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.

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

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 --`

Solution: The goal of this question is to extract data from a different table (`Account`) than the one the query was originally designed for (`Player`). To do this, we use the `UNION` operator. The `UNION` operator allows you to combine the results of two different `SELECT` statements into a single result set.

The injected `SELECT` statement must have the exact same number of columns as the original `SELECT` statement.

Now, turning to the options

- `' OR 1=1; --` Incorrect. This is a tautology attack. It returns all rows from the `Player` table, but it fails to retrieve data from the `Account` table.
- `' UNION SELECT username, password FROM Account --` Correct. This selects exactly 2 columns (`username`, `password`). Since 2 matches the original query's 2, the database accepts it.
- `' UNION SELECT username, password, team_name FROM Account --` Incorrect. This explicitly selects 3 columns, causing a syntax error because the column count does not match the original query.
- `' UNION SELECT password, username FROM Account --` Correct. This also selects exactly 2 columns. The database accepts it regardless of the logical order of data (it simply puts the password in the first slot and username in the second).
- `' UNION SELECT * FROM Account --` Incorrect. This selects all 3 columns from `Account`. Since 3 does not equal 2, this causes a syntax error.
- `' UNION SELECT player_name, team_name FROM Player --` Incorrect. This returns data from the `Account` table, but it doesn't reveal the username or password, so it doesn't meet the requirements of the question.

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'; --`

Solution: The goal is to remove a specific user from the **Account** table. Like the previous question, this requires a Stacked Query injection using a semicolon (**;**) to execute a second command after the initial login query. To be correct, the payload must:

1. Close the original string (**'**).
2. End the statement (**;**).
3. Execute a **DELETE** on the correct table (**Account**).
4. Comment out the rest of the query (**--** or **#**).

Now, turning to the options:

- `'; DELETE FROM Account WHERE username = 'Fred';` Incorrect. The resulting query will end with a mismatched single-quote, which will cause a syntax error.
- `'; DELETE FROM Account WHERE username = 'Fred'; --` Correct. Standard stacked query syntax targeting the correct table.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1` Correct. A `';` will be appended to the end from the original query, so the last quote will be closed and all quotes will match, and the condition `'1' = '1'` will be a tautology.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1';` Incorrect. The original query will append `';`, yielding mismatched quotes.
- `'; DELETE FROM Account WHERE username = 'Fred' AND '1' = '1'; --` Correct. The `--` ensures that the quote appended after this is ignored.
- `'; DELETE FROM Player WHERE player_name = 'Fred';` Incorrect. This will cause a syntax error after `';` is appended, yielding a mismatched quote. Also, it deletes from the wrong table.
- `'; DELETE FROM Player WHERE player_name = 'Fred'; --` Incorrect. This deletes data from the **Player** table. The question specifically asked to delete the row from the **Account** table.

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

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' --`

Solution: The goal is to leak data by making the "User Found" message appear conditionally. We need a query that returns a row **only** if our guess is correct.

- `admin' --` Incorrect. Always finds the admin user. The program prints "User Found" regardless of the password.
- `admin' OR '1'='1' --` Incorrect. Always finds the admin user. The program prints "User Found" regardless of the password.
- `admin' AND substring(password, 1, 1) = 's' --` Correct. The program will print "User Found" if and only if **admin**'s password starts with 's'.
  - If password starts with 's', the expression evaluates to `True AND True` for the **admin** user, so the SQL query returns one row for the **admin** user, so the program prints "User Found".
  - If password does not start with 's', the expression evaluates to `True AND False` for the **admin** user, so the SQL query nothing, so the program prints "User Not Found".
- `... UNION SELECT ...` Incorrect. The `UNION` finds all users whose password starts with 's'. The program prints "User Found" if any user has a password starting with 's' (regardless of whether it is the **admin** user or not), or "User Not Found" if no user has a password starting with 's', so this does not let you infer whether **admin**'s password starts with 's'. Most likely, if there are many users, at least one of them will have a password starting with 's', so most likely the program will print "User Found" no matter what the **admin**'s password might be.

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.

Solution: `Scrub' and '0'='1'; DELETE FROM Player WHERE team_name = 'Winners'; --`

Solution: The goal is to execute an arbitrary `DELETE` command against a specific target (`Winners`), bypassing the team name restriction enforced by the application.

Modifying the existing query is a little tricky because the application appends `AND team_name = 'GuestTeam'` to the end.

The cleanest solution is a Stacked Query injection:

1. Finish the first query: `Scrub' AND '0'='1'` closes the string, and `;` terminates the statement. This lets us start a new SQL command, without deleting any players from other teams. It would also work to provide a player name that is unlikely to appear in any team.
2. Start a new query: `DELETE FROM Player WHERE team_name = 'Winners';` is a completely new command. Since we are starting fresh, we are not bound by the `AND team_name = 'GuestTeam'` constraint of the previous query.
3. Cleanup: `--` comments out the dangling `AND team_name = 'GuestTeam'` that the application appends, preventing a syntax error.

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.

Solution: `x'; DROP TABLE Team; --`

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

Solution: A suitable payload for `$team` is:

```
x'; DROP TABLE Team; --
```

1. At signup, this entire string is stored as the `team_name` for the new account, and stored in the `Account` table. Since the `INSERT` is done using a prepared statement, no injection happens yet.
2. Later, the team page builds a query by concatenating the team name provided earlier: `SELECT * FROM Team WHERE team_name = '$my_team'`;
3. With our payload in `$my_team`, the resulting SQL is: `SELECT * FROM Team WHERE team_name = 'x'; DROP TABLE Team; --'`;
4. The first statement just selects rows with `team_name = 'x'`. The second statement `DROP TABLE Team;` executes and deletes the entire `Team` table.
5. The `--` comments out the trailing ‘ from the original query so there is no syntax error.

The key idea: the malicious string is stored during signup and only turns into executable SQL when the app later injects it into a new query.

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.

There is a default password / Security through obscurity

Solution: In particular, the prepared statement creates an account with the user-supplied username and team name, and uses `dummy_pass` as the initial password. Hopefully, the application later updates the password to whatever password the user selected. However, this provides a brief time window when the account exists with the password `dummy_pass`. During that time, anyone in the world could log into the user’s account, using the password `dummy_pass`.

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

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
- ○ Cross-Site Request Forgery (CSRF)
- ● Reflected XSS
- ○ SQL Injection

Solution: Reflected XSS. The malicious script is not stored on the server’s database; it is “reflected” off the web server immediately as part of the response to the specific request (the search query).

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

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

Solution: Clickjacking (UI Redressing). The attacker uses transparent layers to trick the user into clicking on a UI element (the bank button) that is different from what they perceive they are clicking (the prize button).

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>`

Solution: **`<img src=x onerror="alert(1)">`**. Since the developer only filters the specific **`<script>`** tag, the attacker can use other HTML tags that support event handlers (like **`onerror`** on an image) to execute JavaScript.

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.

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

Solution: Alice's browser automatically attached her `bank.com` session cookies to the request. The core vulnerability of CSRF is that browsers automatically include ambient credentials (cookies) with cross-site requests, so the server cannot distinguish between a user clicking a button and a script clicking it for them.

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.

Solution: By configuring her website to tell the browser not to send a Referer header. If the server "fails open" (accepts requests with no header), the attacker simply needs to suppress the header to bypass the check.

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)
- ○ Cross-Site Request Forgery (CSRF) Token
- ● Content Security Policy (CSP)
- ○ Referer Validation

Solution: Content Security Policy (CSP). CSP allows site administrators to declare approved sources of content that browsers are allowed to load, effectively mitigating XSS by blocking malicious inline scripts or scripts from unauthorized domains.

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

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.

Solution: Mallory cannot read the token from the legitimate site due to the Same-Origin Policy. While Mallory can force the browser to **send** a request (which includes cookies), she cannot **read** the response or the content of the legitimate page to find the secret token. Without that token, the server rejects the forged request.

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