Database Security

A Complete 10-Chapter Security Course

Topics covered:

The infrastructure threat model · Authentication & access control

Least-privilege account design · Network security · Encryption at rest & in transit

Auditing & logging · Backup security · Database-specific hardening · Testing & checklist

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · the infrastructure half of the site's security coverage

Table of Contents

CHAPTER 1 OF 10

The Infrastructure Threat Model

Database Security

Chapter 1 · The Infrastructure Threat Model

The **SQL Injection** course covered attacks that reach the database *through* the application — untrusted input crossing the app's own trust boundary. This course covers a different question entirely: is the database *itself*, as a running piece of infrastructure, actually secured? Accounts, network exposure, encryption, backups, auditing — all of it orthogonal to whether the application's queries are parameterized correctly.

Two Threat Models, Not One

It's tempting to think "we fixed our SQL injection, our database is secure." That conflates two genuinely separate threat models:

	SQL INJECTION (ALREADY COVERED)	DATABASE SECURITY (THIS COURSE)
Attacker's path in	through the application's own queries	directly against the database as infrastructure
Trust boundary crossed	user input → query string	network perimeter, account credentials, storage media
Typical attacker	external, via a web form or API	external (exposed port), or an insider with legitimate access
Primary defence	parameterized queries (`sqli1-7`)	least privilege, network isolation, encryption, auditing

A perfectly parameterized application sitting on top of a database with a default admin password, exposed directly to the internet, is still a breach waiting to happen — the SQLi fix never touches any of that.

Insider Threats

SQLi's threat model assumes an *external* attacker working through the app. But a real, sizable share of data breaches come from people who already have **legitimate database access** — a disgruntled employee, a departing contractor whose account was never revoked, or simply a careless staff member with far broader access than their job needs. No amount of query parameterization defends against someone who can already run `SELECT * FROM customers` directly, because they were handed a working set of credentials.

Misconfigured Access

Default credentials left unchanged after installation, one shared "app" account used by every service and every developer, and overly broad grants ("just give it `ALL PRIVILEGES`, it's easier") are some of the most common — and most preventable — real-world causes of a database breach. Chapters 2 and 3 cover the fix in depth: proper authentication and access control, and least-privilege account design.

Physical & Network Exposure

A database doesn't have to be broken into if it was never actually locked down: a database port bound to `0.0.0.0` and reachable directly from the public internet, an unencrypted connection over a network where traffic can be intercepted, or a laptop with a local copy of production data that gets lost or stolen. None of these require any cleverness from an attacker — Chapter 4 (network security) and Chapter 6 (encryption in transit) cover exactly this ground.

Backup Theft

A backup is a complete copy of the data the live database protects — and it's routinely protected far *less* carefully than the live system itself: sitting in a general-purpose file share, an unencrypted cloud storage bucket, or an old server nobody remembers is still running. An attacker doesn't need to breach a well-defended live database at all if an unencrypted, loosely-guarded backup of the exact same data is sitting somewhere easier to reach. Chapter 8 covers backup security as its own dedicated topic, not an afterthought to the "real" database.

> **MANY REAL BREACHES NEVER TOUCHED A SINGLE QUERY**
>
> A striking number of major data exposures involved no SQL injection, no clever exploit, and no authentication bypass at all — just a database (commonly MongoDB, Elasticsearch, or a similar data store) left reachable on the public internet with authentication disabled, or a backup sitting in a publicly readable storage bucket. The `mongodb2-7` chapter of this site's MongoDB course covers one concrete, well-known example: a wave of ransomware attacks that specifically targeted exposed, unauthenticated MongoDB instances. The lesson generalizes far beyond MongoDB — the infrastructure threat model this course covers is not a theoretical add-on to application security; it's where a large share of real incidents actually originate.

What This Course Covers

CHAPTER	TOPIC
2	Authentication & Access Control
3	Least-Privilege Account Design
4	Network Security for Databases

CHAPTER	TOPIC
5	Encryption at Rest
6	Encryption in Transit
7	Auditing & Logging
8	Backup Security
9	Database-Specific Hardening
10	Testing, Pitfalls & Hardening Checklist

THIS COURSE'S THROUGHLINE

Every chapter that follows treats the database as **infrastructure to be secured**, not as a query target to be defended against malformed input. That's a genuinely different mindset from the SQLi course's — and, as the warning above shows, arguably just as consequential a gap when it's left unaddressed.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why "we use parameterized queries everywhere" does not mean a database is secure. Name at least three concrete attack paths this chapter covered that a SQLi fix does nothing to prevent.

View solution

EXERCISE 2

A company reports "no SQL injection vulnerabilities were found in our application" after a security review. List two realistic ways their database could still be breached despite that finding, using this chapter's threat categories.

View solution

EXERCISE 3

Explain why a database's backup can be a bigger security risk than the live database itself, even though it holds exactly the same data. What does this imply about where security effort should be spent?

View solution

CHAPTER 1 QUICK REFERENCE

- SQLi threat model = attacks reaching the DB **through the app**; Database Security threat model = attacks against the DB **as infrastructure itself**
- **Insider threats** — legitimate credential holders acting maliciously or carelessly; parameterized queries don't defend against this at all
- **Misconfigured access** — default credentials, shared accounts, over-broad grants (Chapters 2–3)
- **Physical/network exposure** — an internet-reachable DB port, an unencrypted connection, a lost device with local data (Chapters 4, 6)
- **Backup theft** — backups are often protected far less carefully than the live database holding the same data (Chapter 8)
- Many real breaches involve **no query exploit at all** — just an exposed, unauthenticated database or an unprotected backup
- **Next chapter:** Authentication & Access Control — database accounts vs. application-level auth, role-based access control in the DB itself

CHAPTER 2 OF 10

Authentication & Access Control

Database Security

Chapter 2 · Authentication & Access Control

Chapter 1 flagged misconfigured access as one of the most common, most preventable causes of a real breach. This chapter is where that gets fixed: the database's own account system — a genuinely separate authentication layer from the application-level login the **Authentication & Session Security** course (`bc1`) already covers.

Two Separate Authentication Layers

It's easy to think "we already covered authentication" after the `bc1` course — but that course is about *end users* logging into the *application*. The database has its own, entirely separate authentication layer: the application itself has to authenticate *to the database*, using its own credentials, regardless of who the end user is.

	APPLICATION-LEVEL AUTH (`BC1` COURSE)	DATABASE-LEVEL AUTH (THIS CHAPTER)
Who authenticates	the end user, via the app's login form	the application itself, via a database account
Credential type	a hashed password, a session/token (`bc1-2` , `bc1-7`)	a database username and password (or certificate)
Where it happens	the app's own login endpoint	the connection string the app uses to reach the DB
End user ever sees it?	yes — it's their login	no — entirely invisible to end users

An end user is never a database account. Every request an application handles — regardless of which end user made it — reaches the database as the *same* application-level account. That account's own security is this chapter's subject.

Database Accounts: Getting the Basics Right

A database account for an application should be created **specifically for that application** — never the database's own built-in superuser/root account, which exists for administration, not routine application traffic.

Chapter 1's warning about default credentials applies directly here: every database ships with (or defaults to) some form of administrative account, and leaving its factory password unchanged is one of the single most common real-world misconfigurations found in the wild.

```
-- creating a dedicated account for an application, not using the admin account
CREATE USER 'shop_app'@'%' IDENTIFIED BY 'a-genuinely-strong-password';
```

Role-Based Access Control (RBAC) in the Database

Creating an account only answers *who* can connect — **authorization** (distinct from authentication, the same distinction `bc1-1` drew for the application layer) answers *what* that account is actually allowed to do once connected. Most databases implement this through **roles** and `GRANT` / `REVOKE` statements.

```
-- granting only what the shop_app account actually needs
GRANT SELECT, INSERT, UPDATE, DELETE ON shop.* TO 'shop_app'@'%';

-- a separate, read-only account for a reporting job
GRANT SELECT ON shop.* TO 'shop_reporting'@'%';
```

This chapter introduces the concept; Chapter 3 goes deep on **designing** exactly which grants an account should have, including the "just grant everything, it's easier" anti-pattern this chapter's next section already starts flagging.

Avoiding Shared & Generic Accounts

A single shared account — "just use the `admin` login, everyone does" — used by every developer, every script, and every service is a genuinely common real-world pattern, and a seriously damaging one. Beyond the obvious blast-radius problem (one leaked password compromises everything that used it), a shared account destroys something Chapter 7 (auditing) depends on entirely: the ability to answer *who* did something.

	ONE SHARED ACCOUNT	INDIVIDUAL ACCOUNTS
Audit log shows	"admin ran this query" — for every person and service that ever used it	exactly which person or service ran it
Revoking one person's access	impossible without changing the password everyone relies on	revoke that one account; nothing else is affected
Blast radius of one leak	every user of the shared account is compromised	only that one account

"IT'S JUST FOR TESTING" ACCOUNTS HAVE A WAY OF BECOMING PERMANENT

A temporary shared account created "just to get the demo working" is exactly the kind of shortcut that quietly becomes permanent infrastructure — nobody circles back to remove it once real traffic depends on it. Treat any shared or generic database account as a finding to fix, not a convenience to tolerate, the moment it's noticed.

AUTHENTICATION ANSWERS WHO; AUTHORIZATION ANSWERS WHAT

The same distinction `bc1-1` drew for the application layer applies here too: **authentication** is the database confirming an account's identity (a valid username/password); **authorization** — RBAC, `GRANT` / `REVOKE` — is the database enforcing what that authenticated account is actually allowed to do. Getting authentication right but leaving every account with blanket privileges solves only half the problem — the half Chapter 3 goes deep on.

Hands-On Exercises

EXERCISE 1

Explain the difference between application-level authentication (the `bc1` course) and database-level authentication (this chapter). Does an end user's login ever directly authenticate them to the database? Why or why not?

View solution

EXERCISE 2

Write the SQL to create a database account called `reporting_job` and grant it read-only access to a database called `shop`. Explain why this should be a separate account rather than reusing the main application's account.

View solution

EXERCISE 3

A team of five developers all connect to the production database using one shared `admin` login. One developer leaves the company. Explain everything that goes wrong with this setup, tying your answer to both this chapter and the forward-reference to auditing (Chapter 7).

View solution

CHAPTER 2 QUICK REFERENCE

- Application-level auth (`bc1`) and database-level auth (this chapter) are **two separate layers** — an end user is never a database account
- Every application needs its **own dedicated database account** — never the database's built-in admin/root account

- Default/factory admin passwords left unchanged are one of the most common real-world misconfigurations
- **Authentication** = who can connect; **authorization (RBAC)** = what they're allowed to do once connected
- `GRANT` / `REVOKE` assign specific privileges to specific accounts — Chapter 3 covers designing these grants in depth
- **Shared/generic accounts** destroy accountability (can't tell who did what) and widen blast radius on a leak — treat them as a finding, not a convenience
- **Next chapter:** Least-Privilege Account Design — separate accounts per service, granular grants, avoiding the "superuser for everything" anti-pattern

CHAPTER 3 OF 10

Least-Privilege Account Design

Database Security

Chapter 3 · Least-Privilege Account Design

Chapter 2 introduced `GRANT` / `REVOKE` and the concept of database roles. This chapter goes deep on actually **designing** those grants — the discipline of giving every account exactly the privilege it needs, and not one grant more.

The Least-Privilege Principle

Least privilege means every account — human or service — gets the *minimum* set of permissions required to do its actual job, nothing broader "just in case" or "to save time." It's the same principle this site's Laravel and Django courses applied to mass-assignment allowlists, and the Auth course applied to role design — here it's applied directly to the database's own grant system.

One Account Per Service, Not One Account Per "The App"

Chapter 2's `reporting_job` example already hinted at this: a real system is rarely just "the app" as one monolithic thing. A web application, a background job processor, a reporting/analytics job, and an admin panel are genuinely different services with genuinely different needs — each should get its *own* database account, scoped to exactly what that service does.

SERVICE	WHAT IT ACTUALLY NEEDS
Web application	`SELECT`, `INSERT`, `UPDATE`, `DELETE` on its own tables
Background job processor	`SELECT` / `UPDATE` on a job queue table only
Reporting/analytics job	`SELECT` only, across whatever it reports on
Admin panel	broader access, but still scoped — not automatically the database's own root/superuser account

Granular GRANT/REVOKE

Privilege doesn't have to be all-or-nothing at the database level — most database engines support granting down to the **table** or even **column** level. A service that only needs a user's display name and avatar has no

legitimate reason to be able to `SELECT` a password hash column, even on the very same table.

```
-- column-level grant: only the columns this service actually needs
GRANT SELECT (id, display_name, avatar_url) ON users TO 'profile_widget'@'%';

-- narrowing an over-broad grant discovered during a review
REVOKE ALL PRIVILEGES ON shop.* FROM 'legacy_service'@'%';
GRANT SELECT ON shop.orders TO 'legacy_service'@'%';
```

Read-Only vs. Read-Write Roles

Rather than hand-crafting an ad-hoc grant list for every single account, defining explicit **roles** — a named, reusable bundle of privileges — keeps grants consistent and reviewable.

```
CREATE ROLE read_only;
GRANT SELECT ON shop.* TO read_only;

CREATE ROLE read_write;
GRANT SELECT, INSERT, UPDATE, DELETE ON shop.* TO read_write;

-- assigning a role to an account, rather than granting privileges one at a time
GRANT read_only TO 'reporting_job'@'%';
```

The "Superuser for Everything" Anti-Pattern

"Just grant it `ALL PRIVILEGES`, it's easier" is the single most common way least privilege gets abandoned in practice — and it turns every account into an equally attractive, equally catastrophic target.

	SUPERUSER-EQUIVALENT APP ACCOUNT	PROPERLY SCOPED ACCOUNT
If this account is compromised	attacker can read/modify/drop *any* table, in *any* database on the server	attacker is limited to exactly what this account was ever granted
Can it create new accounts?	often yes — potentially creating a persistent backdoor	no — account management is a separate, narrower privilege
Can it disable auditing (Ch.7)?	often yes	no

This is the exact same principle the MongoDB course's `mongodb2-7` chapter drew for MongoDB's own `root` versus scoped `readWrite` roles — least privilege isn't a MySQL-specific or Postgres-specific idea, it's a property every database engine's own account system is built to support, and every engine's documentation strongly recommends applying.

LEAST PRIVILEGE IS DEFENSE IN DEPTH AGAINST EVERY OTHER COURSE'S VULNERABILITIES TOO

If a SQL injection bug were ever to recur despite the `SQLi` course's defenses (Chapter 7 there, parameterized queries), the damage it can do is directly bounded by whatever privileges the compromised application's own database account happens to hold. A properly scoped account limits even a successful SQLi exploit to that account's own narrow grants; a superuser-equivalent account turns the exact same bug into a total database compromise. Least privilege doesn't prevent an application-layer bug from existing — it limits how much that bug can cost when one eventually does.

GRANTS ACCUMULATE — AUDIT THEM PERIODICALLY

A grant added "temporarily" to unblock a debugging session has a way of never being revoked once the immediate problem is solved. Over months or years, an account's actual privileges tend to drift wider than what it currently needs, even when it started out correctly scoped. Chapter 10's hardening checklist includes a periodic grant review specifically because this drift is the normal, expected outcome of not checking — not an unusual failure.

Hands-On Exercises

EXERCISE 1

A single "backend" database account is currently used by a web application, a background job processor, and a nightly reporting script. Propose a split into separate accounts, stating what each one should actually be granted.

View solution

EXERCISE 2

A "profile widget" service only ever displays a user's display name and avatar. Write the column-level `GRANT` that gives it exactly that access on a `users` table that also contains an `email` and a `password_hash` column.

View solution

EXERCISE 3

Explain, using this chapter's SQL-injection cross-reference, why least-privilege account design counts as "defense in depth" rather than a replacement for parameterized queries. What specifically does it limit, and what does it not prevent?

View solution

CHAPTER 3 QUICK REFERENCE

- **Least privilege** = every account gets the minimum privilege needed for its actual job, never "just in case" extra
- **One account per service**, not one account for "the app" as a whole — web app, job processor, reporting, and admin panel each get their own scoped account
- Grants can be **table- or column-level**, not just database-wide — restrict sensitive columns (password hashes) even from accounts that need the rest of the table
- **Roles** (`CREATE ROLE`) bundle reusable privilege sets, keeping grants consistent and reviewable across many accounts
- The **"superuser for everything"** anti-pattern turns every account into an equally catastrophic single point of failure
- Least privilege is **defense in depth** — it bounds the damage of a recurring SQLi or any other application bug, it doesn't prevent the bug itself
- **Grants drift wider over time** — periodic review belongs on the Chapter 10 hardening checklist, not left to chance
- **Next chapter:** Network Security for Databases — binding to private networks, firewall rules, never exposing a DB port to the public internet

CHAPTER 4 OF 10

Network Security for Databases

Database Security

Chapter 4 · Network Security for Databases

Chapter 1's warning box described a wave of ransomware attacks against databases exposed directly to the internet — no exploit, no injection, just a reachable port with nothing guarding it. This chapter is the deep dive on exactly that failure mode: making sure the network itself never gives an attacker a path to the database in the first place.

Binding to Localhost / Private Networks Only

A database's own configuration controls which network interfaces it listens on. Binding to `127.0.0.1` (localhost only) or a private network address means the database is physically unreachable from the public internet, regardless of any firewall — there's no route to it at all from outside.

```
# MySQL — my.cnf
[mysqld]
bind-address = 127.0.0.1

# PostgreSQL — postgresql.conf
listen_addresses = 'localhost'
```

When the application and the database run on genuinely separate machines, "localhost only" isn't an option — bind instead to the private network interface (a VPC-internal IP) that only other machines on that same private network can reach, never a public IP.

Firewall Rules

Binding to a private interface handles reachability from the *public* internet — a firewall handles reachability from *everything else*, including other machines on the same private network that still shouldn't have a path to the database port. The rule to apply: allowlist only the specific hosts (application servers, a bastion host) that genuinely need to connect, and deny everything else by default.

```
# ufw — only this one application server's IP may reach MySQL's port
sudo ufw allow from 10.0.1.15 to any port 3306
sudo ufw deny 3306
```

VPN & Bastion Host Access Patterns

Developers and DBAs still legitimately need occasional direct access — but that doesn't mean opening the database port itself to their laptops. The standard pattern: route human access through a **VPN** (the `vpn1` course covers setup) or a **bastion/jump host** — a single, tightly monitored server that itself is allowed to reach the database, which a person tunnels through via SSH.

```
# SSH local port forwarding through a bastion host
# (the exact mechanism remote_lesson_05 covers in the SSH course)
ssh -L 3306:db-internal-host:3306 user@bastion-host

# now connecting to localhost:3306 actually reaches the private database,
# tunnelled through the bastion — the DB port itself is never exposed
```

The database only ever needs to trust the bastion host's own IP in its firewall rules — individual developers' laptops, on changing IPs from home or a coffee shop, never need direct network access at all.

Never Expose a DB Port Directly to the Public Internet

Every rule above serves one central conclusion: a database port reachable from `0.0.0.0/0` (anywhere on the internet) is a mistake, full stop, regardless of whether authentication is enabled. It's not a matter of "probably fine since there's a password" — it's an unnecessary, entirely avoidable exposure of a valuable target to every scanner on the internet.

	PUBLIC-FACING DB PORT	PRIVATE NETWORK + BASTION
Who can even attempt to connect	anyone on the internet	only the application server(s) and the bastion host
Discovered by automated scanners?	routinely, often within minutes to hours	never — there's no public route to find
A leaked/weak password's impact	immediately exploitable by anyone who finds the port	useless without also being on the private network or through the bastion

INTERNET-WIDE SCANNERS FIND EXPOSED DATABASE PORTS WITHIN MINUTES

Automated internet-wide scanning tools continuously probe the entire public IP space for open, well-known database ports — this isn't a targeted attack against a specific organization, it's a standing background process constantly

sweeping the internet. A database port opened to `0.0.0.0/0`, even briefly during setup or testing, gets discovered far faster than most people expect — often before anyone even finishes configuring it. Chapter 1's ransomware example is the second half of this exact story: the scanner finds the open port first; whether authentication was ever properly configured is what determines what happens next. This chapter's job is making sure the first half — the port even being findable — never happens at all.

"WE'LL RESTRICT IT LATER" IS HOW EXPOSURE BECOMES PERMANENT

A database port opened to everywhere "just for initial setup, we'll lock it down after" is exactly the kind of shortcut — like Chapter 2's "temporary" shared account — that quietly becomes the permanent configuration once the setup task is marked done and attention moves elsewhere. Configure the bind address and firewall rule correctly from the very first deployment, not as a follow-up task.

Hands-On Exercises

EXERCISE 1

Explain the difference between binding a database to `127.0.0.1`, binding it to a private network IP, and binding it to `0.0.0.0`. Which of these, on its own, makes a database reachable from the public internet?

View solution

EXERCISE 2

A developer wants occasional direct access to a production database for debugging. Describe the bastion-host access pattern that lets them do this without ever opening the database's port to the public internet.

View solution

EXERCISE 3

A teammate argues "our database has a strong password, so exposing the port publicly for convenience is fine." Using this chapter's material, explain what's wrong with that reasoning.

View solution

CHAPTER 4 QUICK REFERENCE

- **Bind address** controls which network interfaces the database listens on at all — `127.0.0.1` (localhost) or a private IP is safe; `0.0.0.0` without a firewall is not

- **Firewall rules** allowlist only the specific hosts that genuinely need to connect — deny by default
- **VPN or bastion host** access lets humans reach the database without ever opening its port directly — tunnel via SSH (`ssh -L`, per the SSH course's `remote_lesson_05`) or a VPN (`vpn1` course)
- A database port reachable from **anywhere on the internet** is a mistake regardless of password strength — automated scanners find exposed ports within minutes to hours
- Configure network security **from first deployment** — "restrict it later" reliably becomes permanent, the same as Chapter 2's shared-account shortcut
- **Next chapter:** Encryption at Rest — full-disk encryption, transparent data encryption (TDE), column-level encryption, key management basics

CHAPTER 5 OF 10

Encryption at Rest

Database Security

Chapter 5 · Encryption at Rest

Chapter 4 secured the network path to a running database. This chapter addresses a different question entirely: what if someone gets the raw storage itself — a stolen drive, a leaked cloud snapshot, or (Chapter 8's subject) an unprotected backup file? Encryption at rest is the answer to that scenario specifically.

What "At Rest" Means & The Threat It Addresses

Data "at rest" is data sitting on disk — data files, backups, snapshots — as opposed to data moving across a network (**in transit**, Chapter 6's subject). The threat model here is specific: someone obtains the *physical or virtual storage medium itself*, bypassing the network and the application entirely — a stolen laptop or server, a decommissioned drive that was never wiped, or a cloud storage snapshot that leaked or was misconfigured to be publicly readable.

Full-Disk Encryption

Full-disk encryption (e.g. **LUKS** on Linux, or a cloud provider's built-in encrypted-volume option) encrypts the *entire* disk or volume at the operating-system level — completely transparent to the database running on top of it, requiring no application or query changes at all.

FULL-DISK ENCRYPTION DOESN'T PROTECT A RUNNING, UNLOCKED SERVER

Full-disk encryption protects data specifically when the disk is *powered off* or physically removed — a stolen drive is unreadable without the decryption key. It does **not** protect against someone with legitimate OS-level or database-level access to a *running* system: while the server is up, the disk is already unlocked and decrypted, and anything readable through the OS or database is readable in plain form. This is a defense against physical theft specifically, not against a compromised running system.

Transparent Data Encryption (TDE)

TDE moves encryption one level down from the OS into the database engine itself — data files, logs, and the backups the engine produces are encrypted, decrypting automatically and transparently ("transparent" is

literally the name) for any authenticated query. Available as a built-in or extension feature in most major engines (MySQL Enterprise, SQL Server, Oracle, PostgreSQL extensions).

TDE is closer to the data than full-disk encryption — it protects the database's own files specifically, even if they're copied out independently of the disk itself — but shares the exact same fundamental limitation: it decrypts automatically for any authenticated connection. A compromised database account, or a successful SQL injection, reads exactly the same plaintext TDE would have hidden from a stolen disk.

Column-Level Encryption

For especially sensitive individual fields — a Social Security number, a full credit card number — **column-level encryption** encrypts that specific value, typically at the application layer, such that even a legitimate, authenticated database query returns **ciphertext** unless the requesting code also independently holds the decryption key.

```
-- MySQL's built-in AES functions, illustrating the pattern
INSERT INTO customers (name, ssn_encrypted)
VALUES ('Dana', AES_ENCRYPT('123-45-6789', @encryption_key));

-- reading it back requires the SAME key, known only to authorized application code
SELECT name, AES_DECRYPT(ssn_encrypted, @encryption_key) FROM customers;
```

This is the one technique in this chapter that provides genuine defense against a compromised or overprivileged database account itself (Chapter 3's threat model) — a database account with full `SELECT` access still only ever sees ciphertext for that column without the separate encryption key. The trade-off: encrypted columns generally can't be indexed, searched, or sorted on directly by the database, since the stored value is no longer the real one.

	TDE / FULL-DISK ENCRYPTION	COLUMN-LEVEL ENCRYPTION
Protects against	stolen physical/virtual storage media	a compromised or overprivileged database account
Transparent to queries?	yes — fully automatic	no — requires the application to hold the key
Cost	essentially none	loses indexing/searching/sorting on that column

Key Management Basics

Every technique above is only as strong as how its encryption key is protected — and the single most common mistake is storing that key *right next to* the data it encrypts (a key in the same config file, the same server, the same repository), which defeats the entire purpose: whoever obtains the data also obtains the key needed to read it.

- **Never hardcode an encryption key** in application source or commit it to version control — the same discipline this site has stressed for database connection strings (`mongodb2-7`) and CI/CD secrets (`pipelines1-5`) applies directly to encryption keys.
- Use a **dedicated key management service** (a cloud KMS, HashiCorp Vault) that stores keys separately from both the application and the data, with its own access controls and audit trail.
- **Rotate keys periodically** — a leaked key that's rotated regularly has a bounded window of usefulness to an attacker, rather than being valid indefinitely.

> **ENCRYPTION WITHOUT REAL KEY MANAGEMENT IS JUST OBFUSCATION**
>
> Encrypting data next to a key anyone with access to that data can also read isn't meaningfully different from not encrypting it at all — it adds a technical step, not a real security boundary. The genuine security value of encryption at rest comes entirely from the key being stored, managed, and access-controlled *separately* from the data it protects.

Hands-On Exercises

EXERCISE 1

Explain why full-disk encryption protects against a stolen server but does not protect against a compromised database account on a running system. What specifically is different about the two scenarios?

View solution

EXERCISE 2

A team stores customer credit card numbers as plain columns, protected only by TDE. A developer with legitimate read access to the `customers` table (per Chapter 3's least privilege) can still read every card number in plaintext. Propose the fix from this chapter, and explain the trade-off it introduces.

View solution

EXERCISE 3

A company encrypts a sensitive column, but stores the encryption key in the same application config file as the database connection string, both committed to the same private Git repository. Explain why this setup provides little real security benefit.

View solution

CHAPTER 5 QUICK REFERENCE

- **At rest** = data on disk (files, backups, snapshots); **in transit** = data moving over a network (Chapter 6)
- **Full-disk encryption** protects a powered-off/stolen drive; useless against a compromised running system, where the disk is already unlocked
- **TDE** encrypts the database engine's own files/backups transparently — same fundamental limitation as full-disk encryption: decrypts automatically for any authenticated query
- **Column-level encryption** is the one technique that defends against a compromised/overprivileged DB account itself — at the cost of losing indexing/search/sort on that column
- **Never store an encryption key next to the data it protects** — use a dedicated key management service, never a hardcoded key in config or source
- Encryption without real key management is **obfuscation, not security**
- **Next chapter:** Encryption in Transit — TLS for database connections, certificate verification, why an unencrypted connection over a network is a real risk

CHAPTER 6 OF 10

Encryption in Transit

Database Security

Chapter 6 · Encryption in Transit

Chapter 5 protected data sitting on disk. This chapter protects the other half of the data's lifecycle — while it's *moving* across the network between an application and its database. The underlying cryptography is identical to the **HTTPS/TLS** course (`https1`) already on this site; this chapter is that same TLS applied to a database's own wire protocol instead of HTTP.

Why an Unencrypted DB Connection Is a Real Risk

Chapter 4's private-network binding controls *who can reach* the database's port — it says nothing about whether the traffic crossing that network, once a legitimate connection is allowed, can be observed by something in between. Even inside a "private" network, traffic still crosses physical switches, routers, and — in a cloud environment — a shared hypervisor layer other tenants' workloads also run on. An unencrypted connection means the **login credentials** and **every query and response**, including sensitive data, travel across that path in plain, readable text.

TLS for Database Connections

The same TLS mechanics the HTTPS course covered in depth — the handshake (`https1-6`), certificates (`https1-4`), symmetric and asymmetric cryptography (`https1-2` / `https1-3`) — apply directly here, just carrying a database's own wire protocol instead of HTTP. Every major database engine supports requiring TLS for connections.

```
-- PostgreSQL connection string requiring TLS
postgresql://user:pass@dbhost/mydb?sslmode=require

# MySQL client requiring TLS
mysql --ssl-mode=REQUIRED -h dbhost -u shop_app -p

// MongoDB connection string (per mongodb2-7)
mongodb+srv://user:pass@cluster.example.net/mydb?tls=true
```

Certificate Verification: The Part That's Easy to Get Wrong

Enabling TLS on its own only guarantees the connection is *encrypted* — it does not guarantee the server on the other end is actually the real database server, unless the client also **verifies** the server's certificate. This is the exact same certificate-chain-of-trust concept the HTTPS course covered (`https1-5`), and getting it wrong in a database connection string is a genuinely common, well-documented pitfall.

POSTGRESQL `SSLMODE`	ENCRYPTS?	VERIFIES THE SERVER'S CERTIFICATE?
require	yes	no
verify-ca	yes	yes — cert signed by a trusted CA
verify-full	yes	yes — cert signed by a trusted CA *and* matches the hostname

SSLMODE=REQUIRE ENCRYPTS THE CHANNEL — IT DOES NOT CONFIRM WHO'S ON THE OTHER END

`sslmode=require` is the setting most tutorials reach for first, since it's the shortest path to "TLS is now on." But it only encrypts the connection against passive eavesdropping — it does **not** verify the server's certificate at all, which means an attacker able to intercept the connection (a man-in-the-middle) can present *any* certificate, even a self-signed or unrelated one, and the client will happily proceed with an "encrypted" connection to the attacker instead of the real database. Only `verify-ca` or, better, `verify-full` actually confirm the server's identity — the same distinction between "encrypted" and "encrypted *and* authenticated" that `https1-3` drew for HTTPS generally.

Why This Matters Even on a "Private" Network

Chapter 4's network isolation and this chapter's encryption in transit are two **independent** layers, not substitutes for each other — the same defense-in-depth principle Chapter 3 established for least privilege. Network isolation reduces *who can attempt a connection at all*; encryption in transit protects the data *within* a connection that's already permitted, against anything positioned to observe that traffic — a compromised intermediate host, a misbehaving piece of shared cloud infrastructure, or an insider (Chapter 1) already present on the same network segment. Neither layer alone covers what the other one does.

THE CRYPTO IS THE SAME COURSE, JUST A DIFFERENT PROTOCOL

If any of this chapter's TLS concepts feel unfamiliar, the `https1` course covers every one of them in much greater depth — the handshake, certificate chains, cipher suites, symmetric versus asymmetric cryptography. Nothing about TLS changes fundamentally when it protects a database connection instead of a browser's HTTP request; only the specific configuration flags (`sslmode` , `--ssl-mode` , `tls=true`) differ from one engine to the next.

Hands-On Exercises

EXERCISE 1

Explain why a database connection on a "private" network can still benefit from TLS, even though Chapter 4 already restricts who can reach the database's port at all.

View solution

EXERCISE 2

A team configures their PostgreSQL connection string with `sslmode=require` and considers the connection fully secure. Explain what protection this setting is still missing, and which setting would fix it.

View solution

EXERCISE 3

Using what you already know from the HTTPS/TLS course, explain in your own words what a "man-in-the-middle" attack against a database connection would look like, and why certificate verification specifically (not just encryption) is what prevents it.

View solution

CHAPTER 6 QUICK REFERENCE

- **In transit** = data moving across a network, between an application and the database — distinct from Chapter 5's "at rest"
- Even a **private network** can be observed — shared cloud infrastructure, a compromised intermediate host, or an insider already on the network
- TLS for databases uses the **exact same mechanics** as the `https1` course — handshake, certificates, symmetric/asymmetric crypto — applied to a different wire protocol
- **Encryption alone isn't enough** — `sslmode=require` encrypts but does not verify the server's certificate, leaving the connection open to a man-in-the-middle
- `verify-ca` / `verify-full` (or the equivalent in other engines) actually **confirm the server's identity**, not just encrypt the channel
- Network isolation (Ch.4) and encryption in transit (this chapter) are **independent layers** — neither replaces the other
- **Next chapter:** Auditing & Logging — query logging, audit trails for sensitive table access, detecting anomalous access patterns

CHAPTER 7 OF 10

Auditing & Logging

Database Security

Chapter 7 · Auditing & Logging

Chapter 2 argued that individual accounts are worthless for accountability without something actually recording what each one does. This chapter is that missing piece — and the database-specific instance of a category the OWASP course already named: **A09 Security Logging & Monitoring Failures** (`owasp1-9`).

Why Auditing Depends on Chapter 2's Foundation

Auditing only works if it's built on individual, non-shared accounts (Chapter 2) — a log that says "admin ran this query" for five different developers tells you nothing useful at all. This chapter assumes that foundation is already in place and builds the actual logging and detection on top of it.

Query Logging

Most database engines can log every query, along with which account ran it and when — MySQL's general query log, PostgreSQL's `log_statement` , MongoDB's `auditLog` . Logging *everything* is expensive in both storage and performance overhead, which is why most real deployments scope logging deliberately rather than capturing every single statement.

```
-- PostgreSQL: log every data-modifying statement (not every SELECT)
ALTER SYSTEM SET log_statement = 'mod';

-- MySQL: enabling the general query log (expensive — scope carefully)
SET GLOBAL general_log = 'ON';
```

Audit Trails for Sensitive Table Access

Rather than logging every query against every table, a more sustainable approach specifically tracks access to **sensitive** tables — a `customers` or `payments` table, for instance — using dedicated audit tooling (MySQL Enterprise Audit, PostgreSQL's `pgAudit` extension) or triggers that record who read or modified which rows.

	LOGGING EVERYTHING	TARGETED AUDIT LOGGING
Storage & performance cost	high — every query, every table	manageable — scoped to what actually matters
Signal-to-noise ratio	low — buried in routine traffic	high — sensitive-table access stands out
Sustainable long-term?	rarely — usually disabled again once it slows things down	yes

Detecting Anomalous Access Patterns

Logs that just accumulate and are never reviewed provide little real protection — the value is in noticing when something looks **wrong**, ideally close to when it happens. A few concrete anomaly signals worth watching for:

- A **read-only** account (Chapter 3's `reporting_job`) suddenly issuing an `UPDATE` or `DELETE` — either a bug, or a sign that account's credentials are being misused.
- A sudden **spike in query volume** from one account, especially outside its normal usage hours.
- An account **reading far more rows** than its typical pattern — a common signature of data exfiltration in progress.

THIS IS A09, APPLIED TO THE DATABASE SPECIFICALLY

`owasp1-9` already covered insufficient logging, tampered logs, and the absence of real-time monitoring/alerting at the *application* layer. Everything in that chapter applies here too, just aimed at the database's own activity instead of the app's — the database is simply another system that needs logs which are actually complete, actually protected, and actually watched.

AN ATTACKER WITH DATABASE ACCESS MAY TRY TO ERASE THE EVIDENCE FIRST

If an attacker gains genuine database access — through a compromised account, a recurring SQLi bug, or any other path this course has covered — one of their first moves is often covering their tracks: deleting or altering the very audit log meant to catch them, exactly the "tampered logs" failure `owasp1-9` warned about. Audit records need to be shipped to a separate, append-only or write-once system that a compromised database account cannot itself modify — logging to a table inside the same database being audited means the audit trail is only as trustworthy as the database it's supposed to be watching.

Hands-On Exercises

EXERCISE 1

Explain why Chapter 2's individual-accounts requirement is a prerequisite for auditing to be meaningful. What specifically does a shared account's audit log fail to tell you?

View solution

EXERCISE 2

A read-only reporting account (per Chapter 3's least-privilege design) suddenly issues a `DELETE` statement. Even though the account's own database-level privileges should prevent it from succeeding, explain why this event is still worth alerting on immediately.

View solution

EXERCISE 3

Explain why storing audit logs inside the same database they're auditing is a weaker design than shipping them to a separate, append-only system. What specific attack does this protect against?

View solution

CHAPTER 7 QUICK REFERENCE

- Auditing is only meaningful **on top of** Chapter 2's individual accounts — a shared account's log can't identify who actually did something
- **Query logging** (general/audit logs) records every statement, by account, but has real storage/performance overhead — scope it deliberately
- **Targeted audit trails** on specifically sensitive tables (customers, payments) are more sustainable than logging everything
- **Anomalous access patterns** to watch for: a read-only account attempting writes, a query-volume spike, an account reading unusually many rows
- This chapter is **A09 (OWASP)** applied specifically to the database — `owasp1-9` 's insufficient-logging/no-monitoring failures apply here directly
- **Audit logs must be tamper-resistant** — shipped to a separate, append-only system a compromised DB account can't itself alter or delete
- **Next chapter:** Backup Security — encrypting backups, secure storage/access, "your backups are also an attack target"

CHAPTER 8 OF 10

Backup Security

Database Security

Chapter 8 · Backup Security

Chapter 1 named backup theft as its own threat category, distinct from attacking the live database directly. This chapter is the deep dive: encryption, storage, retention, and the uncomfortable truth that a backup which has never been restored isn't actually a verified backup at all.

Backups Are a Complete Copy of Your Data

Every protection this course has covered — least privilege (Ch.3), network isolation (Ch.4), encryption (Ch.5–6), auditing (Ch.7) — has been about the *live* database. A backup is a complete, standalone copy of the exact same data, and by default, none of those protections automatically travel with it. Treating backup security as an afterthought means building a well-defended live system with an equally sensitive, far less defended copy sitting somewhere else.

Encrypting Backups

Chapter 5 noted that TDE encrypts the backups the database engine itself produces — but a huge share of real-world backups are made with plain dump tools (`mysqldump`, `pg_dump`) rather than the engine's own encrypted backup mechanism, and those tools produce **plain-text SQL by default**, regardless of whether TDE is enabled on the live database at all.

```
# A common real gotcha: this backup is plain, readable SQL — even if TDE is on
mysqldump shop_db > backup.sql

# Encrypting the dump directly, piped so it's never written to disk unencrypted
mysqldump shop_db | gpg --encrypt --recipient backups@example.com > backup.sql.gpg
```

The dump tool's own encryption status has nothing to do with the live database's TDE setting — they're entirely separate mechanisms, and it's easy to assume "we have TDE, so we're covered" while a completely unencrypted `mysqldump` output sits on a backup server.

Secure Backup Storage & Access

The exact same principles Chapters 3 and 4 applied to the live database apply directly to wherever backups are stored: least privilege (only the specific accounts that genuinely need backup access should have it — not "the whole ops team by default"), and network security (a backup sitting in a publicly readable cloud storage bucket is exactly as exposed as a database port open to the internet, just discovered differently).

Retention Policy Trade-offs

How long to keep backups is a genuine trade-off, not a "more is always safer" decision:

- **Keeping backups too long** means more copies of sensitive data sitting around, each one a separate attack surface — and can conflict with privacy/compliance obligations if data that should have been deleted per policy still exists in an old backup.
- **Keeping backups too short** risks being unable to recover from a problem discovered late — a slow data-corruption bug, or a breach not noticed for weeks, can mean the only "clean" backup has already aged out and been deleted.

A deliberate retention policy — balancing both risks rather than defaulting to either extreme — belongs on Chapter 10's hardening checklist.

"Your Backups Are Also an Attack Target"

> **RANSOMWARE ROUTINELY TARGETS BACKUPS FIRST, DELIBERATELY**
>
> A well-documented real-world ransomware tactic is to locate and destroy or encrypt an organization's **backups first**, specifically to remove any ability to recover without paying, before ever touching the live system. If backups are reachable from the same network the live database sits on, with the same (or weaker) access controls, an attacker who compromises one often compromises both in the same incident — turning what should have been a recoverable event into a total loss. Backup storage deserves its own access controls and network isolation, not an inherited assumption that "it's just a copy, it's fine."

Testing Restore Procedures

A backup that has never actually been restored is an **assumption**, not a verified backup — silent corruption, an incomplete dump, a misconfigured schedule, or a permissions issue can all mean months of "successful" backup jobs produced files that don't actually restore to a working database.

> **AN UNTESTED BACKUP ONLY REVEALS WHETHER IT WORKS WHEN IT'S TOO LATE**
>
> The worst possible moment to discover a backup doesn't actually restore is during a real incident, when it's needed most and there's no time left to fix it. Scheduled, regular test restores — into an isolated environment, verified against

expected data — are the only way to know a backup process genuinely works, rather than simply assuming it does because the job reports "success" every night.

Hands-On Exercises

EXERCISE 1

A team believes their backups are secure because their database has TDE enabled. Explain why a nightly `mysqldump` backup could still be a serious risk despite this, and propose a fix.

View solution

EXERCISE 2

Explain why a ransomware attacker specifically targets backups before (or instead of) the live database. What does this imply about where backups should be stored relative to the live system?

View solution

EXERCISE 3

A company has run nightly backups successfully (per the job's own logs) for two years but has never restored one. Explain why this doesn't actually guarantee they have a working disaster-recovery plan, and what should change.

View solution

CHAPTER 8 QUICK REFERENCE

- A backup is a **complete copy** of the live database's data — none of the other chapters' protections travel with it automatically
- **Dump tools (mysqldump, pg_dump) produce plain-text output by default** — TDE on the live database does not automatically encrypt these; encrypt the dump itself (e.g. piping through `gpg`)
- Backup storage needs its own **least privilege (Ch.3) and network isolation (Ch.4)** — not inherited assumptions from the live system
- **Retention is a trade-off**: too long = more exposed copies + compliance risk; too short = can't recover from a late-discovered incident
- Ransomware attackers routinely **target backups first** to remove recovery options before demanding payment
- An **untested backup is an assumption**, not a verified recovery plan — regular scheduled test restores are the only real verification

- **Next chapter:** Database-Specific Hardening — disabling unnecessary features, removing default accounts/databases, patching and CVE management

CHAPTER 9 OF 10

Database-Specific Hardening

Database Security

Chapter 9 · Database-Specific Hardening

Accounts, network, encryption, and auditing are all now covered. This chapter closes the remaining gap: is the database *software itself*, as installed, configured defensively — or just left running on whatever the installer defaulted to?

Disabling Unnecessary Features & Extensions

Every enabled feature is additional attack surface, whether or not it's ever actually used. MySQL's `LOAD DATA LOCAL INFILE` — a known vector that, combined with certain application patterns, can let a malicious server read arbitrary files off a connecting client — is a textbook example: a genuinely useful feature for specific bulk-import use cases, and a real risk left enabled everywhere else. The same logic applies to unused stored-procedure languages, unused plugins, and unused extensions generally: if it isn't actively used, disable it.

```
-- checking and disabling a feature that isn't actually needed
SHOW VARIABLES LIKE 'local_infile';
SET GLOBAL local_infile = 0;
```

Removing Default & Sample Accounts and Databases

Many database engines have historically shipped with sample databases and demonstration accounts out of the box — MySQL's old sample `test` database is a well-known example. These are **publicly documented**; an attacker doesn't need to guess they might exist, they already know exactly what to look for and where. Removing them after installation is a five-minute task with essentially no downside.

```
DROP DATABASE IF EXISTS test;
DROP USER IF EXISTS ''@'localhost'; -- the old MySQL "anonymous" account
```

Secure Default Configuration Review

A database engine's out-of-the-box configuration is typically optimized for "gets running with minimal friction," not "secure by default" — the same theme `owasp1-5` (Security Misconfiguration) covered generally.

Reviewing the actual configuration file against a published hardening guide (the **CIS Benchmarks** publish one for most major database engines) rather than trusting factory defaults is the concrete version of that OWASP category applied specifically to the database.

Patching & CVE Management

`owasp1-6` (Vulnerable & Outdated Components) already covered the general risk of running software with known, weaponized CVEs — the database engine itself is exactly this kind of component, not a special exception. A publicly known vulnerability with a documented exploit against an old, unpatched database version is very often a *far* easier path in than any custom attack — no creativity required, just checking the version number and looking up whether it's still vulnerable.

	UNPATCHED, KNOWN-VULNERABLE VERSION	CURRENT, PATCHED VERSION
Attacker effort required	minimal — a public exploit already exists	none of the known paths work
Discoverability	version banners/fingerprinting reveal it instantly	same fingerprinting reveals nothing exploitable

A KNOWN CVE IS A PUBLISHED ROADMAP, NOT A SECRET

Once a database vendor publishes a CVE and a patch, the vulnerability details are public — anyone can look up exactly what's broken in unpatched versions and how to exploit it. Running an old version isn't a matter of "obscure enough that no one will bother" — it's leaving a documented, searchable weakness live and waiting, indefinitely, until it's patched.

THIS CHAPTER IS TWO OWASP CATEGORIES, APPLIED TO THE DATABASE ENGINE

`owasp1-5`'s Security Misconfiguration and `owasp1-6`'s Vulnerable & Outdated Components both already covered these exact failure modes generally. This chapter isn't a new idea — it's confirming those two categories get applied to the database software itself, not just to application dependencies and web-server configuration.

Hands-On Exercises

EXERCISE 1

Explain why a feature like MySQL's `LOAD DATA LOCAL INFILE` should be disabled if it isn't actively used, even though it isn't inherently a bug. What general principle does this illustrate?

View solution

EXERCISE 2

Explain why a default sample database or demo account is more dangerous than a custom, undocumented misconfiguration would be. What does "publicly documented" change about the risk?

View solution

EXERCISE 3

Explain how this chapter's patching/CVE material relates to `owasp1-6`. Is a database engine a fundamentally different kind of risk than an outdated application dependency, or the same category applied to different software?

View solution

CHAPTER 9 QUICK REFERENCE

- **Disable unused features/extensions** — every enabled feature is attack surface, whether or not anyone uses it (e.g. MySQL's `LOAD DATA LOCAL INFILE`)
- **Remove default/sample accounts and databases** — these are publicly documented, well-known targets, not obscure risks
- Vendor defaults optimize for **ease of setup, not security** — review configuration against a published hardening guide (e.g. CIS Benchmarks)
- The database engine is a **component** like any other — `owasp1-6`'s outdated-components risk applies to it directly, not just application dependencies
- A known, unpatched CVE is a **public roadmap** for an attacker, not an obscure risk
- This chapter = **owasp1-5 + owasp1-6**, applied specifically to the database engine itself
- **Next chapter:** Testing, Pitfalls & Hardening Checklist — the closing chapter, following this curriculum's established format, with a deployable checklist

CHAPTER 10 OF 10

Testing, Pitfalls & Hardening Checklist

Database Security

Chapter 10 · Testing, Pitfalls & Hardening Checklist

The finale turns nine chapters of principles into practice: how to actually *verify* a database is secure rather than assume it, the broken-practice patterns to recognize, and a single deployable checklist pulling everything together. (Test only systems you own or are authorized to assess.)

How to Test Your Database's Security

1. **Verify account design** — list every database account; confirm none are shared or generic (Ch.2), and that each one's grants match least-privilege design, not a "superuser for everything" default (Ch.3).
2. **Verify network exposure** — attempt to connect to the database port from outside its private network; it should fail. Confirm the bind address and firewall rules actually match what's documented (Ch.4).
3. **Verify encryption** — confirm at-rest encryption (TDE or full-disk) is genuinely enabled, and that ad-hoc dump tools are independently encrypted too (Ch.5). Confirm connections require TLS with full certificate verification, not just `sslmode=require` (Ch.6).
4. **Verify auditing** — confirm sensitive-table access is actually logged, and that logs are shipped somewhere a compromised database account can't reach or alter (Ch.7).
5. **Verify backups** — perform an actual test restore, not just check that the backup job reports success. Confirm backup storage has its own least-privilege and network isolation, separate from the live system (Ch.8).
6. **Verify hardening** — check for lingering default accounts or sample databases, unused features left enabled, and confirm the engine version has no unpatched known CVEs (Ch.9).

The Broken-Practice Catalogue

BROKEN PRACTICE	WHY IT FAILS	CHAPTER
"We fixed our SQLi, so we're secure"	conflates the app-layer and infrastructure threat models	1
Shared "admin" login for the whole team	no accountability; can't revoke one person without breaking everyone	2

BROKEN PRACTICE	WHY IT FAILS	CHAPTER
App account granted `ALL PRIVILEGES` "to be safe"	turns every account into an equally catastrophic single point of failure	3
DB port open to `0.0.0.0` "just for now"	found by automated scanners within minutes to hours	4
`sslmode=require`, called "encrypted"	doesn't verify server identity — still MITM-able	6
Audit log stored inside the DB it audits	an attacker with DB access can simply delete the evidence	7
`mysqldump` left unencrypted "because TDE is on"	dump tools don't inherit the live database's TDE	8
Backups on the same access/network as the live DB	ransomware takes both in a single incident	8
Sample databases/default accounts never removed	publicly documented, automatable targets	9
Running a years-old DB version, "never been a problem"	a known CVE is a public roadmap, not an obscure risk	9

The Hardening Checklist

- ☑ **Individual accounts only** — no shared or generic logins, ever (Ch.2)
- ☑ **Least privilege per service** — one account per service, narrowest grants that still work, reviewed periodically for drift (Ch.3)
- ☑ **Bind to private/localhost only + firewall allowlist** — human access via bastion/VPN; never a database port open to the public internet (Ch.4)
- ☑ **Encrypt at rest** (TDE/full-disk) **and independently encrypt manual dumps** — don't assume one covers the other (Ch.5)
- ☑ **Require TLS with full certificate verification** (e.g. `verify-full`), not just an encrypted-but-unverified connection (Ch.6)
- ☑ **Audit sensitive-table access**, shipped to a separate, append-only system the database itself can't tamper with (Ch.7)
- ☑ **Encrypt and isolate backups**, and actually test-restore them on a schedule — a backup that's never been restored is an assumption (Ch.8)
- ☑ **Remove default accounts/sample databases**, disable unused features, and patch the engine promptly (Ch.9)

- [x] **Treat the infrastructure threat model as seriously as the application-layer one** — neither substitutes for the other (Ch.1)

THE ONE PRINCIPLE THAT SURVIVES THE WHOLE COURSE

Nothing in this course replaces anything else on this site — it's the missing half. A perfectly parameterized application (the `SQLi` course's whole point) running on a database with default credentials, an internet-facing port, and unencrypted backups is still one incident away from disaster. Equally, a flawlessly locked-down database serving an application riddled with SQL injection is just as exposed — the attacker never needed to touch anything this course covers at all. Chapter 1's two threat models both need real, ongoing attention; neither one is optional, and neither is ever "done" as a one-time setup task.

How the Course Fits Together

The arc: Chapter 1 drew the line between the application-layer threat model (`SQLi`, already covered) and this course's infrastructure threat model. Chapters 2–3 secured *who* can connect and *what* they can do. Chapter 4 secured the network path. Chapters 5–6 secured the data itself, at rest and in transit — reusing the exact cryptography the `https1` course already covered in depth. Chapter 7 made sure activity is actually recorded and tamper-resistant, directly extending `owasp1-9`. Chapter 8 extended every earlier protection to backups specifically. Chapter 9 hardened the software itself, applying `owasp1-5` and `owasp1-6` to the database engine. This chapter closes the loop: verify all of it actually holds, not just that it was configured once.

Hands-On Exercises

EXERCISE 1

Write a database security test plan for a single production database: what you'd check for accounts, network exposure, encryption, auditing, backups, and hardening, and what a passing result looks like for each.

View solution

EXERCISE 2

Audit this setup against the broken-practice catalogue: a single shared "admin" account is used by the whole team, the database port is open to `0.0.0.0` "temporarily," nightly `mysqldump` backups are unencrypted because "TDE is on," and the database still has its default sample schema. List every flaw, its chapter, and the fix.

View solution

EXERCISE 3

Explain why "our application passed its SQL injection review" and "our database is secure" are two different claims, using this course's material. What would a company need to verify in addition to the SQLi review to justify the second claim?

View solution

CHAPTER 10 QUICK REFERENCE

- **Test, don't assume**: verify accounts, network exposure, encryption, auditing, backups, and hardening — each independently
- **Broken practices**: shared logins · superuser app accounts · public DB ports · unverified TLS · in-DB audit logs · unencrypted dumps · backups sharing the live system's access · unremoved defaults · unpatched versions
- **Checklist**: individual accounts · least privilege · network isolation · encryption at rest + in transit · tamper-resistant auditing · encrypted/isolated/tested backups · hardened, patched software · both threat models taken seriously
- **The one principle**: application-layer security (SQLi, XSS, Auth) and infrastructure-layer security (this course) are independent, complementary layers — neither substitutes for the other
- Security here is **never a one-time setup task** — grants drift, versions age, backups go untested unless periodically re-verified

★ Database Security Complete — 10 / 10 chapters

From the infrastructure-vs-application threat model split, through authentication and least-privilege account design, network security, encryption at rest and in transit, auditing, backup security, and database-specific hardening, to a deployable testing and hardening checklist. Paired with SQL Injection, HTTPS/TLS, and the OWASP Top 10, this completes the site's database-focused security coverage — the infrastructure half that application-layer defenses alone were never going to cover.