Chapter 20
Web Concepts

Revision 3.0, December 2025.

Objectives

- Review web basics pertinent to security challenges, including basic types of web requests, HTTP, HTML, cookies, and the Same Origin Policy.

20.1 Web Frameworks

The original World Wide Web was invented at the CERN physics research center in France and Switzerland in 1989 by Tim Berners-Lee, initially as a way to organize and access scientific data. He realized that this the concept of hypertext – documents that not only contain information but also links to other documents – had a much broader applicability than scientific data. This idea resulted in three important developments:

1. URL: The way to name and find data (web pages).
2. HTML: A universal way to format documents that anyone could produce and understand.
3. HTTP: A standard protocol so that any web client could talk to any web server.

Note that security was not a design issue for the original Web. As a result, we have been playing a continuous game of catch-up with security issues as the new uses of the Web outrace the security features.

Modern web applications are unrecognizably advanced when compared to the original World Wide Web, and the evolving security architecture and infrastructure are quite intricate yet increasingly robust. In this chapter, we introduce some of the key security challenges necessary to build secure web solutions. Since the Web is a huge topic and web security is similarly large, this chapter can provide only the basics. It is easy to spend an entire career specializing on just web security.

Note that most modern web applications are built using one of the many Web software frameworks. These frameworks hide many of the details of how the Web works and try to prevent many (but certainly not all) of the common security mistakes that a web programmer can make. It is still important to understand to understand the underlying mechanisms and security issues related to the Web to understand what the frameworks are doing and to know how to stay away from common security mistakes.

20.2 Web Architecture

The Web is fundamentally a client/server architecture, and the core security challenge is that the trust relationship between these two endpoints is complex.

When you browse an unfamiliar website you never know what to expect, who is controlling the server, and whether you can trust them. Likewise, from the perspective of a legitimate web server, when receiving a request, you have no way of knowing who sent it and have no control of the contents of the request.

The web client/server relationship sits atop a complex infrastructure that includes internet service providers (ISPs), backbone communication providers, the domain name system (DNS), and other protocols for routing and delivering messages. While these components typically operate behind the scenes, it is possible that compromise of any of these systems may also threaten the security of the web.

20.3 Web Requests

We begin by considering the fundamental function and structure of a web request. The most basic web operation is when the client (typically a web browser, also known as a *user agent*) requests a resource (typically a web page) from the server (also known as the *web application*). Consider making the following web request:

```
http://goodsite.com/hello.htm
```

The web client would translate that request into HTTP, generating some like what we see in Figure 1.

```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0
Host: goodsite.com
Accept-Language: en-us
```

Figure 1: Basic HTTP Web Request for a `GET`.

The first line of the request describes the request to be made. It includes the operation that is requested, the method (`GET`); the location on the server of the requested resource (`/hello.htm`); and protocol that is being used (HTTP version 1.1). The `GET` method requests the server to send back the contents of the requested resource (web page).

Subsequent lines provide options that detail the request further:

`User-Agent` describes some specifics about the client sending the request, intended to allow the server to potentially customize how it responds.

`Host` provides the domain name of the web server for which the request is intended.

`Accept-Language` indicates the language and locale that this client prefers.

Web requests send a sequence of lines where the first line is always required, followed by option lines beginning with a standard parameter name, colon, and a value for the named parameter. The example above is a simple one using common options, but there are many more available including an extension mechanism where names beginning "`X-`" may be invented for custom use.

Figure 2 shows the response sent back from the server for the request from Figure 1.

The first line of the response consists of the protocol version and the most important part, the status code (`200 OK`). In general, codes in the 200 range mean that the operation completed successfully. The following lines are headers describing the response in a format similar to the request as described above. An empty line separates headers from the body of the response, which is the contents of the requested resource, in this case a web page written in HTML.

```
HTTP/1.1 200 OK
Date: Tue, 15 Oct 2019 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sun, 13 Oct 2019 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<!DOCTYPE html>
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Figure 2: HTTP Response to Basic GET Request

The headers in Figure 2 provide metadata about the response and its contents:

`Date` is the timestamp indicating when the server produced this response.

`Server` is a description of the web server software and its operating system.

`Last-Modified` is the timestamp of when the resource was last modified.

`Content-Length` is the length of the content following in bytes.

`Content-Type` is a description of the content representation, for example, HTML or text.

`Connection` is the status of the connection after the request (when the network connection stays open for subsequent requests or is closed after the current one).

20.4 HTTP vs. HTTPS

HTTP is the underlying protocol for the web, used for transmitting all data between the client and the server. As it was invented before security was a consideration for the Web, it did not include any form of encryption or authentication.

For sensitive web applications such as online shopping or banking, or really any important application including privacy considerations, the lack of encryption and authentication is clearly risky since internet connections travel through multiple hosts and routers (including wireless), housed at locations of which you have no knowledge.

This lack of security in HTTP was recognized early in the development of the web as it became clear that many important applications would be possible if only connections could be secured. In 1994, the developers of Netscape (an early and important web browser) recognized the need for secure web browsing. As a result, they created a secure and encrypted form of HTTP, called HTTPS. They also created SSL (now called TSL, Transport Security Layer[1]), a protocol to conduct encrypted communication over network sockets. In 2000, HTTPS was adopted as an internet standard protocol based on RFC 2818[2].

HTTPS security is based on digital certificates provided by a web server. The servers obtain their digital certificates from certificate authorities who serve to vouch for their identity. These certificate provides the evidence needed for a web server to prove its identity to any client.

---

[1] `https://en.wikipedia.org/wiki/Transport_Layer_Security`
[2] `https://datatracker.ietf.org/doc/html/rfc2818`

Note that HTTPS only authenticates the server to the client. If the client needs to authenticate itself to the server then some sort of logon process has to take place.

Think of HTTPS as a security layer on top of HTTP: the same requests and responses happens, but they are communicated inside of an encrypted channel. To understand what assurances that secure channel provide, in Figure 3, we contrast the risks of using HTTP with the corresponding assurances provided by HTTPS.

Threat	HTTP Risk	HTTPS Assurance
Diversion	The web request is somehow diverted to a different web server than intended.	Only the intended web server is able to decrypt and see the web request details.
Snooping	Someone along the route on the internet data in the request or response is snooped.	Everything is securely encrypted, so nothing is learned from the data on the wire.
Tampering	The web request or response is tampered on the wire and some data is changed.	If the client/server traffic is tampered with, the receiving party can tell and ignore it.
Impersonation	A malicious web server impersonates the real one and sends a bogus response.	The client can check if the intended web server sent the encrypted response or not.

Figure 3: Comparing Risks Associated with HTTP with the Assurances Provided by HTTPS

It is important to understand that the HTTPS secure channel provides important guarantees, but that it is not perfect. Everything hinges on the trustworthiness of digital certificates, including choice of certificate authorities deserving your trust (which is difficult to assess, and most applications depend on defaults chosen by the OS or browser makers).

20.5 HTTP Methods

In this section, we discuss a few of the key and widely used HTTP methods, `GET`, `HEAD`, `POST`, and `PUT`. In Chapter 24 on Cross Origin Resource Sharing (CORS), we will also discuss the `OPTIONS` method. There are four more less frequently used methods listed in the HTTP protocol standard.

A `GET` request tells the web server to send back the contents of the resource (typically a web page) named by the accompanying URL. Note that a `GET` request should not change the state of the server. That is, the implementation on the serve for this method, should never include side effects that modify the state of the resource. The `HEAD` method works similarly to `GET`, but only returns the headers for respond, not the actual contents of the resource.

The full power of the web is only achieved when it becomes possible for client requests to make state changes on the server. For example, with online banking you want to be able to transfer funds between accounts, changing the balances in each account. The two most common methods to use for web requests that will change that state of the server are `POST` and `PUT`.

`POST` requests have headers similar to those used with `GET` requests do, plus a request body that includes the data from the client that describes the state change. The response to a `POST` response is typically a web page that reflects the update. For example, with a web form, the response might either be a message indicating that the request was accepted and change made, or an error response explaining why not. Each time a POST request is made, the state of the server is change. So, if you click on the "purchase" button on a web site, you are likely to end up with two copies of the item that you purchased. This is why you often see websites display a message such as "Purchase in progress. Do not reload the page or click on 'purchase' again."

`PUT` requests are similar to `POST`s in that they change the server state. The big difference is that a `PUT` request should be *idempotent*. Idempotent is a mathematical term that is often means that doing an operation once is equivalent to doing it one or more times. A simple way to visualize the difference is that the programming statement "`x = 1`" is idempotent, but "`x = x+1`" is not. If you repeat the first statement many times the result is still 1, but if you repeat the second statement, the variable value will count the number of repetitions.

The subtle difference that this term expresses is that if you repeat the same `PUT` over and over, it results in the same server state. By contrast, a repeated `POST` operation can incur changes each time.

It is important that it is up to you to understand and respect the semantics of these verbs. When you implement server code for a `GET` operation, generally speaking, no framework is going to stop you from writing code that changes the state of the server. Similarly, when you implement a `PUT` operation, the framework is not going to enforce that it is idempotent. While it is possible to safely get away from bending the rules a bit, it is risky breaking convention since software you do not control (for example, a web proxy) might do something you do not expect and cause problems.

20.6 Cookies

Cookie is a cute term for small pieces of data that a web server returns to the client along with the response page. When you make a subsequent request to that same server, the cookies that you previously received are automatically sent to the server along with the request. A cookie is a (name, value) pair,

where the server assigns both the name and value for the cookie.

In Figure 4, a client makes a request to their bank, and along with the request are sent a couple of cookies (the white ones). Contained in the bank's response is a new cookie (the brown one) that contains, for example, the ID for the current banking session. When the client makes its next request to the bank, that request will contain all the cookies for that site, including the newly sent one.

Cookies are intended to maintain state in the client about what it is doing on the server. So, they might describe your session name (more about that in Chapter 23), whether you clicked on some permission form, or what page you last viewed on this website. Cookies are also (in)famously used to track users across many websites.

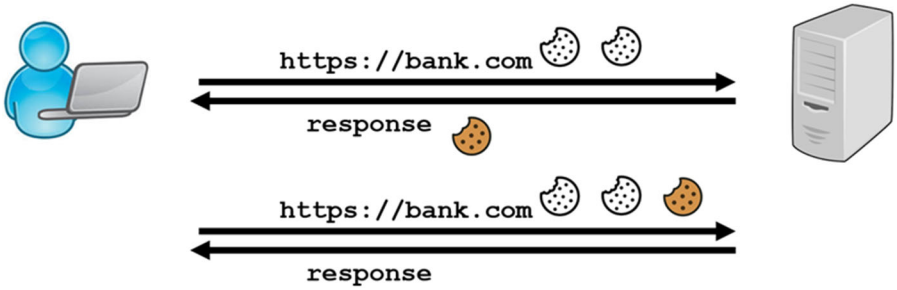

Figure 4: Web Request Sent with Two Cookies and Response with an Additional One

20.7 Same Origin Policy

A critical part of web security is based on a property called the Same Original Policy. This policy tries to prevent a web page from one website from interfering with web pages from another site.

Suppose you had a couple of web pages open in your browser, one from the University of Wisconsin Computer Sciences Department, `cs.wisc.edu`, and one from the City of Madison, `cityofmadison.com`. As we can see from the URLs in Figure 5, they have different domain names for their websites. The browser stores a bunch of information for each of these windows in something called the DOM, the Document Object Model. This information is available to scripts (in other words, JavaScript) running in a window.

There is a long list of things stored in the DOM, but a couple of the most interesting and well known items are the list of cookies for that website (`document.cookie`) and the list of images displayed on that web page (`document.images`).

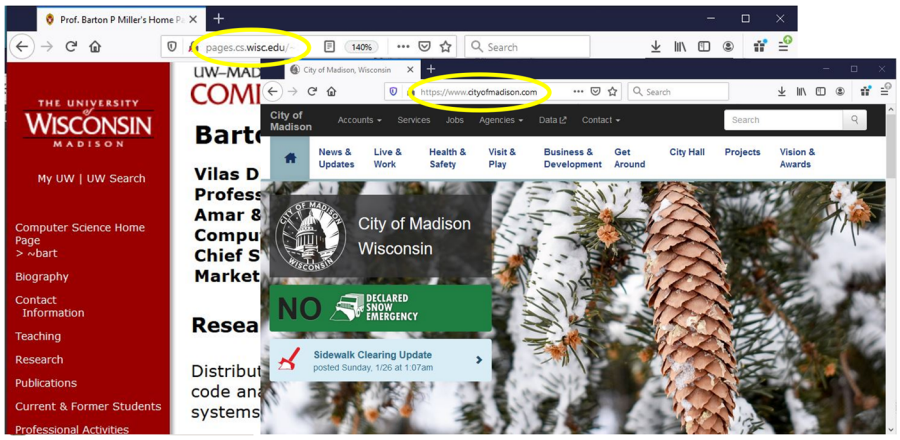

Figure 5: Two Web Pages, One from `cs.wisc.edu` and one from `cityofmadison.com`

The same origin policy controls which of this DOM information can be accessed by a script running in a particular window. That is, a script in a window can access information associated with only its own window or other windows that have the same domain name.

So, in Figure 6, scripts running in the left and center windows can share DOM information and, in fact, share the same collection of cookies. The window on the right cannot access any of the information from the other two windows.

A subtle aspect of this mechanism that is critical to bear in mind or you can subvert the security of your own website. Notwithstanding the protection that Same Origin Policy provides, any webpage is free to include resources for other websites without restriction. That is, you are protected from unknown websites in other windows accessing your page, but if you inject content from other websites by choice then you must trust them and be prepared for the consequences. Common examples of mixing resources from different websites include images, JavaScript, stylesheets, and web fonts. We will see an example of this problem in the next chapter, where we discuss Cross Site Scripting (XSS) attacks.

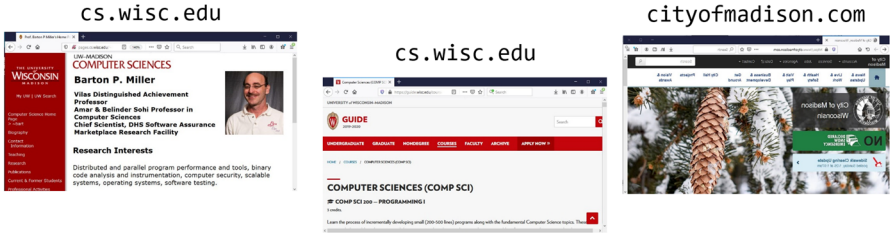

Figure 6: Three Web Pages, Two from `cs.wisc.edu` and one from `cityofmadison.com`

20.8 Summary

In this chapter we covered basic concepts and mechanisms related to the web. We described the core client/server model and the trust challenges it entails, then walked through a basic HTTP request and response, the security protections that HTTPS provides, and the commonly used request verbs (`GET`, `PUT` and `POST`). In addition, we looked at cookies and the Same Origin Policy, including their security properties and the risks they incur.

In the following chapters we will build on these concepts and mechanisms to describe common web vulnerabilities and how to defend against them. Since the web is always evolving and various browsers and web servers tend to provide subtly different implementations, it is important to stay current for the latest most accurate information about all the details.

20.9 Exercises

CAUTION: Ideally you should use a test web server (localhost) for experiments to avoid doing anything to a production website that might look suspicious or unintentionally be harmful.

1. There is an official website for learning about web basics called `www.example.com`. Use one or more tools to explore the inner workings of the web request and response. For example: use your browser's (developer mode) inspect feature; a command line tool such as `curl`[3]; a network monitor tool such as Wireshark[4].
2. Find a relatively simple website and see how it sets cookies (using the same kind of tools as above). Note that the quantity of cookie data in a large complex site could be overwhelming.
3. Configure or write a simple localhost web server to explore Same Origin Policy. Using two instances with different localhost port numbers you can see how different origin hosts are blocked by the browser.

[3] `https://curl.se/`
[4] `https://www.wireshark.org/`