Chapter 1

Web and Browser Security

1.1 Introduction

Definition 1.1.1 ▸ Domain Name

A ***domain name*** is a human-readable name for an internet service.

For simplicity, we will assume that a domain name can only have one associated service/server, and vice versa (i.e. a one-to-one mapping between domain names and servers).

Definition 1.1.2 ▸ Domain Name System (DNS)

A ***Domain Name System*** maps domain names to their corresponding server's IP address. We say the DNS ***resolves*** a domain name to mean the DNS converts a domain name into an IP address.

It's common for computers to store the IP addresses for one or two DNS services. By default, a DNS has no integrity (that is, an attacker on the network could). DNSSEC solves this issue but has low adoption.

what could they do tho

Definition 1.1.3 ▸ Top-Level Domain (TLD), Second-Level Domain, Subdomain, Fully-Qualified Domain Name (FQDN)

Consider the following domain name:

```
userlab.utk.edu
```

- `edu` constitutes the ***top-level domain (TLD)***
- `utk` constitutes the ***second-level domain***
- `userlab` constitutes the ***subdomain*** which provides further naming specification for the server. It can have several "levels" and is assumed to be controlled by its parent (sub)domain.
- `userlab.utk.edu` consitutes the ***fully-qualified domain name (FQDN)***, which is sent to the DNS to be resolved into an IP address

In addition, a domain name is the combination of a second-level domain and a top-level domain.

Machines typically parse domain names from right to left, starting from the most broad identifiers and working towards the most specific ones.

`www` is a common subdomain which is sort of an artifact of the internet's inception. In the early days, people wanted to separate the website from other services from the domain name. There is no guarantee that `www.website.com` hosts the same data as `website.com`. Now, most websites specify `website.com` to simply redirect to `www.website.com`.

Technique 1.1.4 ▶ Resolving Domain Names

Suppose we have some servers with domain name `amazon.com` and Alice who wants to connect to the servers.

- Alice connects to her intermediate access point that actually connects to the internet (most commonly a router).
- Alice prompts the DNS for the IP to `amazon.com`.
- The DNS resolves the IP address and gives it to Alice.
- Alice connects to the resolved IP address.

Definition 1.1.5 ▶ Uniform Resource Locator (URL)

A ***URL*** identifies specific resources on a server. This contrasts from a domain which only specifies which server to connect to.

Definition 1.1.6 ▸ Scheme, Host, Port, Pathname, Query

Consider the following URL:

`scheme://host[:port]/pathname[?query]`

The square brackets denote optional fields.

- The ***scheme*** is the protocol used to communicate with the server (e.g. http, https, ftp, ftps, ws, etc.).
- The ***host*** is usually the domain name or IP address of the server.
- The ***port*** specifies the port to connect to. If omitted, it uses default values for known schemes (e.g. 80 for http, 443 for https).
- The ***pathname*** identifies a specific resource on the server. It resembles a UNIX file system path, but it does not actually need to map to the underlying file system.
- The ***query*** is addition data that further specifies the requested resource. It always starts with `?` and is a collection of key-value pairs, separated by `&`.

When encoding URLs, there are only a certain set of characters that is allowed. Non-allowed characters are encoded as their hexadecimal representation prepended with a `%`.

Definition 1.1.7 ▸ Hypertext Transfer Protocol (HTTP)

HTTP is a common request-response protocol for serving web content. It is built on TCP (ensures reliable data transfer) and is a stateless protocol (that is, connections are all independent of each other at the protocol level; no data is stored about the request itself).

HTTP was originally designed to only serve Hypertext Markup Language (HTML), but is now used to serve general content.

Definition 1.1.8 ▸ HTTP Methods

HTTP requests are sent similar to the following string:

`METHOD /pathname?query HTTP VER Host:`

`hostname:port` $<$ *keyword* $>:<$ *value* $><$ *requestbody* $>$

format this properly like an actual HTTP request

- ***GET:*** retrieve data from the URL
- ***POST:*** push data to the URL
- Others include HEAD, PUT, DELETE, TRACE, CONNECT, OPTIONS, PATCH
- The ***request body*** is the data needed to fulfill the request.

An HTTP response follows the generic form:

```
HTTP/version status_code <keyword>:<value> <response body>
```

- ***Status code:*** 100s informational, 200s successful, 300s redirection, 400s client error, 500s server error
- ***Response Headers:*** metadata about response in key-value pairs, separated by `\r\n`
- Response body: data returned from the request

HTTP Response

Although the HTTP spec specifies the intended uses of each method, servers can implement and handle these methods however they choose. So there is generally no standardization of these methods between different servers.

Definition 1.1.9 ▸ Hypertext Markup Language (HTML)

HTML is a markup languages based on the Standard Generalized Markup (SGML). An HTML document is composed of elements described using ***markup tags***, following a tree structure where each element can have a parent, sibling, or children elements.

In 1998, SGML was reworked into XML, after the creation of HTML. HTML and XML are very similar, but not the same.

Definition 1.1.10 ▸ HTML Element

```
<tag [attribute='value']>[content]</tag>
```

- Tag: the name of the element, such as `p` for paragraph or `h1` for header.
- Attribute list: each element defines a set of attributes with well-defined behavior; HTML also supports arbitrary attributes but without well-defined behavior (up to your own implementation)

- Content: can be composed of other elements (i.e. children), textual content, or a combination of the two
- Closing tag: Only tags that can have contents need to be closed

Example 1.1.11 ▶ HTML Document

- The `html` tag is the root element
- `head` contains metadata elements
- `body` contains the webpage's contents
- Common content tags are `p` (paragraph), `h1` through `h6` (headers), `ol` (ordered list), `ul` (unordered list), etc.
- HTML 5 now supports ***frames*** which are used to display another entirely different HTML document.

The `script` tag can be used to add client-side scripts to add interactivity to the webpage. These scripts are almost exclusively written in JavaScript.

Definition 1.1.12 ▶ HTTPS

HTTPS is simply the HTTP protocol using TLS.

What happens if multiple domains are served by the same IP address?

- This is referred to as ***name-based virtual hosting***
- For HTTP, the `host` header will identify how to route the request

The problem when usnig HTTPS is to figure out which certificate to provide when establishing TLS with the client. Also, what about routing after TLS is established? The HTTP request is encrypted and the host header will not be reachable.

Definition 1.1.13 ▶ Server Name Indication (SNI)

Server name indication (SNI) is a TLS extension which allows clients to indicate which domain they want to communicate with.

All HTTPS requests in this TLS session will be routed to that domain. The downside here is that SNI clearly reveals who the user is communicating with, and it often breaks many corporate security tools. More recently, people have developed encrypted SNI where the client hello is encrypted using a public key for the domain. This public key is hosted in DNS but requires

DNSSEC to be secure.

Definition 1.1.14 ▶ Document Object Model (DOM)

The ***document object model (DOM)*** is the computer's interpretation of an HTML document which is used to actually render the HTML document. The DOM can be modified by scripts and will be immediately reflected.

The DOM is composed of several things, including:

- The HTML document itself
  - `document, document.head, document.body`
  - `document.location`
  - `document.scripts`
  - `document.getElementById()`
  - `document.querySelector()`, `document.querySelectorAll()`
- The window displaying the document
  - `window`, `window.document`
  - `windodw.alert`
  - `window.cookieStore`
  - `window.localStorage`, `window.sessionStorage`
  - `window.location`
- The console, typically used by developers
- `fetch()`: used to retrieve new content from the server; much of the modern web is based on `fetch()`, starting by sending a bare-bones webpage and fetching content as required

As a developer, never trust client-side code. Clients can arbitrarily access, modify, and injects scripts. Big takeaway: **important data should never be stored on the client.**

Sessions Originally, the web was purely static HTML webpages. As such, each request was independent of one another. At most, the `referer` header was used to link requests. In order to create more fully-featured web applications, web developers have created mechanisms for storing session data to be associated with HTTP requests, allowing HTTP requests to be linked to users.

When linking sessions to HTTP requests, we need to ask ourselves, "what session data should be stored by the server? The client?" Sessions are located by the server using a ***session ID***. This identifier is used to lookup session data for a connection.

How does the session know which session ID to use for each request? HTTP is a stateless protocol. Originally, each HTML document was sent in its own TCP connection. As such, it's not possible to bind the session ID to the connection. While HTTP/1.1+ allows many HTML documents to be sent in a single TCP connection, the stateless nature of the protocol was preserved.

Instead, each HTTP request needs to contain the session ID.

Definition 1.1.15 ▸ Cookies

Cookies refer to data stored on the web client that is used for additional functionality and/or as a means of identifying users. They usually exist between webpages and sessions on the same website.

Cookies are the standardized approach for sending session data with each HTTP request. Cookies are sent by the server in an HTTP response, and all future HTTP requests will automatically include the cookies sent by the server. Cookies can also be created, read, updated, or deleted by client-side scripts.

In an HTTP response, the set-cookie header may look something like:

```
Set-Cookie:sessionID=78ac63ea01ce23ca; Path=/; Domain=mystore.com
Set-Cookie:language=french; Path=/faculties; HttpOnly
```

As we can see, it is composed of name-value pairs separated by semicolons. Optional attributes include `domain`, `path`, `max-age`, `Secure`, `HttpOnly`, `SameSite`.

In an HTTP request, the cookie header may look something like:

```
Cookie: sessionID=78ac63ea01ce23ca; language=french
```

This includes all cookies in a single header. Once again, fields are separated by semicolons. These cookies are only sent if:

1. they are for the correct domain and path,
2. they satisfy the `Secure` and `SameSite` attributes, and
3. they have not expired.

1.2 Same Origin Policy

Definition 1.2.1 ▶ Origin, Same-Origin Policy (SOP)

An ***origin*** refers to the scheme (e.g. HTTP), host (fully-qualified domain name or IP address), and port. The ***same-origin policy*** guarantees that content from one origin will have limited access to content from another origin.

For example, `http://site.com/asdf` and `http://site.com:80/asdfzxcv` have the same origin.

The same-origin policy is widely regarded as the most important security feature that web browsers provide.

For example, same-origin policy is applied to:

- ***Frames:*** scripts in one frame are only allowed access to the DOM of another frame if they share the same origin.
- ***Web requests:*** scripts from one origin are only allowed to make `GET`, `POST`, and `HEAD` requests to another origin, and even then have limited headers and cannot view the response bodies. Some safe headers include `Accept`, `Accept-Language`, and `Content-Language`. Some safe `Content-Type` values include plaintext and form data.

Loosening SOP Protections There are some circumstance to loosen SOP protections. Regardless of the intent, extreme precaution must be taken when doing so.

- `document.domain` can be used to override the fully-qualified domain name in the origin. This allows pages from two distinct subdomains to change their domain to the same parent domain, allowing cross-page domain access. This feature is now deprecated.
- Channel messaging API allows frames to send messages to each other. The sender explicitly identifies the domain(s) they send messages to. The receiver explicitly identifies the domains they are receiving from.
- ***Cross-origin resource sharing (CORS):*** A header in the HTTP response indicates origins that are allowed to access the response body; can also allow additional methods and header options.

1.3 Cookie Attacks

If an attacker can steal a user's session identifier cookie, the attacker can use that cookie to impersonate the user! Methods include:

- Intercepting HTTP traffic; can be stopped by using TLS and/or setting the Secure attribute on cookies.
- Malicious client-side scripts: supply-chain threats make this threat increasingly common; can be prevented by setting the `HttpOnly` attribute on cookies.
- Client-side malware: malware can attack the browser and steal the cookies, either out of memory or on the disk where they are stored.

Let's suppose an attacker can't directly steal the cookies. They might still try to misuse them instead. Recall that whenever an HTTP request is made, cookies are automatically attached to that request. The browser sends these cookies. We can trigger a web request using `fetch()` to request a resource from any origin!

Definition 1.3.1 ▶ Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) refers to a common online attack. The user connects to a malicious website which executes a web request for the legitimate website. The browser will attach the legitimate site's cookies to this request.

Such an attack has the following steps:

1. A user connects to a legitimate website and has a cookie.
2. The user unknowingly visits a malicious website or clicks on a malicious link.
3. The malicious website contains a script that generates a request to the legitimate website.
4. The browser, unaware of the malicious intent of the request, automatically attaches the cookie from the legitimate website to this request.
5. The request is sent to the legitimate website. Because the request includes the user's legitimate session cookie, the website assumes the request is from the authenticated user and processes it.
6. The malicious site thus can perform actions on the legitimate site on behalf of the user, without their knowledge or consent. This could include actions like changing the user's

email address or password, performing transactions, or any other action the user is authorized to perform on the legitimate website.

There are several methods for defense against CSRF attacks:

- Limit what can be done with the `GET`, `POST`, and `HEAD` requests; these are the only requests allowed by SOP, but can be overridden by CORS headers
- Set the `SameSite` attribute on the cookies; will only attach cookies originating from the domain that set the cookie
- Use an anti-CSRF token which itself is a cookie. Whenever the user triggers a sensitive action, include the anti-CSRF token in the request. Then the server only allows the operation if both values match.

Another approach is ***cross-site scripting***. The vulnerability lies in the fact that websites often include user input in their output. This can allow attacks to inject malicious scripts.

Definition 1.3.2 ▸ Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) refers to a method of online attack where an attacker finds a means to get a user to run malicious code by visiting some link or website.

These attacks can result in varying outcomes including stolen cookies, stolen data stored in the browser (e.g. `LocalStorage`), redirecting the browser to an attacker-controlled domain, and/or modifying the document itself.

Reflect XSS:

1. An adversary includes some malicious code or script to an otherwise legitimate link
2. A user clicks on the link, where the malicious script runs while connected to the legitimate website

This requires the user to click an attacker-controlled link. This is not effective if the user directly visits the website. In addition, this only impacts a single user. A common approach use a URL shortener to hide the malicious content from the URL. Also, it's common to load some arbitrary image with `onLoad="malicous code"` as well as some code to reload the page to hide the malicious code from the URL.

Stored XSS:

1. The adversary directly connects to the legitimate website and uploads some malicious

code (perhaps through a POST request)

2. The website stores the adversary-owned script in their database
3. A user connects to the legitimate website and sends a request to get relevant items from a database (perhaps comments on a post/thread)
4. The website sends the adversary-owned script to the user, where it is then run on the user's machine.

This does not require any error on the user's part, and works even if the user directly visits the site. Moreover, this impacts many users. Reflected XSS attacks can be used to set up Stored XSS attacks

To address XSS attacks, we never user functionality that can allow an attacker to inject malicious content. Specifically, we avoid setting the `innerHTML` attribute and avoid calling `eval()` on user input. It's best to rely on frameworks that sanitize and filter malicious content before processing user input.

1.4 SQL Attacks

SQL is a language used to query data in a database. SQL injection attacks happen when user input is directly included inside SQL statements executed on the database. It's similar to XSS where the attacker injects malicious code.

SQL itself stores data in ***relational databases*** which can be better thought of simply as tables. Each table has a ***schema*** that defines the columns or fields that make up the tables. In terms of OOP, tables are like classes and each row is an instantiation of that class.

Missing a lot of notes

Allows for arbitrary access to the SQL interface which means the attacker can do anything they want.