Unit 08 Step 3 Reading Questions
Jeremiah Jekich
Web Applications

1. **(2) HTTP is a stateless protocol - what does that mean?**

   The fact that HTTP is stateless means that it retains no "memory" about previous communications. Every request and response transaction that occurs is independent. HTTP alone cannot enable useful functionality such as remembering that a user is logged in or providing a persistent shopping cart. On the up side, this makes for a simpler server design since the server doesn't need to deal with storing extra data, but on the down side, in order to provide persistent/stateful functionality like that mentioned above, more work must be done elsewhere.

2. **(2) What is a session id?**

   A session id is a (theoretically) unique 32 byte long MD5 hash value used to identify a session. Session ids are needed in a stateless protocol to provide the persistent/stateful functionality described in question one above.

3. **(4) Explain the relationship between cookies and sessions**

   A cookie is a small collection of data sent from a server to the user's browser. Whereas sessions contain information stored on the server, cookies contain information stored in the client (browser). The cookie contains the session id, and every time the browser makes a request to the server, it sends along this id. The server then compares it with the id of the session stored server-side to make sure the transaction is legitimate (that the request is valid and coming from the same session that was set up previously). The server can then send session information from the server-side data to the browser. This is the stateful information. Alternatively, the cookie can contain all of the stateful information for the session client-side. Cookies are an alternative to passing the session id in the URL, which has some security issues.

   Also note that the cookies described above are known as session cookies and they are typically deleted when a browser is closed. However, there are also other cookies that are not directly related to the session, and that are deleted (or *expire*) at a specified date.

4. **(2) Name two guidelines for storing sessions**

   1 – *Do not store large objects in a session.* Instead, store the objects in a database and keep their ids in the session. This prevents slowdown in communications and also allows you to modify the server-side object without worrying about having a client-side copy that is out of sync.

2 – *Critical data should not be stored in a session.* Cookies can be deleted, which means you can lose any data stored in them, including critical data. Also, the user can see cookie data, so anything they aren't allowed to see should not be stored in the cookie.

5. **(4) What is CSRF?**

   CSRF, which stands for Cross-Site Request Forgery takes advantage of the fact that a session cookie for an application may still exist in the user's browser after the user has finished with the application but forgot to log out (or the cookie still exists for some other reason). The fact that this cookie still exists means that if the browser attempts to contact that application's server, it will send along the cookie, providing valid session credentials and allowing access to the application. If a hacker tricks the user into initiating a request to the application (for example with an API call hidden in an image tag, which the user triggers simply by loading the page containing the img tag), the hacker can secretly and "legitimately" send the request to the application. This gives the hacker access to the application, for at least on request, under the user's credentials. Thus, the hacker can perform malicious acts, such as deleting data from the application's database.

6. **(4) What steps does Rails take to protect the user from CSRF? Note: You might want to look at application_controller.rb and application.html.erb - after this reading you should understand more about some of the options in these files.**

   Rails uses a required security token that is known by the application, but is not known by other applications. This automatically includes the security token in Rails-generated forms and Ajax requests. The token is checked by the server, and if it doesn't match, an exception is thrown. This functionality is added automatically to a Rails application and is enabled by the following line in the `ApplicationController`:

   ```
   protect_from_forgery with: :exception
   ```

   Rails also allows you to define your own method for responding to invalid authenticity tokens. You can call the method from the `ApplicationController` using the following syntax:

   ```
   rescue_from ActionController::InvalidAuthenticityToken do |exception|
     my_method # method for handling the invalid token
   end
   ```

   Also, Rails does not allow cross-site `<script>` tags to prevent an attacker from making a cross-site request via JavaScript.

7. **(2) Compare whitelisting to blacklisting. How does this relate to the params hash? (not explicitly mentioned in reading)**

   Whitelisting is only allowing items present on a list, known as a whitelist. Whitelisting can be used when you only want to accept a certain set of inputs.

Blacklisting is denying items present on a list, known as a blacklist. Blacklisting can be used when you want to deny a certain set of inputs. This is useful for actions such as denying banned users from a site, preventing communication with a specific set of known malicious sites, or preventing user input that includes characters that cannot be handled, such as single or double quotes.

With regard to the params hash, we whitelist only the fields that exist for a model with a line like the following:

```
params.require(:pet).permit(:name, :age, :description, :pet_type)
```

This prevents anyone from submitting other (potentially malicious) fields to the database. In a sense, by whitelisting these fields, we effectively blacklist all other possible fields.