

Forum: [Hack Forums](#)

Thread: [SQL Injection: A Complete Course](#)

ICTKUN - September 20, 2026, 1:21 am

Recon -> Test -> Dump -> Defend

Before you start

Every target in this guide is one of three things:

A deliberately vulnerable app you run on your own machine (DVWA, OWASP Juice Shop)

A hosted lab built for this purpose (PortSwigger Web Security Academy)

A system you have written authorization to test

Nothing here is for systems you don't own or aren't authorized to test. That's not a disclaimer paragraph to skim — it's the boundary that separates coursework from a crime. If you're not sure whether a target is in scope, it isn't. Close the terminal.

The goal of this guide is not "learn to attack." It's learn to see the bug so you can kill it. SQL injection is one of the most preventable classes of vulnerability in existence, and it's still everywhere. The reason is not that it's hard to fix. It's that people learn it as a trick instead of as a failure mode. This guide teaches it as a failure mode

Part 1 — The Bug

1.1 What SQL injection actually is

SQL injection is a string problem. That's the whole thing. Everything else — the payloads, the tooling, the dumps — is downstream of one mistake: user input was treated as code instead of data.

Here's the setup. An app takes input and glues it into a query:

```
Code query = "SELECT * FROM users WHERE username = " + username + ""
```

The database doesn't know which part of that string came from the developer and which came from the user. To it, it's one string. So when a user types:

```
Code admin' --  
...the query becomes:
```

```
Code SELECT * FROM users WHERE username = 'admin' --'  
-- is a SQL comment. Everything after it is ignored. The password check that was supposed to follow is gone. The database returns the admin row.
```

The user's input became code. That's the bug. Every payload in this guide is a variation on that one sentence.

1.2 The one rule

Never build a query by concatenating user input into a string.

Everything else in this guide is either a way to violate that rule (offense) or a way to enforce it (defense). Parameterized queries fix the bug. All other defenses reduce the blast radius if something slips through.

1.3 The fix, in every stack

Code # Python — WRONG

```
cursor.execute("SELECT * FROM users WHERE username = " + username + "")
```

Python — RIGHT

```
cursor.execute("SELECT * FROM users WHERE username = %s", (username,))
```

javascript

Code // Node/pg — RIGHT

```
await client.query('SELECT * FROM users WHERE username = $1', [username]);
```

java

Code // Java/JDBC — RIGHT

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE username = ?");
```

```
ps.setString(1, username);
```

csharp

Code // C#/ADO.NET — RIGHT

```
var cmd = new SqlCommand("SELECT * FROM users WHERE username = @u", conn);
```

```
cmd.Parameters.AddWithValue("@u", username);
```

The query template and the data are sent to the database separately. The DB parses the template, sees the placeholder, and binds the input as a value. It cannot become code, because parsing already happened.

If you take one thing from this guide: parameterize. Not escape. Not sanitize. Parameterize.

1.4 Why this still matters

SQLi is consistently in the OWASP Top 10. It still shows up in breach reports. It happens in three places:

Legacy code nobody wants to touch

ORM escape hatches — someone couldn't figure out the ORM, wrote raw SQL with f-strings

Search filters and admin panels where devs assumed "only internal users hit this"

The last one is the killer. Internal tools get exposed, or an attacker gets one low-priv account, and the "internal only" assumption evaporates.

Part 2 — Setup

You need three targets and one tool.

1. DVWA (local).

Bash

Code `docker run --rm -it -p 80:80 vulnerables/web-dvwa`

Browse to `http://localhost`, log in admin / password, click Create / Reset Database, set security to Low.

2. OWASP Juice Shop (local).

Bash

Code `docker run --rm -it -p 3000:3000 bkimminich/juice-shop`

Browse to `http://localhost:3000`.

3. PortSwigger Web Security Academy. Free account at portswigger.net/web-security. Each lab is a self-contained instance with an "Access the lab" button.

Tooling. A browser, Burp Suite Community (or dev tools), and curl. That's enough. Don't touch sqlmap until Module 3.5.

Part 3 — Recon (Dorking Your Own Target)

"Dorking" here means finding the attack surface on a target you own. In an engagement, it's scoped recon. In the lab, it's how you locate the endpoints that take input.

3.1 The mindset

You're not looking for a magic query. You're looking for places where user input reaches a database. Those places are almost always:

Code Login forms (username, password)

Search boxes

URL parameters (`?id=`, `?category=`, `?user=`)

Filter/sort dropdowns

API endpoints taking an ID or query string

Hidden form fields

Cookies and headers — User-Agent, Referer, X-Forwarded-For sometimes land in queries

3.2 DVWA

Browse every module with Burp running. Note:

SQL Injection — a form taking a User ID. Primary target.

SQL Injection (Blind) — same, no output.

Brute Force — login form, injectable in some modes.

Note the parameter names, methods, and cookies (PHPSESSID, security).

3.3 Juice Shop

Less obvious on purpose. Use the app as a user:

Login form

Search box in the product list

Product detail page (URL like `##/product/1` — the 1 is an ID)

Basket and feedback endpoints

REST API at `/rest/products/search?q=` — the classic Juice Shop SQLi target

Open dev tools -> Network. Every request carrying an editable parameter is a candidate.

3.4 Your worklist

Code	Target	Endpoint	Parameter	Method
DVWA	/vulnerabilities/sqli/	id	GET	
DVWA Blind	/vulnerabilities/sqli_blind/	id	GET	
Juice Shop	/rest/products/search	q	GET	
PortSwigger	varies per lab	varies	varies	

That list is your worklist. Work it top to bottom.

Wait for part two,