hw03

Daven Farnham

February 2017

1. (a) This seems much less secure since the security question will frequently be easier to crack than the actual password and this protocol effectively lets someone bypass the password if they can figure out the security question. Security questions can often be questions like "what was your first car?" or "what street did you live on as a kid?". These kinds of questions for an attacker, especially one who knows you or, if you're famous, can Google things about you, will be much easier to determine than a password.

   (b) This is also less secure than a website that requires a correct password and security question every time. In this case, you still need to provide a correct password whereas if you can reset your password with a security question, this essentially allows you to bypass the password completely.

2. User Agents:

```
sources:
https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
https://en.wikipedia.org/wiki/HTTP_persistent_connection
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control
```

curl:

```
GET / HTTP/1.1
User-Agent: curl/7.38.0
Host: localhost:1234
Accept: */*
```

wget:

```
GET / HTTP/1.1
```

```
User-Agent: Wget/1.16 (linux-gnu)
Accept: */*
Host: localhost:1234
Connection: Keep-Alive
```

chrome:

```
GET / HTTP/1.1
Host: localhost:1234
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/56.0.2924.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: en-US,en;q=0.8
```

(a) So for all of these, the very first line represents the 'request line' which is made up of the Method SP Request-URI SP HTTP-Version.

- Method: the method above is a 'GET' request meaning most clearly that data produced will be returned in the 'entity' of the response and not the source text. So, for example, when you search for things in Google those are 'GET' requests; variables are passed in the URL making it easy to copy and share the same request. For more secure requests where you might not want the parameters of the search visible, you'd use a 'POST' request.
- Request-URI: this tells the server where to look, so in this case '/' signifies the home directory
- HTTP-Version: the version of HTTP we're using; in this case 1.1

(b) next comes the user-agent, which just describes how we're interacting with the server (either through curl, wget, or a web browser)

(c) the host 'specifies the Internet host and port number of the resource being requested'

(d) the accept request header indicates what media response types are acceptable; in this case '*/*' means all types are acceptable. in addition to this, chrome has a few other type parameters as well as an 'accept parameter' 'q' which denotes the quality it wants to accept.

So, for example, chrome will accept 90 percent quality from something like 'text/html', but only 80 percent from other forms of media not explicitly mentioned.

(e) the connection header, which for wget and chrome specifies to keep the connection alive, basically means I want to keep the TCP connection open and not have to open a new one each time I request and get something back from the server.

I think historically connections haven't been persistent, meaning I would open a connection to a server, request information, get a response, and then close that particular connection. With persistent connections, multiple requests/responses can be performed on the same connection which makes a lot of sense for something like a web brower – it seems more efficient in reducing latency and congestion when you're interacting with a server for a non-trivial amount of time.

For curl, however, which is simply making a single request to the server, the connection doesn't need to be persistent. So, for example, if you made a 'curl "www.google.com"' request you'll get back the html from google's homepage and that's it – curl makes a single request to the server.

(f) For google chrome, it has an 'Upgrade-Insecure-Requests' field which means the client can handle if the server upgrades to a more secure version of the site. Notice if you type 'http://www.google.com/' it'll automatically redirect to 'https://www.google.com/'.

(g) The accept-encoding and accept-language fields are similar to 'Accept', they denote content codings and the set of natural languages preferred on the system. Content codings are generally used to transform or compress a document, so Google says it'll accept certain kinds of compressed data. In addition, it says that it prefers US English.

(h) the cache-control header specifies how long responses can be cached. I think since the max-age is set to 0, that means caches need to revalidate their entries – resources won't be considered "fresh" after their request.

It makes sense google defines more parameters in terms of what it accepts, upgrading to secure versions of a site, etc, but notice you can pass parameters to a number of these user agents, for example you can pass '–no-http-keep-alive' to wget to change the connection header to 'close' instead of 'keep-alive'. Although it might make sense to keep a connection open s.t. multiple requests can be made, you could change it so that you have to open a new TCP connection for each request.

3. (a) the biggest thing I can think of in addition to caches and tracking pixels, which are talked about in the other questions, is simply the IP address of your browser. Whenever your browser requests information from a server, the server needs to know where to send it - the IP address the server responds to can give a general idea of your location.

   For example, if you go to 'https://www.whatismybrowser.com/detect/ip-address-location', unless you're using something like a VPN, it'll give you an approximate location where you are. For me, it can identify that I'm in Providence – information that would be useful to any company trying to determine where their clientele is. Even if you go 'incognito', you won't be able to hide this information.

   (b) tracking pixels are usually little, 1x1 transparent pixels that are embedded on a webpage when a user requests information from a server. So, for example, let's say you're a company and you want to run some ads on ESPN. You prepare an image that is rendered on a client's browser when they visit ESPN, but within this image is a tracking pixel. When this image is loaded, the tracking pixel makes a call to the advertiser's server, which can then take note of the IP address that requested information. The server returns the transparent pixel in response to the HTTP request – the client therefore has no idea it made a server call while the server is able to gather information about the user.

   A tracking pixel is essentially a small, transparent pixel the user doesn't know of that can make server calls to relay information back to another party.

   (c) One thing that needs to be noted is that caches are different from cookies, so even if a user clears their cookies certain information will still be maintained in the browser's cache. What a server could do with ETags would be to generate an ETag for each unique user that comes to the site - so now the ETag is not mainly used to mark information as fresh or not, but rather to identify the client making the server request.

   So, for example, a client makes a request from the server and the server responds with a unique ETag. Now, every time that client makes future server requests that particular ETag will be included. If the server never invalidates the ETag, or just keeps refreshing the same ETag, then as long as that tag is in the user's cache it can be used to identify a particular client.

4. (a) According to the Wikipedia page, it's appropriate to use a GET request when you're requesting information from the server, but not inducing any side effects. POST requests ask the server to accept

some kind of information, such as a post to a message board or an item to be added into a database - something that might change the state of information on the server.

Thinking about this another way, it's appropriate to use a GET request, as opposed to a POST request, when the information being sent over the URL isn't sensitive and you might want to share the URL request with someone. This is why many Google searches use a GET request - if I want to share the results of the search I can simply copy and paste the URL; all the search parameters and variables will be included in the header. It wouldn't make sense to use a GET request if you're sending sensitive information between a client and a server, since in this case anyone viewing the URL would have access to this information.

(b) This leads into where information is stored by POST and GET requests. GET requests don't have a message body, but rather transfer information through the request headers. These headers, however, as described above are visible (either by looking at parts of the URL or sniffing traffic between a server and client). POST requests, however, send information in the message body as opposed to the headers.

(c) As mentioned above, just by seeing a URL or the HTTP headers of a request transmitting data via a GET request you might be able to determine sensitive data communicating between a client and a server. If something like the user's cookies were accidentally transmitted via a GET request, anyone sniffing traffic would have plenty of sensitive information pertinent to an individual user.

(d) From what I can tell, POST requests send information (not immediately obvious since it's not in the HTTP headers) that could possible change the state of something server side or require some kind of data handling. You wouldn't want to POST requests from typing in or clicking a URL since the user might not know what they're causing to happen.

Concretely, let's say there's a link on a website or I send an innocuous looking URL to someone. Someone clicking on the link or pasting in the URL might be sending information they don't want to send to a server without ever being aware of it, since the request is hidden in the message body and not in the header. This is why it's probably not allowed to send POST requests from simple URLs.

5. Maybe what you could do is let's say you're on a site that ostensibly hosts images. Well, instead of uploading an image to the site you instead upload a PHP script. Taking the example from the question, you then run the PHP script by visiting the URL 'http://www.foo.com/uploads/myscript.php' thereby executing the

PHP you wrote on the server (instead of viewing the image, which is what the server intended).

Taking this a little further, set's say that files in the 'uploads' folder aren't executable. Well, if you had an idea where a folder is with the correct permissions, say the root directory, you might be able to name your file something like "../myscript.php" which, depending on how the server is uploading files, might end of storing the file in the root directory ('http://www.foo.com/uploads/../myscript.php'), i.e. you might be able to run this by visiting 'http://www.foo.com/mysript.php'