of the cases described above, because the attacking site and the targeted site are the same; “same-site cross-origin request forgery” (SSCORF?) would be more apt a term to describe such an attack, but I doubt it will ever achieve common use.

What I find fascinating is that `SameSite` is likely to focus the attention of savvy attackers on your subdomains and sibling domains even more than in the past, because those domains are fast becoming the only refuge for cross-origin attacks against battle-hardened Web apps.

Slower adoption of `SameSite=Strict` ¶

Besides, this misconception may also slow down the adoption of the `Strict` value in favour of the `Lax` one. Some people indeed actively discourage practitioners from using `Strict` because they perceive it as a greater impediment to usability than it actually is. For instance, on Dareboost’s blog, you can read the following statement:

>

if we were using `Strict` `Same-Site` on `dareboost.com`, by clicking this link, you would not be detected as logged in, whether you were connected or not.

That statement is incorrect: the link in question is present on `https://blog.dareboost.com` and leads to `https://www.dareboost.com`; therefore, the request triggered by clicking the link would be same-site and carry all the cookies scoped at the `www` subdomain or the parent domain.

The author concludes:

>

The behaviour can be confusing for the final user, so you would prefer using the `Lax` mode.

This is just one example of unjustly disparaging the `Strict` value, but I'm sure you could find more examples elsewhere on the Web.

Parting words ¶

I've reached out to all the people I quoted above who I believe are inaccurate in their description of `SameSite`'s mechanics. So far, only Kristian Bremberg from Detectify and Edwin Foudil (also known as @edoverflow) from Reconless have replied to me. I have high hopes that they will amend their posts after reading this one. I've left a public comment on Troy Hunt's post, but he hasn't gotten back to me yet; and I've reached out to Dareboost, but I have yet to hear back from them.

Edit (2021/02/10): Dareboost have since amended their post.

Remember: `SameSite` is a powerful defence-in-depth mechanism for protecting users against cross-site attacks, but it is powerless against cross-origin, same-site attacks. Don't miss this subtlety! Otherwise, if you're on the defensive side, you may be lulled into a false sense of security and you may get blindsided by attacks you would not have thought possible; and if you're on the offensive side, you may miss out on vulnerability findings and perhaps even sizable bug bounties.

Addendum (2021/01/31) ¶

Since the first publication of this post, I have found more noteworthy instances of incautious use of the terms "origin" and "site" when describing `SameSite`'s mechanics. I have not attempted to contact the authors of the pieces I quote below.

Back in 2016, Sjoerd Langkemper, Web application Hacker at Qbit Cyber Security, wrote the following on his blog:

>

This table shows what cookies are sent with **cross-origin** requests. As you can see cookies without a same-site attribute [...] are always sent. Strict cookies are never sent. Lax cookies are only send with a top-level get request.

(my emphasis)

In May 2018, Artur Janc and Mike West (the author of the *Same-site Cookies* Internet Draft himself) released a report (PDF) entitled "How do we Stop Spilling the Beans Across Origins?" in which you can read the following:

>

SameSite cookies do not directly prevent attackers from loading **cross-origin** resources, **but they cause such requests to be sent without credentials**, rendering the responses of little value to the attacker.

(my emphasis)

Finally, the Wikipedia page about CSRF itself claims the following:

>

If this attribute is set to “strict”, then the cookie will only be sent on **same-origin** requests, making CSRF ineffective.

(my emphasis)

Edit (2021/02/07): the Wikipedia page has since been corrected.

Acknowledgments ¶

I’d like to thank Fredrik N. Almroth, from Detectify, who kindly agreed to review a draft of this post before publication.

SameSitecookiesoriginsame-sitesitesubdomainsubdomain takeover

2542 Words

2021-01-29 13:15 +0000
