Cookies

A. I can see one cookie for `cs338.jeffondich.com`. It has the name “theme” and value “default”.

B. I changed my theme to “Blue” using the menu, and now the “theme” cookie has value “blue”.

C. When I do steps A and B in Burpsuite’s browser, I see the same “theme=default” cookie first and then the “theme=blue” cookie after changing the theme. Both cookies were both listed in the “`Set-Cookie`” header in the HTTP response and then observed again in the inspector.

D. After quitting and relaunching my browser, I still have the same blue theme.

E. The current theme is transmitted between the browser and FDF server using the “theme=blue” cookie, sent by the browser to the server with every HTTP request.

F. When I change the theme via the menu, the browser sends an HTTP GET request to the server with the URL “`/fdf/?theme=blue`”. The server then responds with an HTTP response for the current page (ie. the home page), but includes the “`Set-Cookie`” header with value “theme=blue”. The browser then stores “blue” as the value for the “theme” cookie, overwriting the previous value. (Interestingly, this process doesn’t seem to happen on individual posts. The same URL format (current path followed by “`?theme=[theme-value]`”) is used for the HTTP GET request, but the response doesn’t include the “`Set-Cookie`” header and instead just returns the version of the page with the appropriate theme in place.)

G. Since cookies are stored in my browser, I can directly edit the value of the “theme” cookie within the inspector. If I change the value to “red”, the theme will be red when I reload the page.

H. When Burpsuite intercepts the HTTP GET request, the "theme" cookie is always present. I can simply edit the cookie to have value "blue" and then forward the request, and when the server responds, it will send the version of the requested page with the blue theme.

I. The cookies for my Firefox browser in Kali are stored in the "moz_cookies" table within the "`cookies.sqlite`" SQLite database and can be found at the path "`~/.mozilla/firefox/vq79p5j0.default-esr/cookies.sqlite`".
The cookies for my Burpsuite browser in Kali are stored in the "`cookies`" table within the "`Cookies`" SQLite database and can be found at the path "`~./BurpSuite/pre-wired-browser/Default/Cookies`". However, the values in this table are encrypted via some mechanism, and I wasn't able to figure out how to decrypt them.

Cross-Site Scripting

A. The attack begins with Prof. Moriarty creating his post and including some JavaScript in his post body, surrounded by the HTML script tag. Some time later, Alice clicks on the post. When she does, her browser makes an HTTP GET request for the page with more details about Moriarty's post, and the server sends the HTML of that page back to Alice. Her browser receives the HTML and begins executing it to display the webpage. As it parses through, it comes across the "posts" division, sees the script tag inside it, and interprets the contents of the tag as JavaScript to be run. Prof Moriarty's code executes and the attack is carried out (here, an alert is queued to be displayed once the rest of the HTML is parsed and the page is loaded). Finally, the page finishes loading and Alice can see the post that Prof. Moriarty made.

B. One more dangerous XSS attack that Moriarty could execute is to redirect the user to a different webpage when they click on the post. He could use this webpage to phish the user if he makes it look convincingly similar, or he could just redirect the user to a page which directly installs malware upon access.

C. Another more dangerous XSS attack that Moriarty could execute exploits JavaScript's access to browser cookies. Since JavaScript has full access to the user's cookies, he could execute some JavaScript that steals the session cookie of the user and reports it to Moriarty. Then, Moriarty has access to the user's account (and could easily gain sole control over the account by changing the login credentials).

D. The server can disallow posts which contain HTML script tags. Then, Moriarty couldn't make these posts in the first place and users wouldn't have his scripts running in their browser. Browsers could also be strict in not allowing JavaScript to run (or not allowing it on certain sites). However, both of these actions could prevent scripts from running which users actually want to run. A better approach for the server might be to ensure that character escaping happens correctly (so that something like the Twitter XSS attack from 2010 doesn't happen). Then, users can still feel safe running JavaScript code from the site knowing that it was intentionally programmed by the developers (provided the user trusts the developers). Browsers could also give users the option to disable JavaScript on specific sites, giving the user a bit more control in case there are specific sites they do want to execute scripts from.