18. Introduction to the Web

18. Introduction to the Web

18.1. URLs

Every resource (webpage, image, PDF, etc.) on the web is identified by a URL (Uniform Resource Locator). URLs are designed to describe exactly where to find a piece of information on the Internet. A basic URL consists of three mandatory parts:

http://www.example.com/index.html

The first mandatory part is the *protocol*, located before in the URL. In the example above, the protocol is http. The protocol tells your browser how to retrieve the resource. In this class, the only two protocols you need to know are HTTP, which we will cover in the next section, and HTTPS, which is a secure version of HTTP using TLS (refer to the networking unit for more details). Other protocols include `git+ssh://`, which fetches a git archive over an encrypted tunnel using `ssh`, or `ftp://`, which uses the old FTP (File Transfer Protocol) to fetch data.

The second mandatory part is the *location*, located after but before the next forward slash in the URL. In the example above, the location is www.example.com. This tells your browser which web server to contact to retrieve the resource.

Optionally, the location may contain an optional *username*, which is followed by an `@` character if present. For example, evanbot@www.example.com is a location with a username `evanbot`. All locations must include a computer identifier. This is usually a domain name such as www.example.com. Sometimes the location will also include a port number, such as www.example.com:81, to distinguish between different applications running on the same web server. We will discuss ports a bit more when we talk about TCP during the networking section.

The third mandatory part is the *path*, located after the first single forward slash in the URL. In the example above, the path is /index.html. The path tells your browser which resource on the web server to request. The web server uses the path to determine which page or resource should be returned to you.

One way to think about paths is to imagine a filesystem on the web server you're contacting. The web server can use the path as a filepath to locate a specific page or resource. The path must at least consist of `/`, which is known as the "root"[1] of the filesystem for the remote web site.

Optionally, there can be a `?` character after the path. This indicates that you are supplying additional arguments in the URL for the web server to process. After the `?` character, you can supply an optional set of *parameters* separated by `&` characters. Each parameter is usually encoded as a key-value pair in the format `key=value`. Your browser sends all this information to the web server when fetching a URL. See the next section for more details on URL parameters.

Finally, there can be an optional *anchor* after the arguments, which starts with a `#` character. The anchor text is not sent to the server, but is available to the web page as it runs in the browser.

The anchor is often used to tell your browser to scroll to a certain part of the webpage when loading it. For example, try loading https://en.wikipedia.org/wiki/Dwinelle_Hall#Floor_plan and https:

[1] It is called the root because the filesystem can be treated as a tree and this is where the tree starts.

//en.wikipedia.org/wiki/Dwinelle_Hall#Construction and note that your browser skips to the section of the article specified in the anchor.

In summary, a URL with all elements present may look like this:

http://evanbot@www.cs161.org:161/whoami?k1=v1&k2=v2#anchor

where http is the protocol, evanbot is the username, www.cs161.org is the computer location (domain), 161 is the port, /whoami is the path, k1=v1&k2=v2 are the URL arguments, and anchor is the anchor.

Further reading: What is a URL?

18.2. HTTP

The protocol that powers the World Wide Web is the Hypertext Transfer Protocol, abbreviated as HTTP. It is the language that clients use to communicate with servers in order to fetch resources and issue other requests. While we will not be able to provide you with a full overview of HTTP, this section is meant to get you familiar with several aspects of the protocol that are important to understanding web security.

18.3. HTTP: The Request-Response Model

Fundamentally, HTTP follows a request-response model, where clients (such as browsers) must actively start a connection to the server and issue a request, which the server then responds to. This request can be something like "Send me a webpage" or "Change the password for my user account to `foobar`." In the first example, the server might respond with the contents of the web page, and in the second example, the response might be something as simple as "Okay, I've changed your password." The exact structure of these requests will be covered in further detail in the next couple sections.

The original version of HTTP, HTTP 1.1, is a text-based protocol, where each HTTP request and response contains a *header* with some metadata about the request or response and a *payload* with the actual contents of the request or response. HTTP2, a more recent version of HTTP, is a binary-encoded protocol for efficiency, but the same concepts apply.

For all requests, the server generates and sends a response. The response includes a series of headers and, in the payload, the body of the data requested.

18.4. HTTP: Structure of a Request

Below is a very simple HTTP request.

```
GET / HTTP/1.1
Host: squigler.com
Dnt: 1
```

The first line of the request contains the method of the request (`GET`), the path of the request (`/`), and the protocol version (`HTTP/1.1`). This is an example of a GET request. Each line after the first line is a request header. In this example, there are two headers, the DNT header and the Host header. There are many HTTP headers defined in the HTTP spec which are used to convey various pieces of information, but we will only be covering a couple of them through this chapter.

Here is another HTTP request:

```
POST /login HTTP/1.1
Host: squigler.com
Content-Length: 40
Content-Type: application/x-url-formencoded
Dnt: 1

username=alice@foo.com&password=12345678
```

Here, we have a couple more headers and a different request type: the POST request.

18.5. HTTP: GET vs. POST

While there are quite a few methods for requests, the two types that we will focus on for this course are GET requests and POST requests. GET requests are generally intended for "getting" information from the server. POST requests are intended for sending information to the server that somehow modifies its internal state, such as adding a comment in a forum or changing your password.

In the original HTTP model, GET requests are not supposed to change any server state. However, modern web applications often change server state in response to GET requests in query parameters.

Of note, only POST requests can contain a body in addition to request headers. Notice that the body of the second example request contains the username and password that the user `alice` is using to log in. While GET requests cannot have a body, it can still pass query parameters via the URL itself. Such a request might look something like this:

```
GET /posts?search=security&sortby=popularity
Host: squigler.com
Dnt: 1
```

In this case, there are two query parameters, `search` and `sortby`, which have values of `security` and `popularity`, respectively.

18.6. Elements of a Webpage

The HTTP protocol is designed to return arbitrary files. The response header usually specifies a media type that tells the browser how to interpret the data in the response body.

Although the web can be used to return files of any type, much of the web is built in three languages that provide functionality useful in web applications.

A modern web page can be thought of as a distributed application: there is a component running on the web server and a component running in the web browser. First, the browser makes an HTTP request to a web server. The web server performs some server-side computation and generates and sends an HTTP response. Then, the browser performs some browser-side computation on the HTTP response and displays the result to the user.

18.7. Elements of a Webpage: HTML

HTML (Hypertext Markup Language) lets us create structured documents with paragraphs, links, fillable forms, and embedded images, among other features. You are not expected to know HTML syntax for this course, but some basics are useful for some of the attacks we will cover.

Here are some examples of what HTML can do:

- Create a link to Google: `<a href="http://google.com">Click me</a>`
- Embed a picture in the webpage: `<img src="http://cs161.org/picture.png">`
- Include JavaScript in the webpage: `<script>alert(1)</script>`
- Embed the CS161 webpage in the webpage: `<iframe src="http://cs161.org"></iframe>`

Frames pose a security risk, since the outer page is now including an inner page that may be from a different, possibly malicious source. To protect against this, modern browsers enforce frame isolation, which means the outer page cannot change the contents of the inner page, and the inner page cannot change the contents of the outer page.

18.8. Elements of a Webpage: CSS

CSS (Cascading Style Sheets) lets us modify the appearance of an HTML page by using different fonts, colors, and spacing, among other features. You are not expected to know CSS syntax for this course, but you should know that CSS is as powerful as JavaScript when used maliciously. If an attacker can force a victim to load some malicious CSS, this is functionally equivalent to the attacker forcing the victim to load malicious JavaScript.

18.9. Elements of a Webpage: JavaScript

JavaScript is a programming language that runs in your browser. It is a very powerful language–in general, you can assume JavaScript can arbitrarily modify any HTML or CSS on a webpage. Webpages can include JavaScript in their HTML to allow for dynamic features such as interactive buttons. Almost all modern webpages use JavaScript.

When a browser receives an HTML document, it first converts the HTML into an internal form called the DOM (Document Object Model). The JavaScript is then applied on the DOM to modify how the page is displayed to the user. The browser then renders the DOM to display the result to the user.

Because JavaScript is so powerful, modern web browsers run JavaScript in a sandbox so that any JavaScript code loaded from a webpage cannot access sensitive data on your computer or even data on other webpages.

Most exploits targeting the web browser itself require JavaScript, either because the vulnerability lies in the browser's JavaScript engine, or because JavaScript is used to shape the memory layout of the program for improving the success rate of an attack.

Almost all web browsers implement JavaScript as a Just In Time compiler, dynamically converting JavaScript into machine code[2]. Many modern desktop applications (notably Slack's desktop client) are actually written in the Electron framework, which is effectively a cut down web browser running JavaScript.

[2]Trivia: Running JavaScript fast is considered so important that ARM recently introduced a dedicated instruction, FJCVTZS (Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero), specifically to handle how JavaScript's math operates.