Introduction to Software Security

Chapter 3.9.1: Web Concepts

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 0.1, November 2022.

Objectives

- Review web basics pertinent to security challenges.
- Understand cross site attacks and how to prevent them.
- Understand the basics of session management and potential threats to its integrity.
- Understand the risks related to open direction and how to prevent abuse.

Web frameworks

Modern web applications are unrecognizably advanced by comparison to the original World Wide Web, and the evolved security architecture and infrastructure is quite intricate yet increasingly robust. This chapter is designed to introduce you to some of the key architectural and practical 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. Further reading is needed to learn more about the details. .

The safest and easiest way to address the various threats inherent to the web platform is to rely on a well built framework that handles the details, allowing you to develop a web application at a higher level of abstraction. Yet even then, these web security concepts are important to understand, if only to appreciate what security protections the framework is providing and how they work. This knowledge is especially important whenever you "go outside the lines" and override mechanisms of the framework on which security crucially depends. And of course framework development work requires a deep understanding of these concepts and many details beyond scope here.

Web Architecture

Web browsing is a familiar part of our daily activities, even to the most non-technically oriented of modern digital societies, yet the underlying mechanism is rarely considered by its users. The web is fundamentally a client/server architecture, and the core security challenge is that the trust relationship between the 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 or not. Likewise, from the perspective of a legitimate web server, when receiving a request you have no way of knowing who sent it, friend or foe, and have no control of the software they are using.

On top of the confusing client/server relationship, we must also consider that a complex infrastructure exists to facilitate connections over the Internet. Typical web interactions involve several internet service providers, backbone communications, the domain name system, and other protocols for routing and other technical aspects of internet connectivity. While these parties typically operate behind the scenes, it's possible that compromise of any of these systems may also threaten the security of the web.

The details of how all of this is secured are beyond the scope of this chapter (and the specifics evolve over time as the internet matures and grows), but the basic security stance is clear. Web clients and servers cannot in general trust each other, so must build trust via interactions that form a relationship. Furthermore, even with some degree of trust, it is important to always maintain vigilance since the underlying internet communication channel itself may be subject to various attacks. Keep these practical considerations in mind as we begin to see how the web works with an eye to security.

Web Requests

We begin by considering the fundamental function and structure of web requests. 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. Here is the core content of such a request:

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

The first line of the request declares the *verb* (GET), *path* (`/hello.htm`), and *protocol* (HTTP version 1.1). The GET *verb* requests the server to send back the contents of the resource requested by naming its *path*, and 1.1 is the modern version of HTTP (HyperText Transfer Protocol) that defines how the web operates over the internet.

Subsequent lines detail the request further:

- The `User-Agent` line describes some specifics about the client sending the request, intended to allow the server to potentially customize how it responds.
- The `Host` line provides the domain name of the web server the request is intended for.
- The `Accept-Language` indicates the natural language and locale that this client prefers.

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

Recall the precautionary explanation of trust on the web to think about the interpretation of web requests. Since the web server has no pre-existing relationship with a new client, it must process web requests with caution. Possibly the client is lying about its identity — it might claim to be Mozilla but actually be another browser, and in fact for purposes of compatibility this is commonplace and generally not considered malicious. The requested path may not even exist, or might be an excessively long string attempting to break the server maliciously. On this tenuous basis the web server replies to the client which, of course, cannot always fully trust the server either.

Continuing this example, here is the response sent back from the server to the requesting client:

```
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>
```

The first line of the response is required, consisting of the protocol version, a *status code* (`200`) and a human-readable explanation of that code (`OK`). 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.

In our example, the headers here provide metadata about the response and its contents.

- `Date:` the timestamp indicating when the web server produced this response
- `Server:` a description of the web server software and its operating system
- `Last-Modified:` the timestamp of when the resource was last modified
- `Content-Length:` the length of the content following in bytes
- `Content-Type:` a description of the content representation (for example, HTML or text)
- `Connection:` the status of the connection

The combination of the request and response example above is how a web browser works with the corresponding web server when you type `http://goodsite.com/hello.htm` in the address bar.

For a web browser to securely process such a response from a web server that it cannot trust, it must cautiously parse and process all parts of the response. Even for this simple example there are many possibilities for a malicious web server to fool the client, and even without bad intentions, a buggy web server can do damage as well. Here are a few examples of potential problems (and you can think about many more as well):

- The web server might misidentify itself.

- The web server could report any time in the past or future, or even an invalid timed syntax.
- The web server might provide a misleading content length byte count, accompanied by more or less data than expected.
- The response body might be invalid data for the content claim it claims to be.

Modern web applications are built from an intricate conglomeration of web requests and responses like this that together constitute web pages that include not just display content but also script, styling, web fonts, images, and many other types of data. Since trust is always conditional on the web, browsers and servers must carefully interpret the data they exchange as requests and responses, in order to maintain integrity and avoid compromise. As a general rule, when anything untoward about a request or response is detected the recipient usually ignores suspect data so as to avoid falling into traps.

HTTP vs. HTTPS

HTTP is the original transfer protocol of the web and it sends all data between the client and the server in the open, without encryption or authentication. While the technology underlying the internet is far more complex, this basic form of communication is often referred to as "on the wire" because it is subject to snooping or modification anywhere between the two endpoints. For sensitive web applications such as online shopping or banking, or really any important application including privacy considerations, this is clearly risky since internet connections are complicated and depend on numerous parties typically unknown to the communicants.

This weakness 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. To address this threat, a new protocol was designed with just such security in mind, called HTTPS. The communication protocol itself is called TLS (Transport Layer Security), or formerly, SSL.

Web servers obtain digital certificates from certificate authorities who serve to vouch for their identity. The details are complex, including cryptography as well as subtle legal considerations, but in essence a certificate provides evidence for a web server to prove its identity to any client.

Think of HTTPS as a security layer on top of HTTP: the same request and response exchange happens, but it's all communicated inside of a secure channel. To understand exactly what assurance that secure channel provides — assuming that the certificate authority did their job and the web server itself was not compromised — we can contrast it to some of the risks of plain old HTTP:

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

It's important to understand that the HTTPS secure channel provides important guarantees, but that it isn't 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 OS or browser makers). Security always depends on the quality of the software implementing the security, and the full stack from OS to application on both client and server need to perform correctly to get everything right. Furthermore, note that data tampering on the wire is detectable but such interference on the wire can effectively block communication.

HTTP Verbs

The simple example that opened this chapter was a GET request; now let's consider the meaning and use of other verbs in the HTTP protocol that enhance its richness.

To review, a GET request tells the web server to send back the contents of the resource named by the accompanying URL, that is, the path of the request. An important detail of what this verb means is that this operation does not change the start of the server: that is, the request should never include side effects that potentially modify the state of the resources that the web server maintains.

A lot of web browsing consists entirely of GET requests, or "just looking": reading social media, a blog post or recipe, wiki entry, doing a web search, browsing a photo gallery, and so forth are examples of this kind of access. It's important to note that web servers do log user activity and perform other housekeeping activity that does result in some state on the server changing, however, since data such as system logs is not available for web browsing this doesn't count. The kind of state change that a GET could potentially violate would be for a client to GET "A", and then GET "B" where that request had side effects such that it caused a subsequent GET "A" to respond with a different value.

The full power of the web is only achieved when it becomes possible for client requests to sometimes intentionally change state on the server. For example, with online banking you want to be able to transfer funds between accounts, so that when you review your balances the new amounts can be confirmed. In terms of web programming, web forms and file uploads are common ways that client requests can change state on the server, and typically these are sent with the POST verb.

POST requests have headers as GET requests do, including a resource URL path, as well as a body (separated by a blank line) that encodes the data from the client that describes the state change. The 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 (including a message explaining that a required field was missing or an entry invalid).

PUT requests are similar to POSTs in that they potentially change the server state, however they are *idempotent*. The subtle difference that this term expresses is that if you repeat the same PUT over and over, it only results in the same server change (repetitions have no further effect). By contrast, repeated POSTs potentially incur further changes each time. A simple way to visualize the difference is that the programming statement "`x = 1`" is idempotent, but "`x = x+1`" is not: this is because if you repeat the former many times the result is still 1, but for the latter the variable value will count the number of repetitions. (Note that this is why browsers prompt the user if they refresh a web page that is the result of a POST: if you resend an online order form it could result in duplication and you get twice as much as you thought you were buying.)

There are a few other HTTP verbs defined in the protocol but they are rarely used, and as a practical matter the large majority are GET requests with most of the rest being POSTs. Finally, it's important to note that it's up to you to understand and respect the semantics of these verbs: generally speaking, no browser or framework is going to stop code a web server from modifying state when it shouldn't. While it's possible to safely get away from bending the rules a bit, it's very risky breaking convention since software you don't control (for example, a web proxy) might do something you don't expect and cause problems.

Cookies

Cookies are a clever way that the client and server cooperate make web browsing richer, but as always the issue of mutual trust complicates things. Cookies are traditionally small chunks of data issued by the web server that the client stores locally on the server's behalf.

For example, when bank.com responds to a web request it can include a cookie header that specifies one or more cookies be set for the bank.com domain. In processing the response, the client saves the named cookies and their contents in a special store indexed by the site name. Subsequent web requests to the same bank.com website will then be accompanied by these cookies, sent back in the request header, which gives the web server a consistent view of the client. For example, if the user at some point indicated a preference to the web server, by sending it as a cookie the server can have the client store the preference which is then conveniently conveyed in future requests where it can be easily honored. Note that cookies are sent for all requests to the corresponding web server, even from a different window.

Cookies are extremely handy, and as often happens they can be used in unintended ways that may compromise privacy so the precise rules are complicated. Session cookies only last so long as any window or tab is open to the website, or persistent cookies (that may or may not have expiration dates) outlast web sessions. Cookie sizes are subject to storage limitations and can also be scoped by domain name or URL directory structure.

In practice, web servers use cookies to store session IDs, login account information, flags that you have done some action, user preferences, and to track site usage. Cookies are famously used to track users across many websites. One common technique is to include one invisible pixel as an image which causes a (GET) request to the same tracking site from whatever pages it can get in, including a change to read and update its own set of cookies each time. The next section gives an in-depth look at the security consequences of websites that include resources from other websites.

Same Origin Policy

Web browsing routinely moves seamlessly from one site to another, such as when one site (for example, web search) contains hyperlinks to pages on other sites. Behind the scenes, browsers provide various protections to safely isolate one site from another to make this work smoothly. Windows or tabs open for pages of the same website should be able to interact via script and share the same cookies, however there must be protection against other website pages “reaching in” and stealing data or influencing the content inappropriately.

Same Origin Policy is the core concept that defines how these protections work. Each window (or tab) potentially runs script to access the page as well as site information via the Document Object Model (DOM), including access to other open windows. The DOM contains a list of open window objects, and within each window a document object holds details about the specific page. To list a few examples: `document.cookie` enumerates cookie values set on the page; `document.images` describes image tags on the web page.

The way Same Origin Policy protects websites is by restricting the DOM such that only objects on the same page, or into other window objects from the same website, are accessible via script. This allows windows from the same website (such as different parts of cs.wisc.edu) to interact richly while blocking unwanted access such as from evil-site.com.

There is 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, script, stylesheets, web fonts. Web pages can also include frames and inline frames (iframes) where there are special rules about how these interact.

Modern browsers support Content-Security-Policy headers that allow websites to declare security intentions for interaction with other websites they reference, but given the tricky trust model between clients and servers caution is advised. In general, only reference resources from other websites that you absolutely trust, and even then it’s advisable to use other security mechanisms for additional protection.

Summary

In this chapter we covered basic concepts and mechanisms related to security on the web. First, since the web security model is complicated and subtle to get right, we talked about how frameworks that provide reliable security protections are a good way to build secure web solutions. 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’s important to stay current for the latest most accurate information about all the details.

A great starting point to learn more about all these details is Mozilla’s Web technology for developers.

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(1); a network monitor tool such as wireshark.
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.