CS161 Fall 2026 — Introduction to Computer Security — Exam Prep 8

Q1 *Web Security: Super Query League* ⚽ **(16 points)**

EvanBot has created a fantasy football app called FantasyLeague. Each user is able to name their team, draft players, and manage lineups. The app relies on the following SQL tables:

Account	
username	string
password	string
team_name	string

Team	
team_name	string
draft_order	uint32_t
completed	boolean

Player	
team_name	string
player_name	string
in_lineup	boolean

When users create an account, they select a `team_name` that is used to represent their team. In each round of drafting, the `draft_order` determines the order in which teams get to select players. Once a team has run out of people to draft, `completed` is set to `true` and they can start playing. Each week, they set the `in_lineup` variable for some of the players on their team, and score based on the people who are in their lineup.

Q1.1 (3 points) To search for players, the app executes the following query:

```
SELECT player_name, team_name FROM Player WHERE player_name = '$search';
```

Which of the following payloads, when injected into the `$search` parameter, will allow the attacker to learn the `username` and `password` of every user in the `Account` table, if the attacker can see the results of the query?

Select all that apply.

- ☐ `' OR 1=1; --`
- ☐ `' UNION SELECT username, password FROM Account --`
- ☐ `' UNION SELECT username, password, team_name FROM Account --`
- ☐ `' UNION SELECT password, username FROM Account --`
- ☐ `' UNION SELECT * FROM Account --`
- ☐ `' UNION SELECT player_name, team_name FROM Player --`

This content is protected and may not be shared, uploaded, or distributed.

Q1.2 (3 points) Jonah's jealousy knows no bounds. He decides to ban his rival, **Fred**, from the platform entirely. To login, the app executes the following query:

```
SELECT * FROM Account WHERE username = '$username';
```

When injected into the **$username** parameter of the login query, which of the following payloads will delete the row **Fred** from the **Account** table, and only delete that row?

Select all that apply.

Note: **DELETE FROM table WHERE condition** removes rows from **table** that satisfy **condition**.

- ☐ `'; DELETE FROM Account WHERE username = 'Fred';`
- ☐ `'; DELETE FROM Account WHERE username = 'Fred'; --`
- ☐ `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1`
- ☐ `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1';`
- ☐ `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1'; --`
- ☐ `'; DELETE FROM Player WHERE player_name = 'Fred';`
- ☐ `'; DELETE FROM Player WHERE player_name = 'Fred'; --`

Q1.3 (2 points) The login page has a "Check Username" feature to see if a user exists. It is vulnerable to SQL injection but does not display any database content. It simply prints "User Found" if the query returns any rows, and "User Not Found" otherwise.

```
SELECT * FROM Account WHERE username = '$user';
```

You suspect the **admin** user has a password starting with the letter 's'. Which payload(s) will let you infer whether your suspicion is correct, based on whether the program prints "User Found" or not in response to your payload?

Select all that apply.

Note: **substring(str, start, length)** extracts **length** characters from **str** beginning at index **start**.

- ☐ `admin' --`
- ☐ `admin' OR '1'='1' --`
- ☐ `admin' AND substring(password, 1, 1) = 's' --`
- ☐ `admin' AND '0'='1' UNION SELECT * FROM Account WHERE substring(password, 1, 1) = 's' --`

This content is protected and may not be shared, uploaded, or distributed.

Q1.4 (3 points) You can ask the app to release a player from their team. The app will execute the following SQL command.

```
DELETE FROM Player
 WHERE player_name = '$player'
   AND team_name = '$my_team';
```

You are logged in as `guest` and your team name is `GuestTeam`. Assume `$my_team` is set to `GuestTeam` and cannot be modified. You control `$player` but not `$my_team`.

Provide a SQL payload for the `$player` input that will delete every player belonging to the rival team `Winners`, without deleting players from other teams.

Q1.5 (3 points) On signup, the application executes the following SQL query as a prepared statement. The user-provided inputs `user` and `team` are bound securely to the `?` placeholders:

```
INSERT INTO Account (username, password, team_name)
VALUES (?, 'dummy_pass', ?);
```

Later, when a user views their team page, the app executes the following SQL query (notice how there is no prepared statement here):

```
SELECT * FROM Team WHERE team_name = '$my_team';
```

Assume `$my_team` is retrieved securely from the `Account` table and matches the `team` provided at signup.

Provide a stored-injection payload for the `team` parameter during signup that will drop the entire `Team` table when the user visits their team page later.

Q1.6 (2 points) In 10 words or fewer, describe one other serious security flaw with the signup design in the prior part. Do **not** mention SQL injection; find some other flaw.

Q2 *Web Security: Criss Cross Apple Sauce* 🍎 **(12 points)**

This content is protected and may not be shared, uploaded, or distributed.

Q2.1 (2 points) Alice visits a search engine that displays her search terms back to her on the results page. For example, if she searches for "cats", the page displays: `You searched for: cats`.

Mallory notices that the website does not sanitize this output. She creates a link containing a malicious JavaScript payload in the search query and tricks Alice into clicking it. When Alice clicks the link, the script executes in her browser.

What specific type of attack is this?

- ◯ Stored XSS
- ◯ Reflected XSS
- ◯ Cross-Site Request Forgery (CSRF)
- ◯ SQL Injection

Q2.2 (1 point) Mallory creates a malicious website called `freemoney.com`. On this page, she places a "Claim Prize" button.

Directly on top of this button, she layers a transparent (invisible) `<iframe>` that loads Alice's bank account settings page, specifically positioning the "Delete Ac=count" button directly over her "Claim Prize" button.

Alice visits `freemoney.com` and clicks "Claim Prize," but she unknowingly clicks the "Delete Account" button on the invisible bank page.

What specific type of attack is this?

- ◯ Phishing
- ◯ Cross-Site Request Forgery (CSRF)
- ◯ Clickjacking (UI Redressing)
- ◯ Man-in-the-Middle (MITM)

Q2.3 (2 points) A web developer tries to prevent XSS by writing a filter that simply removes the text `<script>` and `</script>` from all user input.

Which of the following payloads would successfully bypass this specific filter and execute JavaScript?

- ◯ `<script>alert(1)</script>`
- ◯ `<img src=x onerror="alert(1)">`
- ◯ `<bold>alert(1)</bold>`
- ◯ `<a href="http://evil.com">Click me</a>`

This content is protected and may not be shared, uploaded, or distributed.

Q2.4 (2 points) Mallory creates a malicious website containing a hidden HTML form. When Alice visits Mallory's site, a script automatically submits this form to `bank.com` to transfer money.

Why does `bank.com` accept this request and transfer the money, even though Alice did not intend to click the button?

- ◯ Mallory guessed Alice's password and included it in the form.
- ◯ Alice's browser automatically attached her `bank.com` session cookies to the request.
- ◯ Mallory intercepted Alice's Wi-Fi traffic and modified her packets.
- ◯ The bank's server allows any request that comes from the same IP address as Mallory.

Q2.5 (1 point) A server defends against CSRF by checking the `Referer` header. It rejects requests where the Referer is `evil.com`. However, to protect user privacy, the server accepts requests where the `Referer` header is missing (blank).

How can Mallory bypass this defense?

- ◯ By encrypting the Referer header so the server cannot read it.
- ◯ By spoofing the IP address to look like it came from the server.
- ◯ By configuring her website (e.g., using `<meta name="referrer" content="no-referrer">`) to tell the browser not to send a Referer header.
- ◯ It is impossible to bypass; browsers always send the Referer header.

Q2.6 (2 points) Alice wants to implement a strong defense against XSS that restricts where scripts can be loaded from. She configures her server to send a specific HTTP header that tells the browser: "Only load scripts from `https://mysite.com`. Block all other scripts."

What is the name of this defense?

- ◯ Same-Origin Policy (SOP)
- ◯ Content Security Policy (CSP)
- ◯ Cross-Site Request Forgery (CSRF) Token
- ◯ Referer Validation

Q2.7 (2 points) The most robust defense against CSRF is the use of a CSRF Token (a random, secret value included in forms).

Why does this prevent Mallory from successfully forging a request from `evil.com`?

- ◯ The token encrypts the session cookie so Mallory cannot use it.
- ◯ Mallory cannot read the token from the legitimate site due to the Same-Origin Policy, so she cannot include the correct token in her forged request.
- ◯ The token verifies that the IP address of the request matches the IP address of the user.
- ◯ The token prevents the browser from sending cookies to the server.

This content is protected and may not be shared, uploaded, or distributed.