Question 1  \textit{Cross-Site Scripting (XSS)} \hfill (15 \text{ min})

The figure below shows the two different types of XSS.

As part of your daily routine, you are browsing through the news and status updates of your friends on the social network FaceChat.

(a) While looking for a particular friend, you notice that the text you entered in the search string is displayed in the result page. Next to you sits a suspicious looking student with a black hat who asks you to try queries such as

\[ \text{<script>alert(42);</script>} \]

in the search field. What is this student trying to test?

(b) The student also asks you to post the code snippet to the wall of one of your friends. How is this test different from part (a)?

(c) The student is delighted to see that your browser spawns a JavaScript pop-up in both cases. What are the security implications of this observation? Provide a malicious URL that steals other users’ cookies.
(d) Why does an attacker even need to bother with XSS? Wouldn’t it be much easier to just create a malicious page with a script that steals all cookies of all pages from the user’s browser?

(e) FaceChat finds out about this vulnerability and releases a patch. You find out that they fixed the problem by removing all instances of \texttt{<script>} and \texttt{</script>}.

Why is this approach not sufficient to stop XSS attacks? What’s a better way to fix XSS vulnerabilities?
Question 2  Session Fixation  

Some web application frameworks allow cookies to be set by the URL. For example, visiting the URL

\[ \text{http://foobar.edu/page.html?sessionid=42.} \]

will result in the server setting the sessionid cookie to the value “42”.

(a) Can you spot an attack on this scheme?

(b) Suppose the problem you spotted has been fixed as follows. foobar.edu now establishes new sessions with session IDs based on a hash of the tuple (username, time of connection). Is this secure? If not, what would be a better approach?
Question 3  \textit{Cross Site Request Forgery (CSRF)}  \hspace{1cm} (15 \text{ min})

In a CSRF attack, a malicious user is able to take action on behalf of the victim. Consider the following example. Mallory posts the following in a comment on a chat forum:

\begin{verbatim}
<img src="http://patsy-bank.com/transfer?amt=1000&to=mallory"/>
\end{verbatim}

Of course, Patsy-Bank won’t let just anyone request a transaction on behalf of any given account name. Users first need to authenticate with a password. However, once a user has authenticated, Patsy-Bank associates their session ID with an authenticated session state.

(a) Sketch out the process that occurs if Alice wants to transfer money to Bob. Explain what happens in Alice’s browser and patsy-bank.com’s server, as well as what information is communicated and how.

(b) Explain what could happen when Alice visits the chat forum and views Mallory’s comment.

(c) What are possible defenses against this attack?
Question 4  

CSRF++  

Patsy-Bank learned about the CSRF flaw on their site described above. They hired a security consultant who helped them fix it by adding a random CSRF token to the sensitive /transfer request. A valid request now looks like:

https://patsy-bank.com/transfer?to=bob&amount=10&token=<random>

The CSRF token is chosen randomly, separately for each user.

Not one to give up easily, Mallory starts looking at the welcome page. She loads the following URL in her browser:

https://patsy-bank.com/welcome?name=<script>alert("Jackpot!");</script>

When this page loaded, Mallory saw an alert pop up that says “Jackpot!”. She smiles, knowing she can now force other bank customers to send her money.

(a) What kind of attack is the welcome page vulnerable to? Provide the name of the category of attack.

(b) Mallory plans to use this vulnerability to bypass the CSRF token defense. She’ll replace the alert("Jackpot!"); with some carefully chosen JavaScript. What should her JavaScript do?

(c) Mallory wants to attack Bob, a customer of Patsy-Bank. Name one way that Mallory could try to get Bob to click on a link she constructed.
Question 5  *Cross-site not scripting*  

Consider a simple web messaging service. You receive messages from other users. The page shows all messages sent to you. Its HTML looks like this:

```html
<pre>
Mallory: Do you have time for a conference call?
Steam: Your account verification code is 86423
Mallory: Where are you? This is <b>important!!!</b>
Steam: Thank you for your purchase
    <img src="https://store.steampowered.com/assets/thankyou.png">
</pre>
```

The user is off buying video games from Steam, while Mallory is trying to get a hold of them.

Users can send *arbitrary HTML code* that will be concatenated into the page, **un-sanitized**. Sounds crazy, doesn’t it? However, they have a magical technique that prevents *any* JavaScript code from running. Period.

Discuss what an attacker could do to snoop on another user’s messages. What specially crafted messages could Mallory have sent to steal this user’s account verification code?