CSE 127 Scribe Notes Lecture 8: Web Intro

These notes were scribed by students in CSE 127 Winter 2021. They have been lightly edited but may still contain errors.

1 HTTP Protocol

The Hypertext Transfer Protocol (HTTP) was written by Tim Berners-Lee in 1989. It is a hypertext protocol designed to allow fetching of resources such as HTML documents. These resources are specified and located by their uniform resource locator/location (URL), which has a defined format and structure.

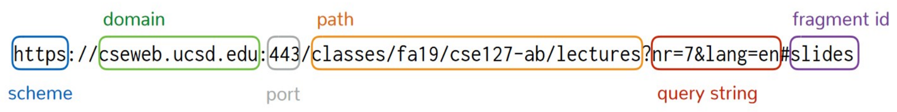

domain
path
fragment id
https://cseweb.ucsd.edu:443/classes/fa19/cse127-ab/lectures?nr=7&lang=en#slides
scheme
port
query string

Figure 1: URL Structure

The scheme specifies the protocol of the resource, which is typically HTTP, HTTPS (Hypertext Transfer Protocol Secure), or FTP (File Transfer Protocol). The domain specifies the "name" of the host. The port is optional, but HTTPS typically uses port 443. The path specifies the location of the file. The query string is optional and can include basic parameters that are forwarded along to scripts. The fragment id is also optional and can be used to specify location within a page as well as many other things.

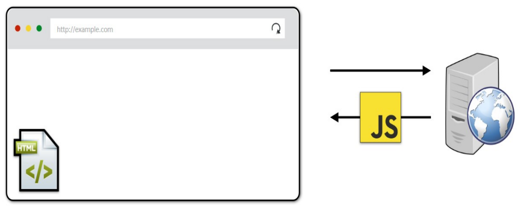
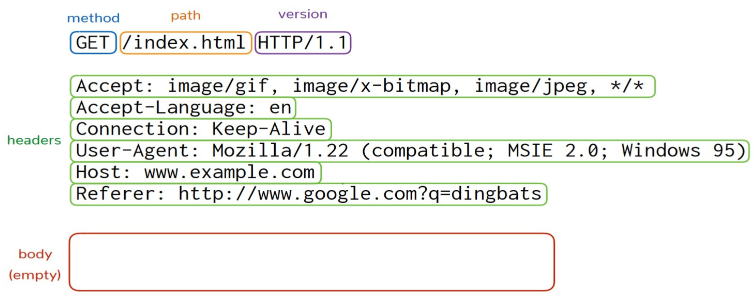


Figure 2: Browser parses HTML and requests Javascript script

In the HTTP protocol, clients and servers work by exchanging individual messages. The client sends a request to the server over something like the browser, and the server responds by sending the resource corresponding to the request. The browser can then parse the resource, and send requests for necessary sub-resources to render the resource.



```
GET /index.html HTTP/1.1

Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: www.example.com
Referer: http://www.google.com?q=dingbats
```

Figure 3: Example HTTP GET request, which has an empty body.

HTTP requests follow a specific text format. The method specifies the "mode" of the request, including GET, POST, PUT, PATCH, and DELETE.

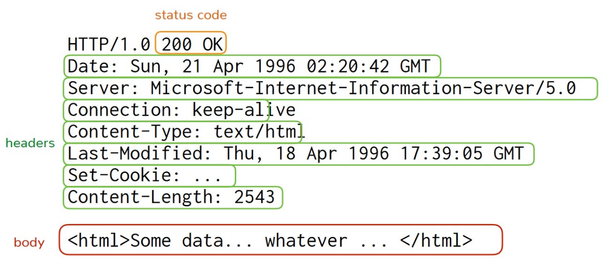
- GET: Get the resource at the specified URL.
- POST: Create new resource at URL with payload.
- PUT: Replace current representation of the target resource with request payload.
- PATCH: Update part of the resource.
- DELETE: Delete the specified URL.

In practice, only GET and POST are encountered most often because PUT, PATCH, and DELETE are not supported by old browsers. The path specifies the path of the resource requested and the version specifies the HTTP version. The header specifies additional information such as accepted file type, user information, and host. The body contains information associated with the request.

HTTP response contains a status code in response to the client's request. The header contains information such as date and file type. The body of a response contains the actual HTML document that is requested and returned.

2 HTTP Cookies and Browser Execution Models

HTTP cookies are a heavily used resource that primarily serves as the method for implementing states on the client browser. It consists of a small

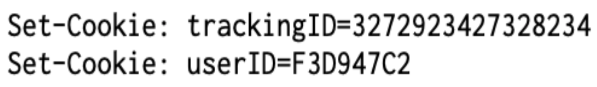


```
HTTP/1.0 200 OK
Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Set-Cookie: ...
Content-Length: 2543

<html>Some data... whatever ... </html>
```

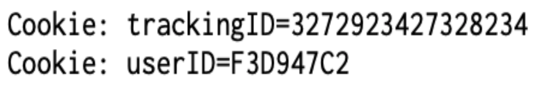
Figure 4: Example response

piece of data, passed from the server to the client browser, to be stored and used in subsequent interactions. When the client browser sends the cookie along with an HTTP request, the server is able to identify the user, enabling essential features such as session management, personalization, and tracking.



```
Set-Cookie: trackingID=3272923427328234
Set-Cookie: userID=F3D947C2
```

Figure 5: Header included in Server HTTP Response



```
Cookie: trackingID=3272923427328234
Cookie: userID=F3D947C2
```

Figure 6: Header included in Client HTTP Request

There have been newer versions of HTTP introduced. Released in 2015, HTTP/2 allows for pipelining requests for multiple objects, multiplexing multiple requests over a single TCP connection, header compression, and server push. HTTP/3, which is currently an Internet Draft although widely supported by browsers, uses the QUIC protocol instead of TCP, allowing for greater encryption of sent packets.

After receiving an HTTP response from the server, the client browser will execute the code contained in the body of the response following an execution model, producing interfaces users see on their screens. A basic browser execution model involves 1)loading the page content, 2)parsing HTML code, 3)fetching required resources as specified in the HTML document, such as images, CSS, JavaScript, etc., 4)running JavaScript, and 5)listening and responding to events such as a mouse click, cursor hovering over a button, loading certain information, and more.

Frames are a way that pages can display contents from other sources on a single page. A frame appears as a rigid visible division of the page, while an iFrame (or inline frame) is a floating frame, meaning its position is adjustable.

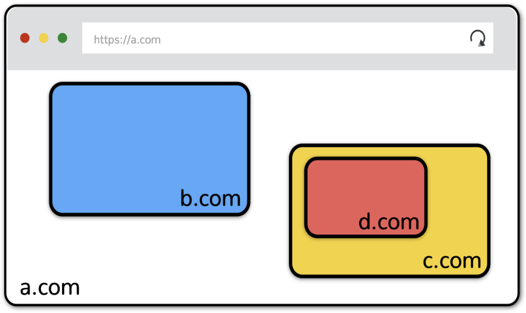


Figure 7: Frame and iFrame

The client browser isolates frames from each other and from the current page, preventing frames from accessing each other's data if it violates the same origin policy. At the same time, if a frame stopped working, the current/parent page still runs.

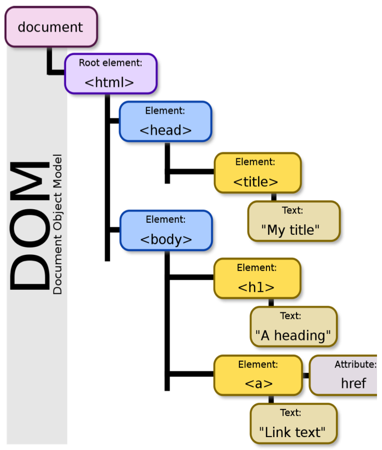


Figure 8: DOM

The DOM provides an object oriented interface for reading and editing website content by ordering page elements into a hierarchy structure. It allows JavaScript to access page data, such as window, document, history, cookies, etc., and modify HTML element on the page.

3 Understanding Relevant Attacker Models

When it comes to attacker models on the web, there are a couple different ones. First we will start with the Network Attacker. The Network Attacker is sitting on the wire between the victim browser and the victim host and can read and modify traffic between the browser and the host.

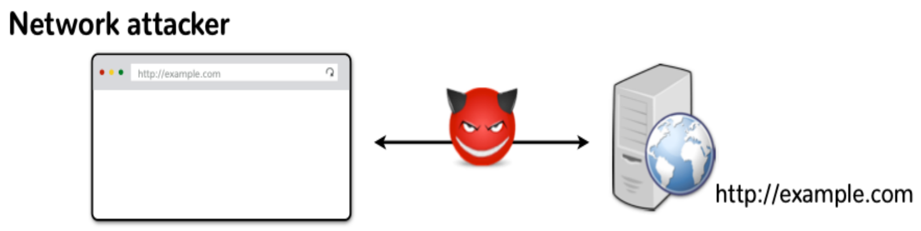


Figure 9: Network Attacker Model

A Web Attacker is a malicious website that is rendered when the user visits the malicious website with their browser. In this case, there is another server that is running some malicious website that wants to violate some web security properties on a target. The target could be the display of another page, a compromised page, victim information, etc.

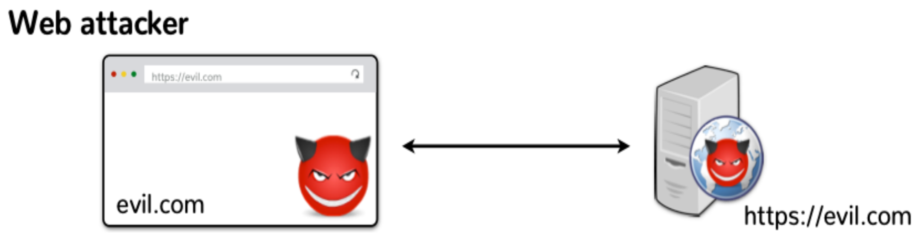


Figure 10: Web Attacker Model

A Gadget Attacker is a sub type of Web Attacker. It is a Web Attacker who is able to inject limited content into an honest page. The honest page could simply be loading an iframe, image, HTML, JavaScript, etc. from the attacker. An example of an innocuous page loading an iframe that contains a malicious website are ads since the innocuous page has no control over the content the ad loads.

When it comes to Web Attackers, there exist other variants to the attacker model described above. For example, we want a user to be able to visit both an innocuous site and an evil site in separate tabs and for the evil site to not be able to do anything the innocuous site.

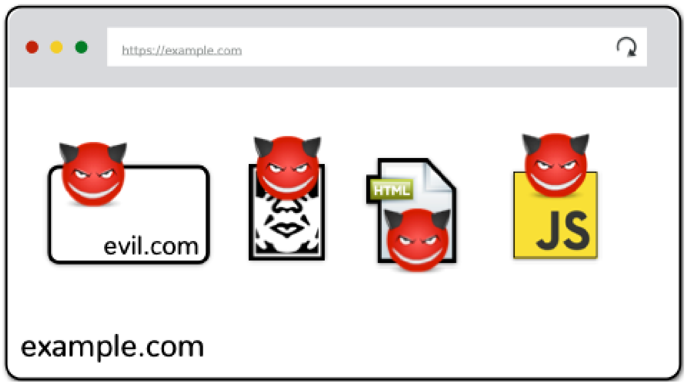


Figure 11: Gadget Attacker Model

Figure 12: Separate Tabs Variant

Another variant is when a site loads a sub resource from an evil site. In this case, we would like for the evil site to not be able to do anything malicious to the innocuous site. A third variant is when an evil site loads content from an innocuous site. In this case, we would like for the evil site to not be able to do malicious things to the innocuous site. Here is an example scenario of this variant in an ideal web model. Lets let the evil site be *evilbank.com*, and the evil site is loading an iframe of the innocuous site *bank.com*. Then, there is some evil script running on the evil site that steals the information that the user enters into the legitimate *bank.com* site and gets their password.

One counter measure against the scenarios above is the browser security mechanism called the "Browser Same Origin Policy."

4 Same Origin Policy

The purpose of the any web security model is ensure we can safely browse the web even in the presence of attackers. If we visit two websites, one good and one malicious, then the malicious website shouldn't be able to compromise or

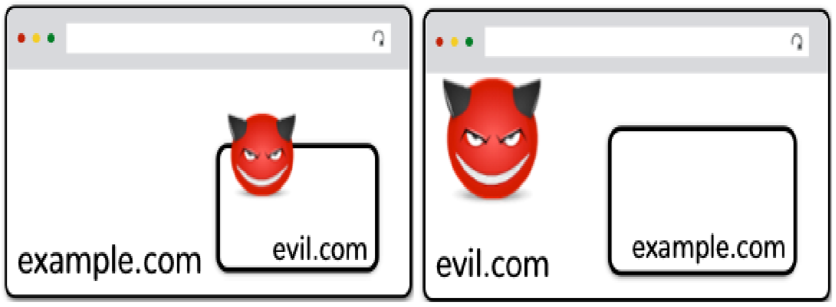


Figure 13: (Left) Malicious iframe Variant, (Right) Malicious Browser Variant

access information from the other website. The same origin policy (SOP) works similarly to how operating systems protect separate processes from affecting each other (virtual machine separation, user id resource allocation, sandboxing).

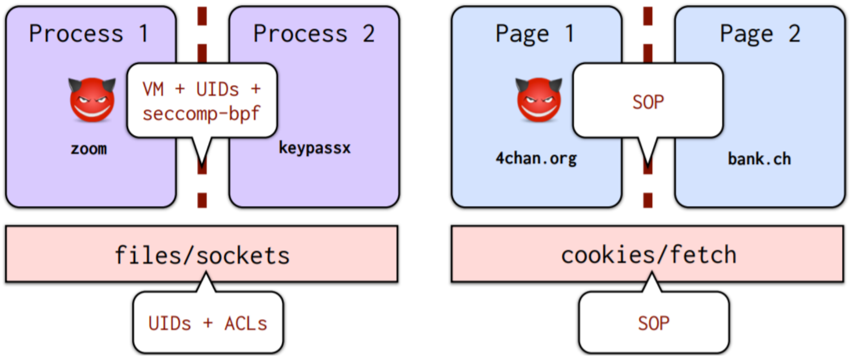


Figure 14: OS and SOP comparison

SOP works by enforcing isolation on the content of different origins. An origin is the scheme, domain, and port. One example where the SOP is used is when a website loads resources from two origins. If one sub resource is malicious, then it will not be able to access or modify content and scripts from the other resource. For each type of resource (DOM, message passing, cookies, etc), there is a different SOP.

4.1 DOM SOP

For the DOM, each frame in a window has its own origin. The frame can only access data if it has the same origin. Data for the DOM includes the DOM tree, local storage, cookies, etc. In Figure 7, the two frames of (https, a.com, 443) can share data since they have the same origin, but evil.ch cannot access the data of either frame.

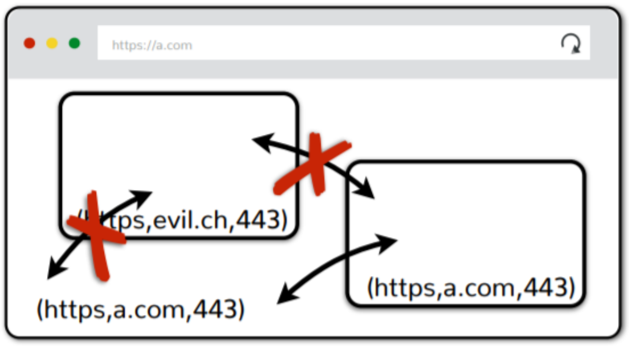


Figure 15: Frame separation SOP

However, frames sometimes do need to communicate with each other. One way this happens is through message passing. This messaging framework allows for the target applications to safely control what messages come across the trust boundary instead of letting scripts access anything they want.

Sender Code

```
targetWindow.postMessage(message, targetOrigin);
```

Receiver Code

```
function receiveMessage(event){
 if (event.origin !== "http://example.com")
 return;
  ...
}
```

4.2 HTTP responses SOP

For HTTP requests, pages can request data across origins and load resources (documents, images, etc) from other origins. The SOP policy does not prevent a page sharing data to another origin in the URL, request body, etc; but does prevent it from directly inspecting the responses through code. Metadata about the response is also visible by the sender.

5 SOP Across Frames, Scripts, Images and Cookies

There is a constant trade off between usability and security. The web would not be very usable if there was a pure focus on security, as certain elements need to be rendered ahead of time. Below are some descriptions regarding how SOP applies to different elements of the web.

5.1 Frames

The iframe tag is a common HTML tag that is utilized by many websites. This tag embeds other documents within the current HTML document. While the web permits cross-origin HTML, it prevents users from modifying or inspecting the frame content.

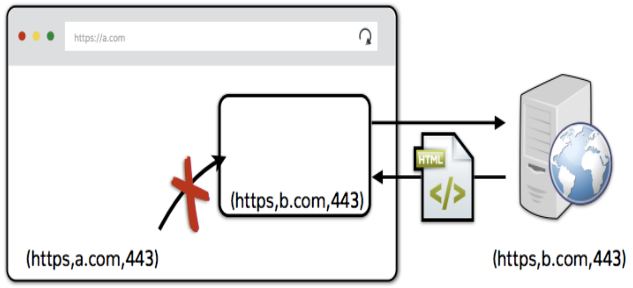


Figure 16: View Only Restriction

5.2 Scripts

Scripts are commonly loaded across different origins, including remote scripts or local scripts. Scripts are essential for accessing certain libraries or code that can help achieve desired functionality. Something to consider is that the script has the same privileges as the page that it is utilized in. Before importing a script, make sure that the script is trusted, and that you are familiar with the origin. The source can be viewed using the toString() method.

5.3 Images

Images are essential for the web. Without them, the web would be very dry and boring. Although images are commonly rendered across different origins, the SOP prevents individual pixels from being inspected. This can prevent images from being reconstructed and prevents sensitive information from being leaked. However, certain information can still be obtained, such as the dimensions of the image, and the specified location. This is why it may be helpful to keep dimensions similar to leak as little information as possible. Figure 17 reveals how a web-page can render different images based on certain criteria, such as the user login status. The figure below reveals a potential attack based on the image width.

The same principle applies to how one styles the page. Cascading Style Sheets, which are used to make pages responsive and presentable, can leak certain information if the page changes according to the device. One example is the font of the page. The line spacing, height, and width can be determined, similar to the discussion on images.

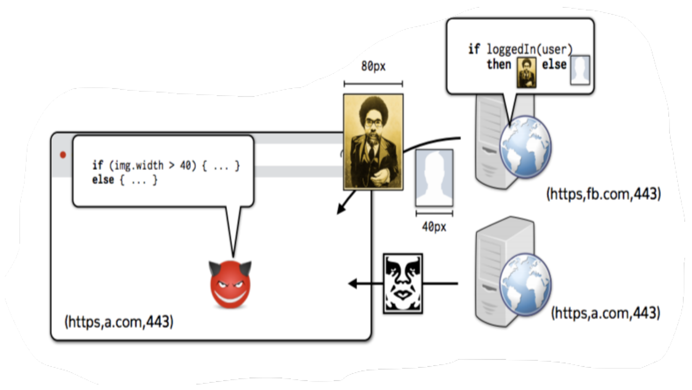


Figure 17: Image Attack

5.4 Cookies

Cookies are a small piece of data that is generated by the server. If a web page is visited again, the user's browser will send the cookie to the server, so long as the cookie is within scope of that web page. Cookies are used by the server to help identify a specific user, and it helps match a user with specific information that the server stores in a database, such as whether the user is logged in. If an attacker learns a user's cookie for a web site, the attacker could impersonate that logged-in user to the web site. For a bank web site, this might allow the attacker to initiate bank transfers; for a webmail client, the attacker could read the victim's email and send mail as them; for a social media web site the attacker would be able to see private information or impersonate the victim to others.

Cookies use a separate definition of origin. Rather than utilizing the port, which is utilized by the DOM SOP, cookies specify the path. The structure can be broken down as [scheme, domain, path]. An example would be (https, ceng.ucsd.edu, /department/cse/classes/cse127).

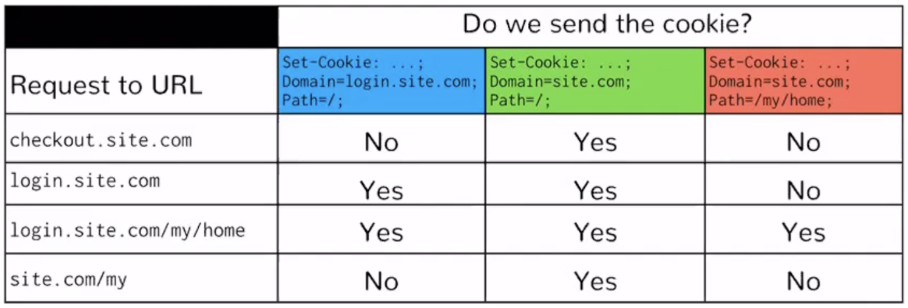
In terms of scope, a page can set a cookie for its own domain, or any parent domain. Pages are also restricted from reading cookies from outside sources, and can only read from its own domain or sub-domain.

The public suffix list, maintained by Mozilla, is a list of top level domain name suffixes that are compiled by volunteers. Some well known examples are .com and .org. The goal of the public suffix list is to reduce the scope of third party cookies. The public suffix list prevents sub-domains in control of third parties from setting cookies on top level domains. Considering the rules of scope above, the list prevents cookies from being set on top level domains and domain names that allow third parties to use subdomains.

The browser determines if a cookie is sent. Historically, browsers sent cookies if they were in the same scope of the url. The cookies domain is defined by a domain suffix of the URLs domain, and the path is defined by a prefix of the URL path. However, this decision can now be controlled by a flag on the cookie called SameSite. If SameSite is set to false, it would adhere to historical convention.

6 Examples of Browsers using Cookies

As mentioned above, browsers send cookies if they are in the same scope of the URL. To illustrate how a browser determines whether or not to send cookies, we use the below chart (Figure 18).



	Do we send the cookie?		
Request to URL	`Set-Cookie: ...; Domain=login.site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/;`	`Set-Cookie: ...; Domain=site.com; Path=/my/home;`
`checkout.site.com`	No	Yes	No
`login.site.com`	Yes	Yes	No
`login.site.com/my/home`	Yes	Yes	Yes
`site.com/my`	No	Yes	No

Figure 18: Example Browser Cookies

In Figure 18, the columns represent potential cookies that the browser can send based on the URLs in each row. For example, when accessing the URL `checkout.site.com`, the cookie with the domain `login.site.com` and path "/" is not sent because the cookies domain is not a suffix of the URL's domain. On the other hand, when accessing the same URL, `checkout.site.com`, the cookie with the domain `site.com` and path "/" is sent because the cookie's domain *is* a suffix of the URLs domain.

In another example from Figure 18, when accessing the URL `site.com/my`, the cookie with the domain `site.com` and path "/" is sent but the cookie with the domain `site.com` and path "/my/home" is not sent. This is because the browser only sends cookies whose paths are a prefix of the URL's path, and "/my/home" is not a prefix of "/my".

These examples describe the browser's policy for sending cookies. If the cookie's domain is a suffix of the URL's domain, and the cookie's path is a prefix of the URL's path, then it is sent.

7 Isolation with Cookies

While the Cookie SOP can prevent malicious sites from accessing cookies they shouldn't be, the DOM SOP still exists and allows malicious sites to access cookies through the DOM. For example, while the Cookie SOP prevents site.com/foo from seeing cookies from site.com/bar and vice versa, the two sites are still able to access the DOM of each other.

One example of accessing the cookie

```
const iframe = document.createElement("iframe");
iframe.src = "https://site.com/foo";
document.body.appendChild(iframe);

alert(iframe.contentWindow.document.cookie);
```

JavaScript scripts originating from sites other than the original site are also able to access cookies as they run with *the origin's privileges.* In other words, they are also able to access `document.cookie`. For example, if your bank includes Google Analytics JavaScript, the script will be able to access your bank's authentication cookie.

One solution to this is to use `HttpOnly` cookie. These are cookies that are only sent, so they are note exposed to JavaScript through `document.cookie`.

Another type of cookie is the `SameSite` cookie. There are three different policies for SameSite cookies: Strict, Lax, and None. As the name suggest, a strict policy means that cookies are only sent when the request is from the same site. A lax policy means that cookies are able to be sent via top-level safe "navigations" (meaning that links are still allowed as they are "safe", the cookie is only sent if the user clicks on the link). The none policy will always send a cookie, and doesn't account for any sort of context. According to historical cookie scope rules, this can result in possible attacks such as Cross-site request forgery (CRSF) attacks.

`Secure` cookies are yet another type of cookie. These are only sent through to the server with an encrypted request using HTTPS to keep network attackers from being able to view cookies in transit. If the server allows unencrypted HTTP traffic, a network attacker can directly steal cookies as shown in *Fig. 20.* A web attacker can also make a cross origin request and steal the cookie from the unencrypted request as shown in *Fig. 19.*

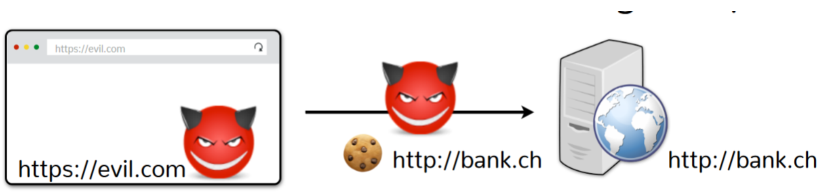


Figure 19: Web attack making cross-origin request

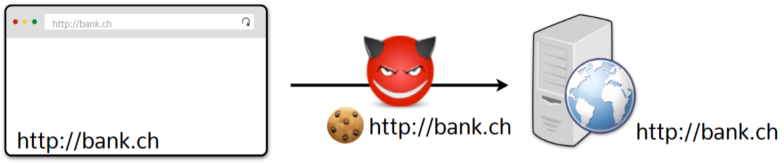


Figure 20: Network attacker stealing cookies on unencrypted HTTP traffic