

BE SOFTWARE ARCHITECTURE WISE · CHAPTER 8

Security, Privacy and Compliance by Design

Security and privacy are qualities of a system's structure, decided by where trust boundaries lie, who may do what and where data travels. This chapter covers the principles of secure design, threat modelling with data-flow diagrams, STRIDE and attack trees, identity and access with OAuth 2.0 and OpenID Connect, encryption, key management and secrets, zero trust between services, software supply-chain security, privacy by design under the GDPR, and security in operation. It is for developers and architects who know the distributed and data architectures of Chapters 6 and 7 and must now make them safe.

bewiseapps.com/software-architecture-wise/security-by-design/ · Version 2026-11-03. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Apply the CIA triad, Saltzer and Schroeder's design principles, defence in depth and secure defaults to architectural decisions, and explain what architects own in a secure development lifecycle.
- Threat-model a design using a data-flow diagram with trust boundaries, STRIDE and attack trees, and rate and record the resulting risks and mitigations.
- Distinguish authentication from authorisation, choose between sessions and tokens, and explain OAuth 2.0 with PKCE, client credentials, OpenID Connect and RBAC, ABAC and relationship-based access control.
- Decide what to encrypt, where and with which keys, using envelope encryption, a KMS, password hashing, secrets management and tokenisation.
- Explain zero trust architecture and design service-to-service security with mutual TLS, workload identity and policy enforcement points.
- Reduce software supply-chain risk with SBOMs, the SLSA framework, signing, pinning and isolated builds, and summarise the obligations of the EU Cyber Resilience Act.
- Design for the GDPR's principles, data protection by design and by default, and data subjects' rights, including erasure across backups, caches and event logs.
- Plan security logging, detection, vulnerability management and incident response, including the GDPR's 72-hour breach notification.

Before you start

Chapters 6 and 7, especially microservices, event-driven architecture and data ownership; Chapter 2's section on security as a quality attribute; Chapter 0's section on networks, HTTPS and TLS.

Time

About 4 hours to read; 45 minutes for the test.

Key terms

trust boundary · least privilege · defence in depth · threat modelling · STRIDE · OAuth 2.0 · OpenID Connect · PKCE · envelope encryption · zero trust · mutual TLS · SBOM · pseudonymisation · crypto-shredding

Security is a property of the structure

Most serious breaches are not caused by an exotic cryptographic attack. They come from structural decisions: an internal service that trusted every caller on the network, an administrator API that skipped an authorisation check, a database backup copied to a storage bucket that anyone could read, an access key committed to a repository, a dependency nobody knew the system used. Each of these was an architectural choice, usually made implicitly,

about where trust lies, which component may do what and where data travels. That is why this chapter treats security as a quality attribute designed into the structure, in the sense of Chapter 2's security scenarios, rather than as a list of products bought at the end.

It is also why security is inseparable from the styles of Chapters 6 and 7. Every service boundary, queue, cache, replica and analytics copy adds to the attack surface and creates another place where data must be protected. Distribution multiplies the number of trust boundaries; data architecture decides how many copies of personal data exist and how long they live. The running example is the food-delivery platform used throughout the course: customers, restaurants and couriers, an API gateway, ordering, payments, dispatch and notification services, an event stream and a data warehouse.

How the eight sections fit together

The first four sections build the core of secure design. **Security principles** sets out the vocabulary: the CIA triad, Saltzer and Schroeder's 1975 design principles such as least privilege and fail-safe defaults, defence in depth, secure defaults, the OWASP Top 10 and a secure development lifecycle based on NIST's SSDF. **Threat modelling** turns those principles into a repeatable design activity: draw a data-flow diagram with its trust boundaries, ask what can go wrong with STRIDE and attack trees, rate the risks and decide on mitigations. **Identity and access** covers who is calling: passwords as NIST now recommends, passkeys, sessions and tokens, OAuth 2.0 with PKCE, OpenID Connect, and the authorisation models RBAC, ABAC and relationship-based access control. **Protecting data** covers encryption in transit and at rest, password hashing, key management with envelope encryption, secrets, tokenisation and the decision about what to encrypt where.

The second four sections widen the view. **Zero trust** replaces the idea of a trusted internal network with per-request authentication and authorisation between services, using mutual TLS, certificates and policy decision and enforcement points. **Supply-chain security** treats dependencies, build systems and container images as part of the attack surface, with software bills of materials, the SLSA framework, signing and the EU Cyber Resilience Act. **Privacy by design** explains how the GDPR and the UK GDPR shape architecture: data minimisation, pseudonymisation, data subjects' rights, retention and erasure across backups and event logs. **Security in operation** closes the loop with security logging, detection, vulnerability management, incident response and breach notification.

What architects decide, and what they delegate

An architect does not need to be a penetration tester or a cryptographer, and should not invent cryptography. The architect's job is to make the decisions that are expensive to change and that no single team can make alone: where the trust boundaries lie, which identity provider and

token model the whole platform uses, which component is the policy decision point for which resources, which data is classified as restricted and therefore encrypted with which keys, how services prove their identity to one another, and how the organisation will detect and respond to an attack. These decisions belong in architecture decision records and in the cross-cutting concepts of an arc42 description, like any other significant decision.

Developers own a different, equally important set of concerns: validating input, encoding output, using the platform's authentication and authorisation libraries correctly, handling errors without failing open, keeping dependencies up to date and writing tests for security behaviour. The two meet in threat modelling, in secure design reviews and in automated checks in the delivery pipeline. Where practitioners disagree, for example about how much security work to shift onto developers or how formal threat modelling should be, the chapter presents both positions.

Law, standards and accuracy

Security and privacy are full of numbered standards and legal provisions, and this chapter names them precisely: NIST special publications, IETF RFCs, OWASP projects, PCI DSS, the GDPR and the UK GDPR, and the EU Cyber Resilience Act. Standards change, so each mention gives the edition current at the last review shown on the section and points to the authoritative source. The material on regulation explains how the law affects architecture; it is information, not legal advice, and an organisation's own lawyers and data protection officer remain the people to ask about a specific situation.

The one idea to take away

Decide deliberately, early and in writing who and what your system trusts, for what, and why, and make every one of those trust decisions verifiable. Most of the techniques in this chapter, from threat models and least privilege to OAuth scopes, envelope encryption, mutual TLS, signed builds and audit logs, are ways of replacing implicit trust with explicit, checkable trust. A system designed that way limits the damage of the breach that will eventually come, and can show its customers, auditors and regulators how it protects them.

SECTION 8.1 OF 8.8 · INTERMEDIATE · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Security principles for architects

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 0.6 APIs and integration
- 2.7 Security, usability and more

Terms it uses from earlier: CIA triad, API gateway, TLS and HTTPS, Rate limit

THE IDEA IN BRIEF

Think about how a bank branch is protected. The vault is not guarded by one lock but by several: a locked front door, staff who only have keys to the rooms they need, a vault that needs two people to open, cameras, and an alarm that calls the police. The building was designed with these in mind from the start; nobody bolted a vault onto a shop afterwards. Software security works the same way. Architects decide early where the locked doors are, who gets which keys and what happens when one lock fails, so that a single mistake does not open everything.

WHAT YOU NEED TO KNOW

- Security is a quality of the structure: where trust boundaries lie, which component may do what and where data travels decide most security properties, and most breaches exploit such decisions rather than cryptography.
- Confidentiality, integrity and availability, plus authenticity, accountability and non-repudiation, must be named per asset because they conflict: failing closed protects data but costs availability.
- Saltzer and Schroeder's 1975 principles (economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, psychological acceptability) remain the foundation of secure design.
- Defence in depth layers independent controls so that one failure does not expose an asset; secure defaults and a paved road make the easy path the safe one.
- Reduce the attack surface and put authentication, authorisation and validation at every trust boundary; anything on the client is outside the boundary.

- The OWASP Top 10:2025 is an awareness list, not a standard; NIST's SSDF (SP 800-218) describes a secure development lifecycle, and architects own the cross-team security decisions while teams own their correct application.

Security as a quality of the structure

Chapter 2 introduced security as a quality attribute: the system's ability to protect data and functions from unauthorised use while serving those who are authorised, expressed through testable security scenarios that name an attacker, an attack, a target and a measurable response. This chapter takes the next step and asks how architecture produces that quality. The answer is that most security properties are decided by structure: which components exist, which of them trust which others, where data is stored and copied, and through which few points every request must pass.

Consider how real breaches happen. An attacker changes the order number in a URL and sees another customer's address, because the service checked that the user was logged in but not that the order was theirs. A storage bucket holding database exports is readable from the internet because a default was changed years ago. An internal reporting service accepts any request from inside the network, so one compromised laptop reaches every customer record. None of these needs clever cryptography to exploit; each is a design decision about trust that nobody made explicitly. Security by design means making those decisions deliberately, early and in writing, and then making them hard to violate by accident.

Security architecture therefore sits alongside the other cross-cutting concerns of arc42's section 8: an identity model, an authorisation model, a data protection concept and a logging concept that every team follows. The rest of this section sets out the vocabulary and principles on which the remaining sections of the chapter build.

Confidentiality, integrity and availability in a real system

The CIA triad is the starting point. On the food-delivery platform, **confidentiality** covers customers' names, addresses, phone numbers and order histories, couriers' live locations and identity documents, restaurants' payout bank details and the platform's own secrets. **Integrity** covers prices, basket totals, refunds, courier payouts and the menu a customer sees: an attacker who can change the price of a meal to zero, or redirect a restaurant's payouts to their own bank account, has not read anything secret but has done serious harm. **Availability** matters most during the evening peak, when a denial-of-service attack or a ransomware infection costs revenue by the minute.

ISO/IEC 25010 also lists **authenticity** (identities can be proved), **accountability** (actions can be traced to whoever performed them), **non-repudiation** (actions can be proved to have happened) and, new in the 2023 edition, **resistance** (the product keeps working as intended under attack). Architects should name which properties matter for each asset, because they pull in different directions. Failing closed protects confidentiality and integrity but costs availability: if the authorisation service is down, a fail-closed ordering service refuses orders. Failing open preserves availability and gambles with everything else. The right choice differs per operation: viewing a public menu can fail open to a cached copy, while issuing a refund must fail closed.

Saltzer and Schroeder's design principles

In 1975 Jerome Saltzer and Michael Schroeder, drawing on their experience with the Multics operating system at MIT, published eight design principles for protection mechanisms in *The Protection of Information in Computer Systems*. Half a century later they remain the most cited foundation of secure design, because they are about structure rather than technology. The table restates each one and gives a modern example from the food-delivery platform.

The eight design principles with modern examples

Principle	What it asks	Modern example
Economy of mechanism	Keep the security design as small and simple as possible, so that it can be inspected and tested thoroughly.	One shared, well-reviewed authorisation library or policy engine instead of permission checks hand-written differently in every service.
Fail-safe defaults	Base access on explicit permission; when in doubt, or when something fails, the answer is no.	A new API endpoint is denied until a policy grants access to it; an unreachable policy engine means deny, not allow.
Complete mediation	Check every access to every object every time, not just the first time.	The ordering service checks ownership on every `GET /orders/{id}` rather than trusting any ID from a logged-in user; cached decisions expire quickly.
Open design	Security must not depend on the design being secret; only keys and passwords should be secret.	Published algorithms (AES, TLS 1.3) and standard protocols (OAuth 2.0) that stay secure even if attackers read the source code.
Separation of privilege	Where possible, require two independent conditions (two keys) before granting access.	Refunds above a threshold need a second approver; a production release needs both a code review and a passing pipeline; multi-factor login.
Least privilege	Give every program and user only the privileges needed for the job, for as long as needed.	The menu service's database account cannot read the payments schema; a build job's cloud role can push images but not delete clusters.
Least common mechanism	Minimise mechanisms shared between users, because each shared mechanism is a path for information to leak.	Separate caches, queues or connection pools per tenant or trust level; a shared cache keyed carelessly serves one user's data to another.
Psychological acceptability	Protection must be easy to use correctly, or people will work around it.	Passkeys instead of complex password rules; secure defaults in the platform; warnings that are rare and therefore read.

WHERE THIS CAME FROM

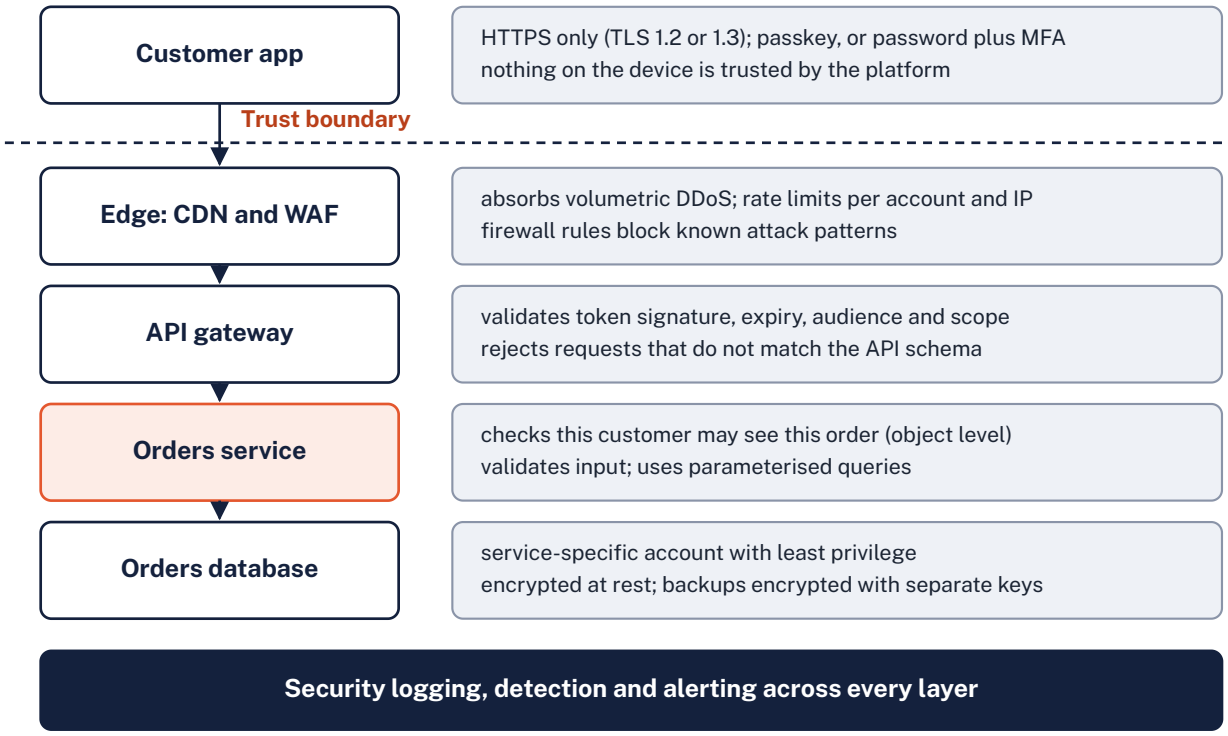
Where the principles came from

Saltzer and Schroeder's paper appeared in the *Proceedings of the IEEE* in September 1975. Besides the eight principles, it described two more, **work factor** (compare the cost of circumventing a mechanism with the attacker's resources) and **compromise recording** (record that a compromise happened even if it cannot be prevented), which the authors noted apply only imperfectly to computer systems. Both survive today: work factor in password hashing and key lengths, compromise recording in tamper-evident audit logs. Open design restates Auguste Kerckhoffs's 1883 principle for military ciphers: a system should remain secure even if everything about it except the key is public.

Defence in depth

Defence in depth means layering several independent controls, so that the failure or bypass of one does not expose the asset. The idea comes from military fortification and is standard practice in safety engineering as well as security. Its value depends entirely on the word *independent*: two layers that fail for the same reason, such as two services that both trust a forged header, are one layer drawn twice.

The diagram shows the layers on the food-delivery platform's order path. If the gateway's token validation has a bug, the ordering service's object-level check still prevents one customer reading another's order. If an attacker steals the database's disk snapshots, encryption at rest with keys held elsewhere protects them. If all of that fails, monitoring should notice the unusual pattern of requests and someone should respond. Defence in depth is also the honest answer to the question "what if this control fails?", which every threat model asks.



Each layer applies its own controls; a single bypassed control does not expose the data.

IF YOU ARE NEW TO THIS

Step by step: when two locks are really one

The idea that needs the most care here is that defence in depth works only if the layers are independent, so that each can stop an attack that gets past the others. Follow one request along the order path in the diagram.

- A logged-in customer asks to see their order 1042. The app sends the request with the customer's access token, which shows who they are.
- The API gateway checks that the token is genuine, unexpired and meant for this API, and passes the request on to the ordering service.
- Now the customer edits the request, swapping 1042 for 1043, someone else's order. The token is still valid, so the gateway lets it through: it cannot know who owns which order. The ordering service checks, finds that order 1043 is not this customer's, and refuses.
- That second check is a separate layer. Even if a bug let a bad token past the gateway, the ordering service would still refuse to show one customer another's order.
- But suppose the ordering service also checked the token, using the same shared library as the gateway, and a flaw in that library accepted tokens with no valid signature. One forged token would then fool both checks at once. Two checks that fail for the same reason are one layer drawn twice.
- The lower layers follow the same rule. Encryption at rest protects stolen disk snapshots only because its keys are held elsewhere, and monitoring is there to notice an attack if everything else fails.
- **The lesson:** for every control, ask 'what if this one fails?', and make sure that whatever defeats it would not also defeat the layers behind it.

Secure by default and secure by design

A product is secure by default when its out-of-the-box configuration is the safe one, so that weakening protection requires a deliberate act. Many large breaches came from insecure defaults left in place: databases listening on the internet without a password, administrator consoles with factory credentials, storage buckets that could be made public with one setting. Vendors have moved: since April 2023, for example, new Amazon S3 buckets are created with Block Public Access switched on and access control lists disabled. In 2023 the US Cybersecurity and Infrastructure Security Agency (CISA) and partner agencies from several countries published joint guidance urging manufacturers to make products secure by design and by default, and in 2024 CISA launched a voluntary Secure by Design pledge for software makers.

For architects, secure by default applies to the platform as much as to products. If the shared service template already includes token validation, structured security logging, a non-root container user and a least-privilege database account, every new service starts secure and teams have to opt out explicitly, which is visible in review. This is what some organisations call a **paved road**: the easiest path is also the secure one.

Closely related is how a system behaves when something goes wrong. The OWASP Top 10:2025 added a category for mishandling exceptional conditions, which includes failing open: an error in a permission check that is caught and ignored, a timeout that defaults to "allowed", a parsing failure that skips validation. Fail-safe defaults, applied in code, mean that every exception path in a security decision ends in denial.

Attack surface and trust boundaries

The attack surface is everything an attacker can reach and influence: public endpoints and their parameters, open ports, file uploads, message queues that accept input from partners, webhooks, administrator consoles, mobile apps that can be decompiled, and the third-party code and people the system trusts. Michael Howard at Microsoft popularised the idea of measuring it in the early 2000s; the practical rule is to make it as small as possible and to know exactly what is on it. Removing an unused endpoint, closing an administrator port to the internet, or turning off a feature nobody uses reduces risk more cheaply than any added control. Every architectural style choice in Chapter 6 has a security cost here: each new network-reachable service adds endpoints, credentials and traffic to protect.

A trust boundary is where data or control passes between parts with different levels of trust or privilege. The most obvious is between the internet and the platform, but there are many more: between the customer app and the API, between a restaurant's tablet and the menu service, between services owned by different teams, between the platform and a payment provider, between application code and the database, and between ordinary users and administrators. Trust boundaries matter because they are where checks belong: authentication, authorisation, input validation and output encoding happen when data crosses a boundary, never on the assumption that the other side has already done it. Drawing the trust boundaries on a data-flow diagram is the first step of threat modelling.

WATCH OUT

The client is outside the boundary

Anything that runs on the customer's device, whether a mobile app, a single-page web application or a restaurant's tablet, is under the attacker's control. Prices, discounts, permissions and order totals calculated on the client are suggestions; the server must recompute or verify them. Hiding a button in the user interface is not authorisation. API keys embedded in a mobile app are not secret, because the app can be decompiled.

The OWASP Top 10 as an awareness list

The OWASP Top 10 is the Open Worldwide Application Security Project's ranked list of the most critical web application security risks, compiled from contributed vulnerability data and a community survey. The current edition, the **OWASP Top 10:2025**, is the eighth. Compared with the 2021 edition, broken access control stays first and now includes server-side request forgery, security misconfiguration rises to second, the old vulnerable and outdated components category widens into **software supply chain failures**, and a new category covers **mishandling of exceptional conditions**.

OWASP Top 10:2025 and the architectural levers

Category	What goes wrong	Architectural lever
A01 Broken Access Control	Users act outside their permissions, such as reading another customer's order by changing an ID; includes server-side request forgery.	One authorisation model, deny by default, object-level checks in every service, restricted outbound network access.
A02 Security Misconfiguration	Insecure defaults, open storage, verbose error pages, unnecessary features left on.	Infrastructure as code, hardened templates, automated policy checks in the pipeline.
A03 Software Supply Chain Failures	Vulnerable or compromised dependencies, build tools and distribution channels.	Dependency inventory (SBOM), pinning, signed provenance; see supply-chain security.
A04 Cryptographic Failures	Sensitive data unencrypted, weak algorithms, poor key handling.	Data classification, TLS everywhere, a KMS and envelope encryption.
A05 Injection	Untrusted data interpreted as code: SQL, operating-system commands, cross-site scripting.	Parameterised queries, context-aware output encoding, safe framework APIs.
A06 Insecure Design	Missing or ineffective controls that no amount of correct coding can fix.	Threat modelling, secure design patterns, abuse cases alongside user stories.
A07 Authentication Failures	Credential stuffing, weak recovery flows, session handling flaws.	A central identity provider, MFA or passkeys, rate limits on login.
A08 Software or Data Integrity Failures	Trusting code, updates or data without verifying integrity, such as unsigned updates or unsafe deserialisation.	Signature verification, integrity checks on data crossing boundaries.
A09 Security Logging and Alerting Failures	Attacks are not recorded, or recorded but never noticed or acted on.	A security logging standard, alerting, incident runbooks.
A10 Mishandling of Exceptional Conditions	Errors handled inconsistently, systems failing open, error messages leaking details.	Fail closed, central error handling, resource limits.

Treat the list as what OWASP says it is: an awareness document, not a standard. It is useful for training, for checking that a threat model has not missed a whole class of problem and for talking to stakeholders. It is not a complete set of requirements, and "we are OWASP Top 10 compliant" means little. For verifiable requirements, OWASP's Application Security Verification Standard (ASVS) lists testable controls at three levels of rigour, and the separate OWASP API Security Top 10 covers risks specific to APIs, such as broken object-level authorisation.

A secure development lifecycle and shifting left

A secure development lifecycle builds security activities into every stage of delivery rather than relying on a test just before release. Microsoft made its Security Development Lifecycle mandatory in 2004, after Bill Gates's 2002 Trustworthy Computing memo, and the idea spread across the industry. The current reference is NIST's Secure Software Development Framework (SSDF), SP 800-218, version 1.1 of which was published in February 2022; NIST released a draft revision (version 1.2) for comment in December 2025, which had not been finalised at the time of writing. The SSDF groups practices into four families: **prepare the organisation** (roles, training, toolchains, security requirements), **protect the software** (protect code and build integrity, keep provenance), **produce well-secured software** (design to meet security requirements, threat-model, review and test, configure securely by default) and **respond to vulnerabilities** (identify, fix, and analyse root causes so the same class does not recur). It describes outcomes rather than prescribing tools, which makes it a good checklist for an architect assessing a delivery process.

To shift left is to move security work earlier: security requirements and abuse cases with the user stories, threat modelling in design, static analysis and secret scanning on every commit, dependency checks in the pipeline. The economic argument is familiar from Chapter 1: a missing authorisation check found in a design review costs an hour; the same flaw found after a breach costs a notification to regulators. Practitioners disagree about how far to push it. Advocates say developers who understand security write fewer vulnerabilities. Critics point out that simply moving security tools in front of developers floods them with low-quality findings and turns security into everyone's part-time job and nobody's expertise; they argue for secure defaults built into the platform (the paved road), so that the secure path needs no extra effort, and for security specialists embedded where the risk is highest. Most mature organisations do both, and also keep testing and monitoring in production, which is why some now speak of shifting security everywhere.

Testing the running system: DAST and fuzzing

Static analysis reads code; these techniques exercise it. **Dynamic application security testing** (DAST) probes a running application from the outside with hostile requests, without needing its source, looking for flaws such as SQL injection, cross-site scripting, weak authentication and server misconfiguration. It suits a nightly run against a test environment, but it finds only what it can reach and is generally weak on business logic, such as one customer viewing another's order. **Fuzzing** feeds a program malformed or unexpected input until it fails. It began with Barton Miller's group at the University of Wisconsin-Madison, whose `fuzz` tool crashed or hung about a quarter to a third of the UNIX utilities they tested. It is especially useful for parsers, such as the platform's menu-import code. Google's OSS-Fuzz runs it at scale for open-source projects.

What architects own and what developers own

Security work is shared, and confusion about who owns what is itself a vulnerability. The architect owns decisions that cross team boundaries and are expensive to change; teams own their correct application. A common pattern is a small central security team that sets standards and runs the platform's security services, security champions in each product team who lead threat modelling and reviews, and architects who make sure the structural decisions are made and recorded.

Dividing security responsibilities

Concern	Architect (and security architecture)	Development teams
Trust boundaries and data flows	Define them, document them, lead threat models for significant changes.	Keep the code consistent with them; raise changes that move a boundary.
Identity and access	Choose the identity provider, token model, authorisation model and where policy is decided.	Enforce authorisation at every endpoint and object; test it.
Data protection	Classify data; decide what is encrypted where, with which keys; choose the KMS and secrets vault.	Use the approved libraries; never log secrets or personal data unnecessarily.
Input and output handling	Set validation and encoding strategy at boundaries; provide shared libraries.	Validate, encode and parameterise in every handler.
Dependencies and build	Set pipeline controls, approved sources and SBOM policy.	Keep dependencies current; act on alerts within agreed times.
Detection and response	Define security logging, audit events and incident roles.	Emit the agreed events; fix vulnerabilities within the agreed service levels.

WORKED EXAMPLE

Worked example: applying the principles to a support refund feature

Support agents on the food-delivery platform need to refund customers for failed deliveries from the back-office tool. A first design lets any agent call the Payments service's refund endpoint with any order number and amount. The table shows how the principles reshape it.

Least privilege	Agents get a `refund:create` permission only for orders in their own market; the back-office tool's service identity may call the refund API but never read card data.
Fail-safe defaults	If the policy check or the order lookup fails or times out, the refund is refused with a "try again" message; there is no fallback path that skips checks.
Complete mediation	Payments itself re-checks every refund against the original charge (the refund cannot exceed the captured amount minus earlier refunds), rather than trusting the back-office tool.
Separation of privilege	Refunds above £100, or a third refund to the same customer in a week, need a supervisor's approval in the same tool.
Economy of mechanism	Uses the platform's existing policy engine and audit library; no special-case permission code is written for refunds.
Psychological acceptability	Approval is requested with one click; reasons are chosen from a short list, so the control does not slow honest agents down.
Accountability	Every refund is written to an append-only audit log with agent, approver, order, amount and reason.
Defence in depth	The back-office tool is reachable only through single sign-on with phishing-resistant MFA from managed devices.

Both a dishonest agent and an attacker who has taken over an agent's account are now limited to small, logged refunds in one market.

The cost is a short delay for large refunds and some extra design work, which is far cheaper than retrofitting these controls after fraud is discovered.

CHECK YOUR UNDERSTANDING

▼ Why does defence in depth depend on the layers being independent?

If two layers fail for the same reason, such as both trusting the same forged header, one attack defeats both, so they give no more protection than a single layer.

▼ Which Saltzer and Schroeder principle is violated when a service checks that a user is logged in but not that the requested order belongs to them?

Complete mediation: every access to every object must be checked for authority, not just the fact that the user has authenticated.

▼ A team says its application is "OWASP Top 10 compliant". What is wrong with that claim?

The Top 10 is an awareness list of risk categories, not a standard with testable requirements; verification needs something like OWASP ASVS and a threat model of the actual system.

▼ Give one example of failing open and explain why it is dangerous.

A permission check that treats a timeout from the policy engine as "allowed". An attacker who can slow or break the policy engine then gets access to everything it protects.

▼ Try it: Any box-office clerk can apply a 100% discount to any booking, and the app merely hides the button from junior staff. Which security principles does this break? Redesign it briefly, including who can approve large discounts.

Hiding a button is client-side, so it is outside the trust boundary, and complete mediation fails: the server must check every discount. Least privilege fails too. Give clerks a limited permission, say discounts up to 10% in their own venue, denied by default. Require a second person for larger discounts, which is separation of privilege, and record who did what. Fail closed if the permission check is unavailable, accepting a slower queue.

SOURCES

- OWASP Top Ten project (OWASP Top 10:2025): https://owasp.org/www-project-top-ten/
- NIST SP 800-218: Secure Software Development Framework (SSDF) version 1.1: https://csrc.nist.gov/pubs/sp/800/218/final
- Saltzer and Schroeder, The Protection of Information in Computer Systems (1975): https://doi.org/10.1109/PROC.1975.9939
- OWASP SAMM: Software Assurance Maturity Model: https://owasp.org/www-project-samm/

- OWASP DevSecOps Guideline: Dynamic Application Security Testing: https://owasp.github.io/www-project-devsecops-guideline/latest/02b-Dynamic-Application-Security-Testing
- OWASP: Fuzzing: https://community.owasp.org/Fuzzing
- Miller, Fredriksen and So: An Empirical Study of the Reliability of UNIX Utilities: https://ftp1.cs.wisc.edu/par-distr-sys/papers/fuzz.pdf
- OSS-Fuzz documentation: https://google.github.io/oss-fuzz/

SECTION 8.2 OF 8.8 · INTERMEDIATE · 23 MIN READ · LAST REVIEWED NOVEMBER 2026

Threat modelling: STRIDE, attack trees and data-flow diagrams

BEFORE YOU READ THIS

This section builds on:

- 2.7 Security, usability and more
- 4.3 The C4 model
- 8.1 Security principles

Terms it uses from earlier: Trust boundary, Container diagram, Webhook, API gateway, Architecture Decision Record (ADR)

THE IDEA IN BRIEF

Before a family goes on holiday, someone usually walks round the house asking what could go wrong: is the back door locked, does a neighbour have a key, is the spare key under the flowerpot, will the post pile up and show that nobody is home? Then they decide what to do about each: lock the door, ask the neighbour to collect the post, move the key. Threat modelling is the same walk round a software design. The team draws how data moves through the system, asks what an attacker could do at each step, and decides what to fix, what to accept and what to remove.

WHAT YOU NEED TO KNOW

- Threat modelling answers Shostack's four questions: what are we working on, what can go wrong, what are we going to do about it, and did we do a good enough job.
- A data-flow diagram of external entities, processes, data stores, data flows and trust boundaries is the usual model; flows that cross trust boundaries deserve the most attention.
- STRIDE (Kohnfelder and Garg, Microsoft, 1999) names six threat categories, each violating one property: spoofing (authenticity), tampering (integrity), repudiation (non-repudiation), information disclosure (confidentiality), denial of service (availability) and elevation of privilege (authorisation).
- Attack trees (Schneier, 1999) model an attacker's goal as AND and OR nodes; the cheapest path through the tree shows where a mitigation matters most.

- Rate threats by likelihood and impact with written reasons; DREAD fell out of favour because its scores were subjective, and CVSS rates known vulnerabilities, not design threats.
- Every threat ends as mitigated, eliminated, transferred or explicitly accepted by someone with authority; threat-model at design time and again whenever trust boundaries, data or authentication change.

What threat modelling is

Threat modelling is the activity of examining a design to find what could go wrong with its security, and deciding what to do about it, before attackers or incidents find it for you. It is the main way security principles become concrete decisions, and the most direct answer to the OWASP category of insecure design: flaws that no amount of correct coding can fix because the needed control was never designed in.

Adam Shostack, who led threat modelling work at Microsoft and wrote the standard book on it, reduces the activity to four questions. **What are we working on?** (a model of the system, usually a diagram). **What can go wrong?** (threats, found systematically). **What are we going to do about it?** (mitigate, eliminate, transfer or accept each threat). **Did we do a good enough job?** (review the model, the threats and the follow-up). Any method that answers the four questions is threat modelling; the rest of this section describes the techniques most teams use to answer them.

In 2020 a group of practitioners, including Shostack, published the Threat Modeling Manifesto, modelled on the Agile Manifesto. It values a culture of finding and fixing design issues over checkbox compliance, people and collaboration over processes and tools, a journey of understanding over a one-off security snapshot, doing threat modelling over talking about it, and continuous refinement over a single delivery. Its point is that threat modelling is a habit of the team that builds the system, not a document produced by outsiders once a year.

Modelling the system: data-flow diagrams and trust boundaries

The first question needs a model of the system that shows where data goes and where trust changes. The traditional choice is a data-flow diagram (DFD) with five kinds of element: **external entities** (people or systems outside your control, drawn as rectangles), **processes** (code that you run, drawn as circles or ellipses), **data stores** (databases, files, queues and caches, drawn between two parallel lines), **data flows** (arrows showing data moving between the others) and **trust boundaries** (dashed lines or boxes where the level of trust changes).

DFDs suit threat modelling because attacks follow data: every flow that crosses a trust boundary is a place where an attacker can inject, intercept or replay something. A C4 container diagram from Chapter 4 is a good starting point; add the trust boundaries, make every arrow a data flow with a direction and a label, and include the stores and queues that container diagrams sometimes leave implicit. Keep the diagram at the level where threats are found: one diagram per significant feature or flow is more useful than one huge diagram of everything.

Finding threats with STRIDE

STRIDE was created at Microsoft in 1999 by Loren Kohnfelder and Praerit Garg as a mnemonic to help developers think about threats to their products. Each letter names a category of threat, and each category is the violation of one security property, which is what makes it useful: it is a systematic prompt, not a list of known attacks.

STRIDE and the property each threat violates

Threat	Property violated	Food-delivery example	Typical mitigations
Spoofing	Authenticity	An attacker logs in as a customer with a stuffed password; a rogue process calls Payments pretending to be Orders.	Strong authentication (MFA, passkeys), rate limits, mutual TLS and workload identity between services.
Tampering	Integrity	A client sends a changed basket total; someone alters a payment webhook or a database row.	Server-side recomputation, signatures and MACs, TLS, least-privilege write access.
Repudiation	Non-repudiation (accountability)	A customer denies placing an order; a support agent denies issuing a refund, and nothing proves otherwise.	Audit logs tied to authenticated identities, timestamps, tamper-evident storage.
Information disclosure	Confidentiality	An API returns customers' addresses to couriers who are not delivering to them; error messages reveal internals.	Authorisation on every object, data minimisation, encryption, generic error messages.
Denial of service	Availability	Bots flood checkout during the evening peak; an expensive search query exhausts the database.	Rate limits and quotas, timeouts, bulkheads, autoscaling, query cost limits.
Elevation of privilege	Authorisation	A customer calls a restaurant-manager endpoint; injection lets an attacker run code as the service.	Deny-by-default authorisation, input validation, sandboxing, least privilege.

A practical way to apply STRIDE is **STRIDE per element**: walk through every element of the DFD and ask only the questions that apply to its type. External entities can be spoofed and can repudiate; processes are exposed to all six categories; data stores to tampering, information disclosure and denial of service (and to repudiation when the store is a log); data flows to tampering, information disclosure and denial of service. Spending most attention on flows that cross trust boundaries keeps the exercise focused.

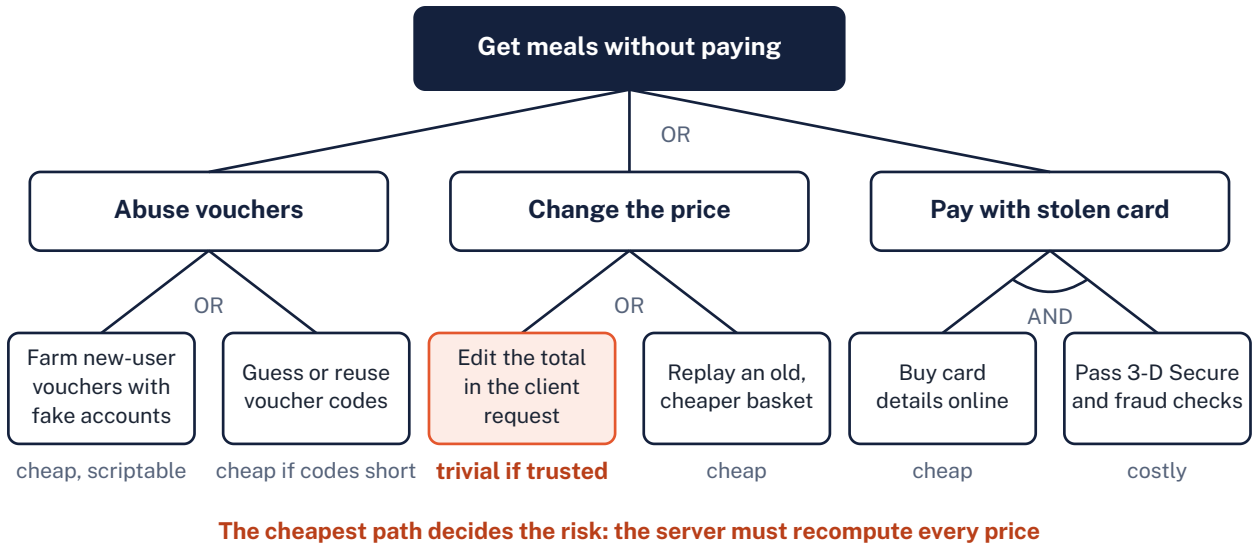
STRIDE finds threats; it does not rank them or choose mitigations. Its categories also overlap in practice (a spoofed identity usually leads to information disclosure or elevation of privilege), which does not matter: the goal is to find the threat, not to classify it perfectly. For privacy

threats, such as linking or identifying people from data that looks harmless, the LINDDUN method from KU Leuven plays the role that STRIDE plays for security; it is discussed with privacy by design.

Attack trees

An attack tree, popularised by Bruce Schneier in a 1999 article in *Dr. Dobb's Journal*, looks at the system from the attacker's side. The root is a goal, such as getting meals without paying; each child is a way of reaching its parent. Children of an OR node are alternatives; all children of an AND node are needed together. Leaves can be annotated with properties such as cost, skill needed, or whether the step is possible at all, and those values propagate up the tree: the cost of an OR node is that of its cheapest child, the cost of an AND node the sum of its children.

Attack trees complement STRIDE well. STRIDE walks the design element by element and is good at coverage; an attack tree follows a motivated attacker and is good at showing which path is cheapest and therefore where a mitigation buys most. In the tree below, paying with a stolen card is relatively costly because it needs two things at once, while changing the price in the client request is trivial if the server trusts it. The cheapest path sets the risk, so recomputing every price on the server is worth more than any amount of card-fraud tuning.



An attack tree for one attacker goal; the cheapest leaf shows where a mitigation matters most.

IF YOU ARE NEW TO THIS

Step by step: reading an attack tree

The step most people trip over here is how an attack tree shows where a fix matters most. An attacker takes the cheapest route, so a goal is only as hard to reach as its cheapest path. Work up the tree in the diagram from the bottom.

- Start with the leaves, each a single step with a rough cost. Farming vouchers with fake accounts is cheap; getting a stolen card past 3-D Secure (the bank's extra check when paying) and fraud checks is costly; editing the total in the client request is trivial if the server trusts it.
- 'Pay with stolen card' is an AND node: the attacker needs card details and a way past the checks together. Its cost is the sum of the two, so the one costly step makes the whole branch costly.
- 'Change the price' is an OR node: either of its leaves will do, so its cost is that of its cheapest child. The branch stays trivial for as long as the server trusts prices sent by the client.
- The goal at the top is also an OR node, costing what its cheapest branch costs: changing the price. That is the platform's real risk, however good its fraud checks are.
- So the team makes the server recompute every price. An edited total now achieves nothing, and that leaf stops being an option.
- Work upwards again. The cheapest remaining leaves, such as farming vouchers with fake accounts, now set the risk, and the next fix is worth most there.
- **The lesson:** making an expensive path harder changes nothing while a cheap one remains. Find the cheapest path, close it, then look again, because a motivated attacker simply moves to the next cheapest.

Rating risk

A threat model typically finds more threats than a team can address at once, so each needs a rating. The usual approach is to judge **likelihood** (how easy, how attractive, how exposed) and **impact** (what is lost if it happens: money, personal data, availability, reputation, legal exposure) on simple scales such as low, medium and high, and to combine them in a matrix. The value of the rating lies less in the precision of the number than in a written reason that others can challenge. Organisations with mature risk management sometimes use quantitative methods such as FAIR (Factor Analysis of Information Risk) to express risk as a range of expected loss.

Two popular shortcuts are worth knowing because they are often misused. **DREAD** (damage, reproducibility, exploitability, affected users, discoverability), once used at Microsoft, scored each threat on five factors (often from 1 to 10) and averaged them. It fell out of favour, including at Microsoft, because different people scored the same threat very differently and the averaged number gave a false sense of precision; discoverability in particular rewards obscurity, which contradicts open design. CVSS, the Common Vulnerability Scoring System maintained by FIRST (version 4.0, 2023), is the right tool for a different job: rating the severity of a known vulnerability in a product, such as a flaw in a library. Its base score describes the vulnerability, not the risk of a design threat in your context, so it should not be used to rank the output of a threat model.

Every threat ends in one of four decisions: **mitigate** (add a control), **eliminate** (remove the feature or the data, often the best option), **transfer** (let someone better placed carry it, such as a payment provider holding card data) or **accept** (record that a person with the authority to do so has accepted the residual risk, and why). Unrecorded acceptance is the most common failure: the threat was found, nobody fixed it, and nobody decided not to.

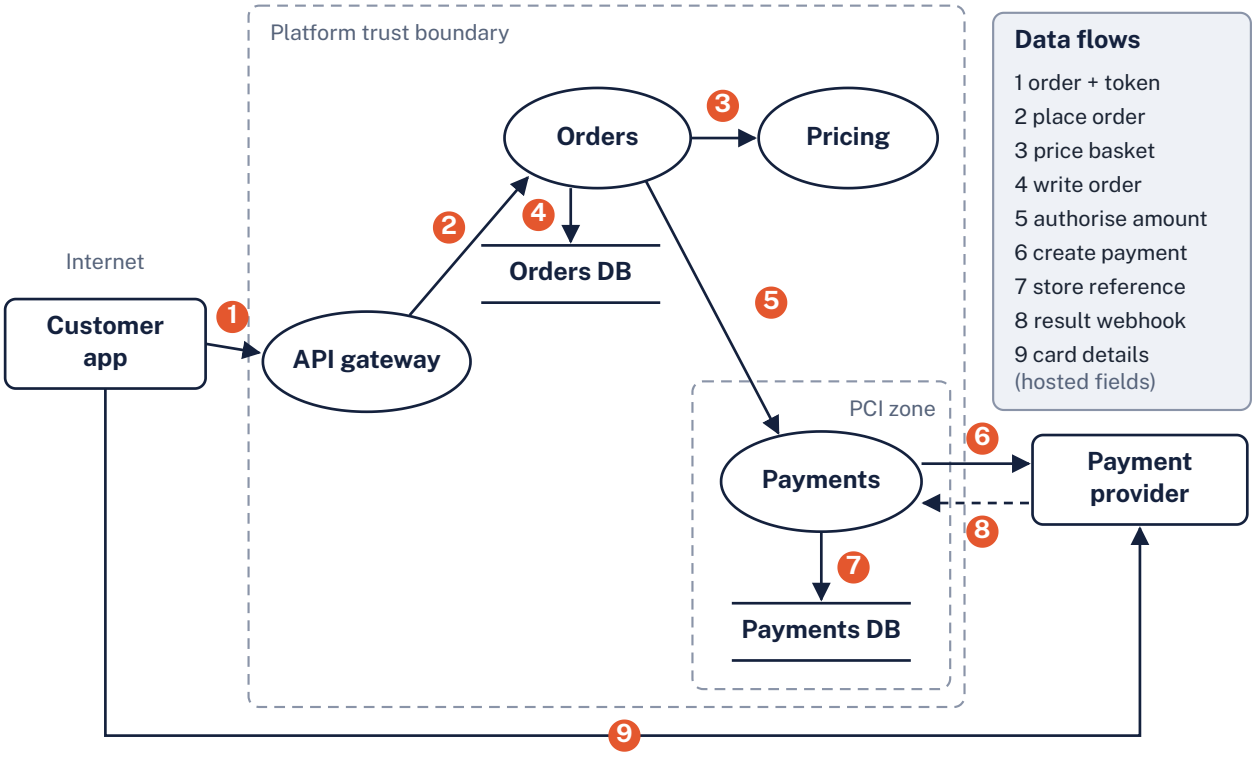
WORKED EXAMPLE

Worked example: threat-modelling the checkout

The food-delivery team is redesigning checkout so that customers can pay in the app with a card or a wallet. The architect runs a 90-minute session with the ordering and payments developers, a product owner and a security champion. The DFD below answers the first question, what are we working on. It shows two trust boundaries: the platform as a whole, and a narrower, tightly restricted payments zone (labelled PCI zone) around the payments service and its database, which hold payment tokens and references and talk to the payment provider; isolating them keeps the platform's PCI DSS scope small. Card details themselves (flow 9) go straight from hosted payment fields in the app to the payment provider and never enter the platform.

1. What are we working on?	The checkout DFD below: customer app, API gateway, Orders, Pricing, Payments, two databases and the payment provider, with two trust boundaries (the platform and the PCI zone) and nine numbered flows.
2. What can go wrong?	STRIDE per element on every flow that crosses a boundary (1, 5, 6, 8 and 9) and on the order API, plus the attack tree for getting meals without paying shown earlier.
3. What are we going to do about it?	Each threat is rated, given a decision (mitigate, eliminate, transfer or accept) and an owner; the most important are in the table below.
4. Did we do a good enough job?	A review a month later compares the model with what was built and checks that every high risk was mitigated or accepted in writing.

The cheapest way to protect card numbers is not to hold them: flow 9 keeps them out of the platform entirely, which removes threats rather than mitigating them.



The checkout data-flow diagram with its two trust boundaries and numbered data flows.

Selected threats from the checkout threat model

ID and where	STRIDE	Threat	Risk	Decision and mitigation
T1, flow 1	S	Credential stuffing takes over accounts that have saved cards.	High	Mitigate: passkeys, or MFA on new devices; rate limits per account and IP; breached-password check.
T2, flows 1 and 2	T	The client sends an altered price, discount or delivery fee.	High	Mitigate: Orders ignores client prices; Pricing recomputes from menus and vouchers; the payment is created for the server's total.
T3, flow 5	S, E	Another internal service, or an attacker inside the network, asks Payments to authorise or refund arbitrary amounts.	High	Mitigate: mutual TLS with workload identity; Payments accepts authorisations only from Orders' identity, bound to an order ID and amount.
T4, flow 8	S, T	A forged "payment succeeded" webhook marks an unpaid order as paid.	High	Mitigate: verify the provider's signature and timestamp; confirm status with the provider's API before releasing the order; process each event once.
T5, `GET /orders/{id}`	I	Enumerating order IDs reveals other customers' addresses (broken object-level authorisation).	High	Mitigate: ownership check on every request; random identifiers as a secondary measure only.
T6, Orders DB	I	Order records with addresses leak from a backup or through an over-broad analytics account.	Medium	Mitigate: encryption at rest with KMS keys, least-privilege accounts, backups in a separate account.
T7, flow 1	D	Bots exhaust checkout capacity at the evening peak.	Medium	Mitigate: rate limits and bot detection at the edge, reserved capacity for checkout, a waiting room if needed.
T8, Orders	R	A customer disputes placing an order and there is no evidence.	Low	Mitigate: audit event with user ID, device, time and order details, retained according to policy.

ID and where	STRIDE	Threat	Risk	Decision and mitigation
T9, flow 9	I	Card numbers passing through the platform would bring every component into PCI DSS scope.	Design	Eliminate and transfer: hosted fields send card data directly to the provider; the platform stores only tokens and references.

The answers to the last two questions are as important as the table. Each mitigation becomes a backlog item with an owner; T3 and T4 also become security scenarios with automated tests (a webhook with a bad signature must be rejected; a call to Payments from any identity other than Orders must fail). The two architectural decisions, hosted payment fields and mutual TLS between Orders and Payments, are recorded as ADRs. For the fourth question, the team reviews the model a month later against what was actually built, and the security champion checks that every high-risk item was either mitigated or explicitly accepted.

When and how often to threat-model

Threat modelling pays off most at design time, when changing a decision costs a whiteboard eraser. It should then be repeated whenever something significant changes: a new trust boundary or external integration, a new kind of sensitive data, a change to authentication or authorisation, a new deployment environment, or a move from a monolith to services. A sensible rule is to make "does this change the threat model?" a question in design reviews and in the template for architecture decision records, and to schedule a review of each system's model at least yearly and after every security incident.

The effort should be proportionate, in the spirit of the risk-driven approach in Chapter 1. A new microservice that processes payments deserves a half-day session with a full DFD; adding a field to a public menu needs five minutes and a STRIDE glance at the changed flow. Many teams keep their DFD as code next to the system (OWASP Threat Dragon and Microsoft's Threat Modeling Tool are free diagramming tools, and libraries such as OWASP pytm generate diagrams and candidate threats from code), so that the model is reviewed and updated with the system it describes.

WATCH OUT

Common threat modelling mistakes

- Trying to model the whole enterprise at once; start with one feature, flow or service.
- Doing it once for an audit and filing the document; the model must change with the system.
- Leaving it to the security team alone; the people who build the system know where its weak points are.
- Speculating about who the attacker is instead of what can go wrong with the design.
- Listing threats without decisions, owners or follow-up, or accepting risks without recording who accepted them.
- Accepting hundreds of generic threats from a tool without judging which ones apply.

GOING DEEPER

Other methods

PASTA (Process for Attack Simulation and Threat Analysis, described by Tony UcedaVélez and Marco Morana in 2015) is a seven-stage, risk-centric method that starts from business objectives and ends in attack simulation; it suits organisations that want threat modelling tied closely to business risk. MITRE ATT&CK is a knowledge base of real attacker tactics and techniques, more useful for detection engineering in security operations than for design. Systems that include machine-learning models add threats such as prompt injection and training-data poisoning, covered in Chapter 12.

CHECK YOUR UNDERSTANDING

▼ **Which STRIDE categories apply to a data flow in STRIDE per element?**

Tampering, information disclosure and denial of service: data in transit can be altered, read or blocked.

▼ **Why is CVSS the wrong tool for ranking threats found in a threat model?**

CVSS scores the severity of a specific known vulnerability in a product; it does not account for your design, assets and context, so it does not measure the risk of a design-level threat.

▼ **In an attack tree, how is the cost of an AND node computed from its children, and why does that matter?**

It is the sum of its children's costs, because all steps are needed. AND nodes are therefore usually costly paths, while the cheapest child of an OR node sets the risk.

▼ **What is the most common failure in the fourth step of threat modelling?**

Threats that were found but neither fixed nor explicitly accepted: nobody owns them and nobody decided to live with them.

▼ **Try it: Checkout has a webhook where the payment provider reports success. Using STRIDE, name the threat if anyone can post to it, one more threat, and a mitigation for each. How does your work end for each threat?**

Spoofing: a forged success message marks an unpaid order as paid. Mitigate by verifying the provider's signature and confirming the payment with the provider. Denial of service: flooding the endpoint blocks real confirmations; mitigate with rate limits and a queue. Tampering with the amount is covered by the signature. The endpoint crosses a trust boundary, so it deserves attention. Each threat must end as mitigated, eliminated, transferred or explicitly accepted by someone with authority.

SOURCES

- Threat Modeling Manifesto (2020): https://www.threatmodelingmanifesto.org/
- OWASP Threat Modeling Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Threat_Modeling_Cheat_Sheet.html
- Microsoft Threat Modeling Tool: STRIDE threat categories: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats
- Bruce Schneier, Attack Trees (Dr. Dobb's Journal, December 1999): https://www.schneier.com/academic/archives/1999/12/attack_trees.html
- FIRST: Common Vulnerability Scoring System: https://www.first.org/cvss/

SECTION 8.3 OF 8.8 · ADVANCED · 29 MIN READ · LAST REVIEWED NOVEMBER 2026

Identity and access: authentication, authorisation, OAuth 2.0 and OpenID Connect

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 0.6 APIs and integration
- 8.1 Security principles

Terms it uses from earlier: Access token, API gateway, TLS and HTTPS, HTTP

THE IDEA IN BRIEF

Think of a large office building. At reception you show a photo ID and get a visitor badge: that is authentication, proving who you are. The badge then opens some doors and not others: that is authorisation, deciding what you may do. If you send a courier to collect a parcel for you, you do not give them your own badge; reception gives them a limited pass that opens only the post room, only today. OAuth works like that limited pass for apps. And the building's own machines, such as the lifts and the alarm system, need their own badges too, so they can prove to each other that they are genuine.

WHAT YOU NEED TO KNOW

- Authentication establishes who is calling and is best centralised in one identity provider; authorisation decides what they may do to which object and must be enforced where the data is, in every service.
- NIST SP 800-63B (revision 4, 2025) favours length over composition rules, requires blocklist checks against common and breached passwords, forbids forced periodic changes and expects phishing-resistant options such as passkeys.
- Server-side sessions are easy to revoke; self-contained JWTs scale but cannot be revoked before expiry, so keep them short-lived, pin the algorithm, check issuer, audience and expiry, and keep them out of localStorage.
- OAuth 2.0 (RFC 6749) delegates limited access through tokens; users' apps should use the authorisation code flow with PKCE (RFC 7636), services client credentials; OAuth 2.1 is still an Internet-Draft in 2026.
- OpenID Connect adds an ID token telling the client who logged in; the ID token is for the client and the access token for the API.

- RBAC, ABAC and relationship-based access control (Google Zanzibar, 2019) suit different needs; policy engines such as OPA separate decision from enforcement, and workload identity (SPIFFE, mTLS) replaces shared service secrets.

Authentication and authorisation

Every request to the food-delivery platform raises two separate questions. Authentication asks *who is this?* and is answered by evidence: a password, a passkey, a certificate, a signed token. Authorisation asks *may they do this, to this, now?* and is answered by policy: this customer may read this order because they placed it; this courier may see this pickup address because the delivery is assigned to them. HTTP keeps them apart too: `401 Unauthorized` means the request lacks valid authentication, while `403 Forbidden` means the caller is known but not allowed.

The distinction matters architecturally because the two are solved in different places. Authentication is best centralised in one identity provider (IdP) that all applications trust, so that password rules, multi-factor authentication, account recovery and lockout are implemented once. Authorisation is inherently distributed: only the ordering service knows who owns an order, so the final decision about an order must be made there, even if coarse checks happen earlier at the API gateway. Most access-control breaches in the OWASP Top 10 come from confusing the two: an endpoint that checks the token is valid and then returns whatever object ID it is given.

Passwords, multi-factor authentication and passkeys

Passwords remain the most common authenticator, and NIST's Digital Identity Guidelines are the most widely followed rules for them. Revision 4 of SP 800-63B, published in July 2025, reverses several long-standing habits. Length matters more than composition: a password used as the only factor must be at least 15 characters long (at least 8 when it is used only as part of multi-factor authentication), and verifiers should accept at least 64 characters, including spaces and Unicode. Verifiers must not impose composition rules such as requiring a digit and a symbol, which make passwords harder to remember without making them much harder to guess. They must check new passwords against a blocklist of common, expected and breached passwords. They must not force periodic changes, only a change when there is evidence of compromise. They must allow password managers and should allow pasting, must not offer password hints readable before login, and must not prompt for security questions. Failed attempts must be rate-limited.

Multi-factor authentication (MFA) combines factors of different kinds: something you know (a password), something you have (a phone or security key) and something you are (a fingerprint, usually used only to unlock a device). It defeats most credential stuffing, where attackers try username and password pairs leaked from other sites. Not all second factors are equal: one-time codes sent by SMS can be stolen by SIM swapping, and no one-time code, whether from SMS or an authenticator app, resists phishing, because a user can be tricked into typing it into a fake site that relays it immediately. NIST's guidelines therefore require services at authentication assurance level 2 to offer at least one phishing-resistant option, and level 3 to use one.

A passkey is the phishing-resistant option most users can actually use. Passkeys are FIDO2 credentials, built on the W3C Web Authentication API (WebAuthn) and the FIDO Alliance's Client to Authenticator Protocol (CTAP). At registration the user's device creates a new key pair for that website; the server stores only the public key. At login the server sends a random challenge, and the device signs it with the private key after the user unlocks it with a fingerprint, face or PIN. Because the browser binds the credential to the website's origin, a look-alike phishing site cannot obtain a valid signature, and because the server holds no shared secret, a database breach yields nothing an attacker can replay. Passkeys may be device-bound (on a hardware security key) or synced between a user's devices by a platform or password manager; NIST's 2025 guidance accepts synced passkeys at level 2, while level 3 requires a key that cannot be exported.

The architectural weak point of every authentication scheme is **account recovery**. If a lost passkey can be replaced by clicking an emailed link, the strength of the whole system is the strength of the email account. Recovery flows deserve their own threat model, rate limits, notifications to the user's other devices and, for high-value accounts, delays or human checks.

Sessions and tokens

Once a user has authenticated, the system must recognise their later requests without asking again. There are two basic designs. In a **server-side session** the server stores the session (user, expiry, perhaps roles) and gives the browser a long random session identifier in a cookie. In a **self-contained token** design, the issuer signs a token carrying the claims, typically a JSON Web Token (JWT, RFC 7519), and any service holding the issuer's public key can validate it without a lookup.

Server-side sessions compared with self-contained tokens

Aspect	Server-side session	Self-contained token (JWT)
Where state lives	In a session store on the server; the browser holds only a random identifier.	In the signed token; the server keeps nothing per user.
Revocation	Delete the session: immediate.	Not possible before expiry without a denylist or a lookup, so lifetimes must be short.
Scaling across services	Needs a shared session store or sticky routing.	Any service can validate the signature locally.
Size on the wire	A small cookie.	Hundreds of bytes to kilobytes on every request.
Typical use	Browser sessions for web applications and backends for frontends.	Access tokens for APIs, mobile apps and service-to-service calls.
Main risks	Session fixation and cross-site request forgery (mitigated by regenerating the ID at login and SameSite cookies).	Theft through script injection if stored in browser storage; validation mistakes; stale claims.

For browser applications, a server-side session in a cookie marked `HttpOnly` (invisible to JavaScript), `Secure` (HTTPS only) and `SameSite` is hard to beat. Single-page applications increasingly use a **backend for frontend** (BFF): the browser holds only a session cookie for its own small backend, which holds the OAuth tokens and calls the APIs on its behalf, an approach the IETF OAuth working group's guidance for browser-based applications recommends. JWTs shine between services and in mobile apps, where a stateless check at each hop avoids a central bottleneck.

WATCH OUT

JWT pitfalls

- **No revocation.** A stolen JWT is valid until it expires. Keep access tokens short-lived (minutes), use refresh tokens that the authorisation server can revoke, and keep a denylist of token IDs for emergencies, or use opaque tokens checked by introspection (RFC 7662) where immediate revocation matters.
- **Algorithm confusion.** Some libraries once accepted the algorithm named in the token header, including `none`, or verified an RS256 token as HS256 using the public key as an HMAC secret, letting attackers forge tokens. Always fix the expected algorithms in code, as RFC 8725 (JWT best current practices) advises.
- **Storing tokens in `localStorage`.** Any script running on the page can read it, so one cross-site scripting flaw leaks every user's token. Prefer HttpOnly cookies through a BFF, or keep tokens in memory and bind them to the client with sender-constrained tokens such as DPoP (RFC 9449).
- **Signed is not encrypted.** A JWT's payload is merely Base64url-encoded; anyone holding the token can read it. Do not put personal data in it.
- **Skipping claim checks.** Validate issuer, audience and expiry, not just the signature, or a token issued for one API will be accepted by another.

```python
import jwt  # PyJWT
from jwt import PyJWKClient

ISSUER = "https://id.fooddelivery.example/"
jwks = PyJWKClient(ISSUER + ".well-known/jwks.json")  # issuer's public keys

def verify_access_token(token: str) -> dict:
    signing_key = jwks.get_signing_key_from_jwt(token)
    return jwt.decode(
        token,
        signing_key.key,
        algorithms=["RS256"],      # pinned: never read from the token header
        audience="orders-api",      # the token must be meant for this API
        issuer=ISSUER,
        options={"require": ["exp", "iat", "sub"]},
    )  # raises an exception on any failure: fail closed
```

Validating an access token in a Python API with PyJWT: the algorithm, audience and issuer are fixed by the API, never taken from the token.

IF YOU ARE NEW TO THIS

Step by step: a token nobody can take back

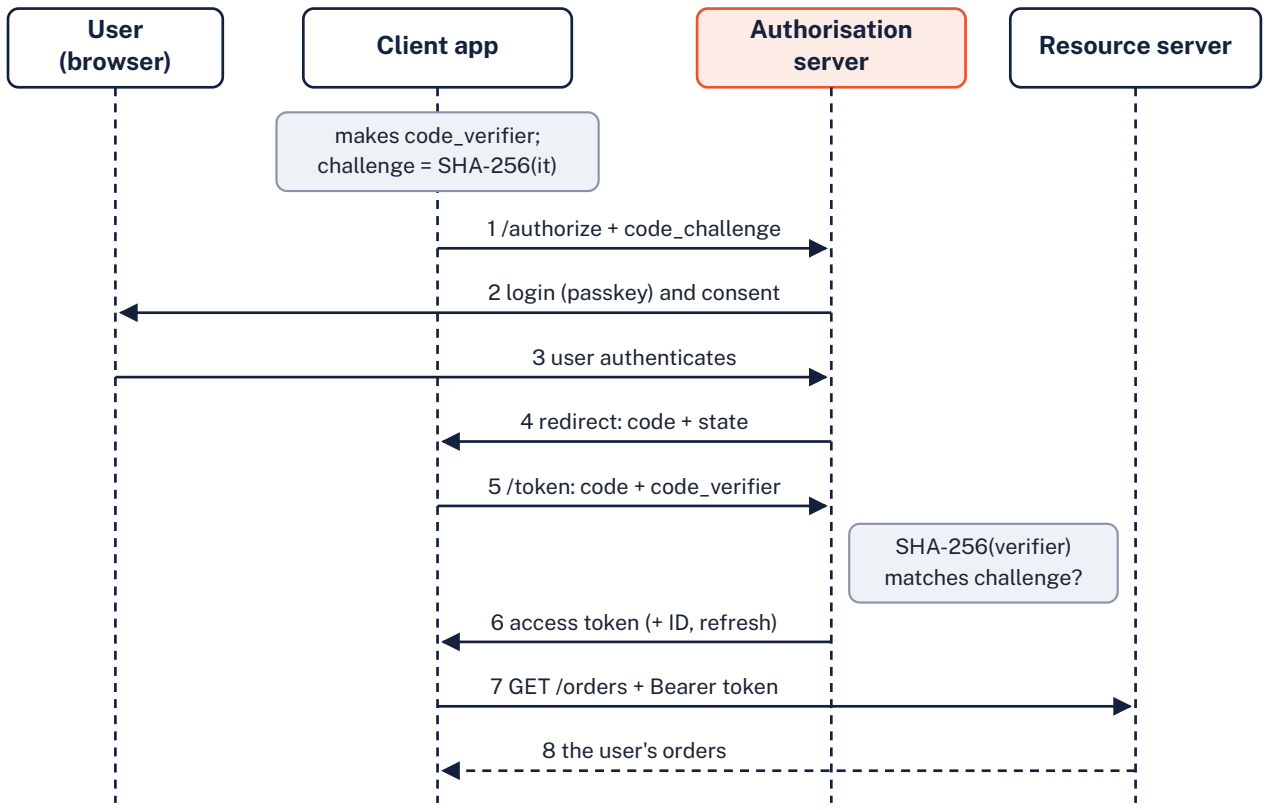
This section's hardest idea is why a self-contained token cannot simply be cancelled, and what designers do about it. Follow one theft under each design in the table.

- With a server-side session, the server records the customer's login in its session store and gives the browser only a long random identifier in a cookie, which it looks up on every request.
- An attacker steals the identifier. Once the theft is noticed, the platform deletes the session record, so the attacker's next request finds nothing and is refused: cancelling is immediate.
- With the other design, the customer's mobile app holds a signed JSON Web Token stating who the customer is and when the token expires. The ordering service checks the signature, much as a shop checks a banknote's watermark, and the token's issuer, audience and expiry, then trusts it without asking anyone.
- Now the token is stolen. There is no record to delete, so every service keeps accepting it, because it is still genuine, until it expires.
- Hence the remedies in 'JWT pitfalls': make access tokens expire within minutes and renew them with a refresh token that the issuer can revoke; where cancelling at once matters, keep a denylist of stolen tokens, or use tokens that each service must check with the issuer, which brings back the very lookup that tokens avoid.
- So it is a trade-off: tokens let each service check a caller on its own, which suits mobile apps and calls between services; sessions are easy to cancel and suit browser applications.
- **The lesson:** on its own, a self-contained token is a promise that cannot be withdrawn before it expires. Keep the promise short, and add a lookup only where cancelling at once matters.

OAuth 2.0: delegated authorisation

OAuth 2.0 (RFC 6749, 2012) solves a specific problem: letting an application act on a user's behalf without learning the user's password. Before it, a service that wanted to import your contacts asked for your email password. OAuth defines four roles: the **resource owner** (the user), the **client** (the application wanting access), the **authorisation server** (which authenticates the user, asks for consent and issues tokens) and the **resource server** (the API holding the data). The client receives an **access token** limited by **scopes** (such as `orders:read`), a lifetime and an audience, and often a **refresh token** to obtain new access tokens without asking the user again.

OAuth defines several ways, called grants, to obtain a token. For any application acting for a user, whether a server-side web app, a single-page app or a mobile app, the recommended grant is the **authorisation code flow with PKCE** (Proof Key for Code Exchange, RFC 7636, 2015). The client first creates a random `code_verifier` and sends only its SHA-256 hash, the `code_challenge`, when it redirects the user to the authorisation server. After the user logs in and consents, the authorisation server redirects back with a short-lived authorisation code. The client then exchanges the code for tokens in a direct back-channel request, sending the original verifier; the server checks that its hash matches the challenge. An attacker who intercepts the code, for example a malicious mobile app registered for the same redirect address, cannot redeem it without the verifier, which never left the client.



The authorisation code flow with PKCE: only the hash of the verifier travels in the browser redirects (the front channel); the verifier itself is sent directly to the token endpoint.

The **client credentials grant** covers the case with no user: a service obtains a token with its own authority, for example the dispatch service calling a mapping partner's API, or a partner's system calling the platform's order-status API. The client should authenticate to the token endpoint with a private key (a signed JWT assertion) or mutual TLS (RFC 8705) rather than a shared secret. Two older grants are now discouraged: the **implicit grant**, which returned tokens in the browser's URL, and the **resource owner password credentials grant**, which had the

client collect the user's password and so defeated the purpose of OAuth. RFC 9700, the OAuth 2.0 Security Best Current Practice published in January 2025, records these lessons, including exact matching of redirect addresses and sender-constrained tokens.

OAuth 2.1 consolidates OAuth 2.0 and these best practices into one document: PKCE is required for the authorisation code flow, the implicit and password grants are removed, redirect addresses must match exactly and bearer tokens may no longer be sent in URL query strings. At the last review of this section it was still an IETF Internet-Draft, not an RFC (check its current status), but its rules already describe what well-run authorisation servers do.

DECISION GUIDE

Decision guide: which OAuth 2.0 flow?

For each kind of caller of the food-delivery APIs.

Q1 **Is the device unable to show a browser or hard to type on, such as a restaurant's kitchen display?**

- **Yes** Use the device authorisation grant (RFC 8628): the device shows a short code that the user approves on their phone.
- **No** Go to the next question.

Q2 **Is a person present, and does the application act on their behalf?**

- **Yes** Use the authorisation code flow with PKCE, for web, single-page and mobile apps alike; never the implicit or password grants.
- **No** Go to the next question.

Q3 **Is it one of your own services calling another inside your platform?**

- **Yes** Prefer workload identity (mutual TLS with platform-issued identities); if the API expects OAuth tokens, use client credentials with key-based client authentication.
- **No** Go to the next question.

Q4 **Is it a partner's system calling your API with its own authority?**

- **Yes** Use client credentials with narrowly scoped, short-lived tokens, one client per partner and per-partner rate limits.
- **No** Revisit the requirement: a design in which your client needs a user's password is a warning sign.

OpenID Connect, single sign-on and SAML

OAuth issues access tokens for APIs; it does not, by itself, tell the client who the user is. Using an access token as proof of login is a well-known mistake, because a token issued to one application can be replayed to another that does not check its audience. OpenID Connect (OIDC, Core 1.0, 2014) adds the missing identity layer. With the `openid` scope, the authorisation server, now acting as an OpenID provider, also returns an **ID token**: a signed JWT for the client, stating the issuer, the user's stable identifier (`sub`), the audience (the client's ID), issue and expiry times, a `nonce` that ties it to this login and, optionally, how and when the user authenticated. OIDC also standardises a user information endpoint and discovery documents that publish the provider's endpoints and signing keys. The rule of thumb: the ID token is for the client, to know who logged in; the access token is for the API, to authorise calls.

Single sign-on (SSO) follows naturally: once a user has a session with the identity provider, every application that trusts it can log them in without another password prompt, and disabling the account in one place cuts access everywhere. In enterprises, SSO often still uses **SAML 2.0** (an OASIS standard from 2005), which exchanges signed XML assertions through the browser. The food-delivery platform might use OIDC for its own apps and accept SAML from large restaurant chains that want their staff to log in to the restaurant portal with their corporate accounts. Provisioning and deprovisioning accounts across systems is usually automated with SCIM (RFC 7643 and 7644).

Authorisation models: RBAC, ABAC and relationships

Authentication tells the system who is calling; an authorisation model decides what they may do. Three models dominate, and real systems usually combine them.

Three authorisation models

Model	How it decides	Strengths	Weaknesses
Role-based (RBAC)	Permissions are granted to roles; users are assigned roles. Formalised by David Ferraiolo and Richard Kuhn at NIST in 1992.	Simple to understand and audit; maps to job functions such as support agent.	Cannot express "only their own" without extra rules; scoping by branch or market leads to role explosion.
Attribute-based (ABAC)	A policy evaluates attributes of subject, resource, action and environment; described in NIST SP 800-162 (2014).	Very expressive: market, data classification, time, device posture, amount.	Harder to answer "who can access this?"; policies can become opaque and slow.
Relationship-based (ReBAC)	Access follows relationships stored as a graph: a user manages a branch, a branch belongs to a restaurant, an order belongs to a branch.	Natural for sharing and hierarchies; scales to fine-grained, per-object permissions.	Needs a dedicated, consistent relationship store and careful modelling.

Relationship-based access control is best known from Google's 2019 paper on **Zanzibar**, the global authorisation system behind Google Drive, Calendar, YouTube and many other products. Zanzibar stores relation tuples of the form *object#relation@user*, evaluates permissions by following them, and uses consistency tokens so that a permission check is never evaluated against data older than the content it protects, the so-called new enemy problem: removing someone from a document and then adding sensitive content must not let them see it. Open-source systems inspired by it include SpiceDB, OpenFGA and Ory Keto.

Whatever the model, it helps to separate the policy decision point (PDP), which evaluates policy and answers allow or deny, from the **policy enforcement point** (PEP), the code in the gateway or service that asks the question and enforces the answer. General-purpose policy engines such as Open Policy Agent (OPA, a graduated Cloud Native Computing Foundation project, with policies written in its Rego language) and Cedar (open-sourced by Amazon Web Services in 2023) let policy live in versioned files, be tested like code and be evaluated in-process or as a sidecar, close to the service. A sensible layering on the platform is: the API gateway rejects requests without a valid token or the right scope (coarse), and each service decides object-level questions with data only it has (fine). Broken object-level authorisation is the first risk in the OWASP API Security Top 10 (2023 edition) precisely because the fine check is the one teams forget.

WORKED EXAMPLE

Worked example: who may change a menu, mark an order ready or see a pickup address

Restaurants on the platform range from single cafés to chains with dozens of branches. Requirements: chain owners manage every branch; branch managers edit their own branch's menu; kitchen staff mark their branch's orders ready; a courier sees the pickup details only for orders assigned to them; managers may issue goodwill refunds up to £20.

First attempt: RBAC	Roles `owner`, `manager`, `staff`, `courier`. Every check then needs "for which branch?" or "which order?", leading to roles such as `manager_branch_1742`: thousands of roles, one per branch and job.
ReBAC tuples	`restaurant:bella#owner@user:ana`; `branch:bella-soho#parent@restaurant:bella`; `branch:bella-soho#manager@user:ben`; `branch:bella-soho#staff@user:kai`; `order:9812#branch@branch:bella-soho`; `order:9812#courier@user:cara`
Model rules	`branch.edit_menu` = manager, or owner of the parent restaurant. `order.mark_ready` = staff or manager of the order's branch. `order.view_pickup` = the assigned courier, or staff of the order's branch.
Checks	Can ben edit bella-soho's menu? Yes (manager). Can ana? Yes (owner of the parent). Can cara see order 9812's pickup? Yes, until Dispatch deletes her courier tuple when the delivery completes; then no.
ABAC for the refund limit	The refund rule depends on an attribute of the request, not a relationship: the policy engine allows `refund` if the user manages the order's branch and the amount is at most £20.
Where enforced	The gateway checks token validity and scope; the menu, ordering and dispatch services each ask the PDP for the object-level decision before acting.

Relationships answer audit questions directly: listing everyone who can edit a branch's menu is a graph query, not a review of code.

Relationship data now has an owner too: Dispatch writes courier tuples, the restaurant portal writes staff tuples, and stale tuples are a security bug.

Workload identity: when the caller is a service

Services authenticate to each other far more often than people log in, yet service credentials are often the weakest: a long-lived API key in an environment variable, shared by several services and never rotated. If that key leaks through a log, a repository or a container image,

the attacker becomes the service. Workload identity replaces such secrets with identities issued by the platform to the running workload, after the platform has verified what the workload is.

The open standard is **SPIFFE** (Secure Production Identity Framework for Everyone), a graduated Cloud Native Computing Foundation project. It names each workload with a URI such as `spiffe://fooddelivery.example/ns/prod/sa/orders` and issues a short-lived, automatically rotated document proving it, called an SVID, usually an X.509 certificate (or a JWT). Its reference implementation, SPIRE, attests the node and the workload (for example its Kubernetes namespace and service account) before issuing the identity. Services then use the certificates for mutual TLS, so the payments service knows that the caller is really the ordering service, and can authorise by identity. Cloud platforms offer equivalents (roles for workloads, managed identities, workload identity federation), and CI systems such as GitHub Actions can exchange a signed OIDC token for short-lived cloud credentials, so pipelines need no stored keys. How these identities are used to build a network with no implicit trust is the subject of zero trust.

Just-in-time and break-glass administrative access

A permanent administrator account is a permanent target. With **just-in-time** access, an engineer holds no standing privilege, only eligibility: to fix a production database they activate a role for a limited time, give a reason, pass multi-factor authentication and, for sensitive roles, get approval, and every step is logged. Microsoft Entra Privileged Identity Management is one example. **Break-glass** accounts cover the day that process fails, for instance because the identity provider is down. Microsoft's guidance is at least two accounts, kept apart from normal sign-in dependencies, protected by phishing-resistant authentication, alerting on every use, and tested regularly. Each use is then reviewed, and the emergency route is itself a risk to be managed.

Authorisation on message topics

Authentication at the broker is not enough: events (Section 6.4) drive real actions, so a service that can publish a forged `PaymentCaptured` event can get orders dispatched unpaid. Grant permissions per topic and per direction: only Payments may write that topic, and only named consumers may read it, which also limits who sees any personal data in it. Apache Kafka expresses this as access control lists stating that a principal may perform an operation, such as Read or Write, on a resource; with an authoriser configured, a resource with no matching list is available only to super users unless `allow.everyone.if.no.acl.found` is set. Defaults differ: in RabbitMQ, topic permissions are opt-in. Check your broker's documentation, and keep these rules in version control.

CHECK YOUR UNDERSTANDING

▼ An API returns 403 to a caller with a valid token. What does that tell you?

The caller was authenticated but is not authorised for this action or object; 401 would mean missing or invalid authentication.

▼ What does PKCE protect against in the authorisation code flow?

Interception or injection of the authorisation code: without the code verifier, which never travels in the browser redirects, a stolen code cannot be exchanged for tokens.

▼ Why should a service not accept an OpenID Connect ID token as an API access token?

The ID token is addressed to the client (its audience is the client ID) to tell it who logged in; APIs should accept only access tokens issued for them, checked for audience and scope.

▼ Why does a role such as manager_branch_1742 suggest the wrong authorisation model?

It encodes a relationship (manager of a particular branch) as a role, which leads to role explosion; relationship-based or attribute-based rules express it directly.

▼ Try it: Pick the OAuth 2.0 flow for: a restaurant's native mobile app used by staff, and a nightly back-end job calling the payments API. A developer also sends the ID token to the orders API. Advise on all three.

The mobile app acts for a user and cannot keep a secret, so use the authorisation code flow with PKCE. The nightly job has no user, so use client credentials or, better, workload identity. The developer is wrong: the ID token tells the client who logged in, and the API should receive the access token, checking issuer, audience and expiry. Keep tokens short-lived, and let each service still make its own object-level checks.

SOURCES

- RFC 6749: The OAuth 2.0 Authorization Framework: https://www.rfc-editor.org/rfc/rfc6749
- RFC 7636: Proof Key for Code Exchange by OAuth Public Clients: https://www.rfc-editor.org/rfc/rfc7636
- RFC 9700: Best Current Practice for OAuth 2.0 Security: https://www.rfc-editor.org/rfc/rfc9700
- NIST SP 800-63B-4: Authentication and Authenticator Management: https://pages.nist.gov/800-63-4/sp800-63b.html
- Zanzibar: Google's Consistent, Global Authorization System (USENIX ATC 2019): https://research.google/pubs/zanzibar-googles-consistent-global-authorization-system/

- Microsoft Entra Privileged Identity Management: https://learn.microsoft.com/entra/id-governance/privileged-identity-management/pim-configure
- Microsoft: Manage emergency access accounts: https://learn.microsoft.com/en-us/entra/identity/role-based-access-control/security-emergency-access
- Apache Kafka: Authorization and ACLs: https://kafka.apache.org/44/security/authorization-and-acls/
- RabbitMQ: Authentication, authorisation and access control: https://www.rabbitmq.com/docs/access-control

SECTION 8.4 OF 8.8 · ADVANCED · 28 MIN READ · LAST REVIEWED NOVEMBER 2026

Encryption, key management and secrets

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 0.5 Storing data
- 8.3 Identity and access

Terms it uses from earlier: TLS and HTTPS, Workload identity, Mutual TLS (mTLS), Passkey, JSON Web Token (JWT)

THE IDEA IN BRIEF

Think of a hotel. Guests' valuables go in room safes, and the safes are only as good as the control over who holds the codes. The hotel does not keep every guest's code in one book at reception; the master key sits in a locked cabinet that only the manager can open, and opening it is written in a log. For the most precious items, the hotel does not keep them at all: it gives the guest a numbered ticket and the bank down the road holds the item. Protecting data works the same way: encrypt it, guard the keys more carefully than the data, and where possible do not hold the most sensitive data at all.

WHAT YOU NEED TO KNOW

- Each layer of encryption protects against particular parties: TLS against the network, storage encryption against stolen media and backups, field-level encryption against database administrators and downstream copies; protection always reduces to control over the keys.
- Symmetric algorithms (AES-GCM) encrypt bulk data; asymmetric cryptography provides key agreement and digital signatures; NIST's first post-quantum standards (FIPS 203 to 205, 2024) make crypto agility an architectural concern.
- Hashing, MACs, signatures, encryption and encoding are different tools; passwords need slow, salted password hashing such as Argon2id with OWASP's current parameters, or better, passkeys through an identity provider.
- Envelope encryption wraps per-record data keys under a key-encryption key held in a KMS backed by HSMs, making rotation cheap, limiting blast radius and enabling crypto-shredding, at the cost of a critical dependency.
- Secrets belong in a vault, preferably as short-lived dynamic credentials or replaced by workload identity; a leaked secret must be revoked first and cleaned up afterwards.

- Tokenisation through a payment provider keeps card numbers out of the platform and shrinks PCI DSS v4.0.1 scope; a data classification scheme decides where field-level encryption is worth its costs.

What encryption protects, and what it does not

Encryption is often treated as a box to tick: "data is encrypted in transit and at rest". The phrase hides the question that matters, which is **against whom**. Every layer of encryption protects data from some parties and not from others, and in every case the protection is exactly as strong as the control over who can use the key. Encryption does not remove the access-control problem; it moves it from the data to the keys, where it is usually easier to manage, audit and revoke.

Encryption **in transit** protects data moving across a network from being read or altered on the way. On the web that means TLS, introduced in Chapter 0; between services it means TLS or mutual TLS, discussed with zero trust. Encryption **at rest** protects stored data, but at rest is not one thing: the table shows the common layers on the food-delivery platform.

Layers of encryption and the threats each addresses

Layer	Protects against	Does not protect against
TLS in transit	Eavesdropping and tampering on the network, including inside the data centre.	A compromised service or endpoint, which sees the data in plaintext.
Disk or volume encryption (default in major clouds)	Stolen or discarded disks, hardware returned to the provider.	Anyone who reaches the data through the running operating system or database.
Database or storage-service encryption (for example transparent data encryption)	Copies of data files, snapshots and backups taken at the storage layer.	Any database account that can run queries, including the application's own and an attacker using SQL injection.
Application-level (field-level) encryption	Database administrators, over-broad accounts, analytics copies and leaked backups, because they hold only ciphertext.	The application itself if compromised, since it can use the key; the database can no longer search or sort the field.
End-to-end encryption	Every intermediary, including the platform operator.	Nothing in between, but the platform can then do nothing useful with the data.

Reading down the table, each layer protects against more parties and costs more functionality. A sound design uses the cheap, transparent layers everywhere (TLS and storage encryption are close to free today) and the expensive ones selectively, for the data whose exposure would do the most harm. Deciding which data that is is the job of data classification, covered later in this section together with tokenisation and a worked example.

Symmetric and asymmetric cryptography

Symmetric encryption uses one secret key to encrypt and decrypt. It is fast enough for bulk data, and the standard algorithm is AES (FIPS 197, 2001) with 128- or 256-bit keys. Modern designs use **authenticated encryption**, which detects tampering as well as hiding content: AES in Galois/Counter Mode (AES-GCM) or ChaCha20-Poly1305 (RFC 8439). Authenticated modes need a unique nonce for every message encrypted under the same key; reusing a nonce with AES-GCM can reveal plaintext and allow forgeries, which is one of many reasons to use well-tested libraries rather than assembling primitives yourself.

Asymmetric cryptography uses a key pair: a public key that anyone may hold and a private key kept secret. It enables two things symmetric cryptography cannot. **Key agreement** lets two parties who have never met establish a shared symmetric key over an open network (elliptic-curve Diffie-Hellman in TLS 1.3). **Digital signatures** (RSA, ECDSA, Ed25519) let anyone with the public key verify that data came from the private key's holder and was not changed, which gives integrity, authenticity and a basis for non-repudiation. Asymmetric operations are slow, so real protocols are hybrid: TLS uses asymmetric cryptography to authenticate the server and agree keys, then symmetric encryption for the data.

A large quantum computer would break today's RSA and elliptic-curve algorithms, though not AES with sufficiently long keys. In August 2024 NIST published its first post-quantum standards: FIPS 203 (ML-KEM, for key establishment), FIPS 204 (ML-DSA) and FIPS 205 (SLH-DSA), both for signatures. Data that must stay confidential for many years is at risk today from attackers who record encrypted traffic now to decrypt it later, so the architectural response is **crypto agility**: keep an inventory of where which algorithms are used, avoid hard-coding them, and prefer libraries, protocols and providers that can switch, as major browsers and TLS libraries have begun to do with hybrid post-quantum key exchange.

Hashing, MACs, signatures and encryption are different tools

Confusing these primitives is a common source of design flaws, such as "encrypting" passwords so that they can be decrypted, or trusting a plain hash to prove that a webhook came from a payment provider. The table sets them side by side.

Choosing the right primitive

Tool	Reversible?	Key	Use it for	Example
Encoding (Base64, hex)	Yes, by anyone	None	Representing bytes as text. Provides no security.	A JWT payload
Cryptographic hash (SHA-256)	No	None	Fingerprints, integrity of files against accidental change, content addressing. MD5 and SHA-1 are broken for collision resistance.	Checksum of a downloaded release
Message authentication code (HMAC-SHA-256)	No	Shared secret	Proving a message came from someone holding the key and was not changed.	Verifying a payment provider's webhook
Digital signature (Ed25519, ECDSA)	No (it is verified)	Private key signs, public key verifies	Proof of origin that anyone can check without being able to forge it.	JWT access tokens, signed container images
Encryption (AES-GCM)	Yes, with the key	Secret (or a key pair, for hybrid schemes)	Confidentiality, plus integrity in authenticated modes.	A courier's identity document in storage
Password hash (Argon2id)	No, and deliberately slow	Per-password salt, optional pepper	Storing passwords so that a stolen database resists guessing.	The identity provider's credential store

Storing passwords

A general-purpose hash such as SHA-256 is designed to be fast, which is exactly wrong for passwords: graphics processors can test billions of guesses per second against a stolen table of fast hashes. Password hashing uses functions that are deliberately slow and, ideally, memory-hard, so that each guess costs an attacker real time and memory, with a unique random **salt** per password so that identical passwords produce different hashes and precomputed tables are useless.

The OWASP Password Storage Cheat Sheet gives current recommendations in order of preference. **Argon2id** (a variant of Argon2, winner of the 2015 Password Hashing Competition), with a minimum of 19 MiB of memory, two iterations and one degree of parallelism (or an equivalent trade-off, such as 46 MiB with one iteration). **scrypt** where Argon2id is unavailable,

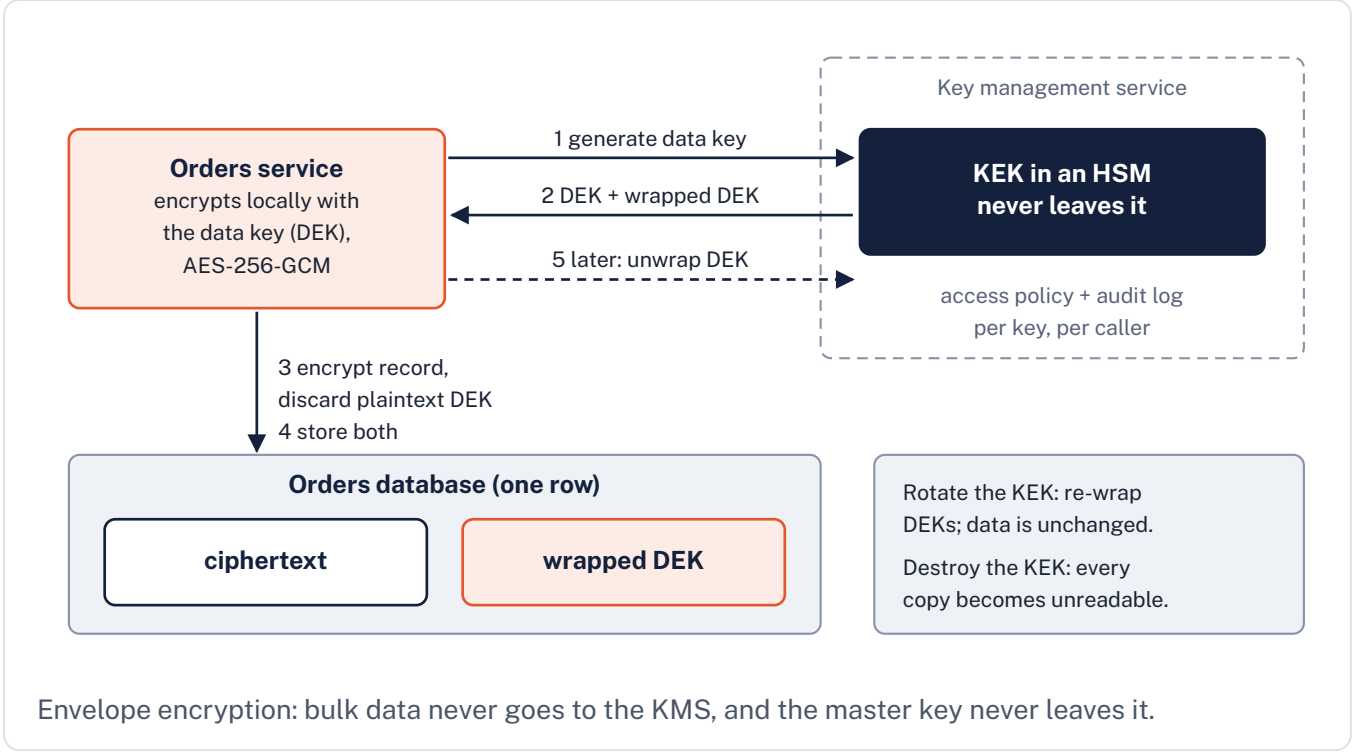
with a cost parameter of 2^{17} (128 MiB), a block size of 8 and parallelism of 1. **bcrypt** only for legacy systems, with a work factor of at least 10, remembering that most implementations ignore input beyond 72 bytes. Where FIPS 140 compliance is required, **PBKDF2** with HMAC-SHA-256 and 600,000 iterations. These figures rise as hardware improves, so check the current cheat sheet rather than copying numbers into a design document.

Three design points follow. Store the algorithm and parameters with each hash (the standard encoded formats do this), so that parameters can be raised later: when a user logs in successfully, verify against the old hash and immediately rehash with the new parameters. Consider a **pepper**, a secret added to every password before hashing and kept in a vault or HSM rather than the database, which defeats an attacker who steals only the database. Most important, ask whether you should store passwords at all: delegating authentication to an identity provider and offering passkeys removes the most attacked secret from most of your services.

Key management and envelope encryption

Keys have a lifecycle, described in NIST SP 800-57 Part 1: generation (from a proper random source), storage, distribution, use, rotation at the end of a defined cryptoperiod, revocation if compromised, and destruction. Doing all of that safely by hand is impractical, which is why architectures rely on a key management service (KMS) such as AWS KMS, Azure Key Vault or Google Cloud KMS, or on a self-run system such as HashiCorp Vault. A KMS holds master keys inside hardware security modules (HSMs), tamper-resistant devices validated against FIPS 140, and performs operations with those keys on request, subject to an access policy, so that applications never see the master keys and every use is logged.

Sending every record to the KMS to be encrypted would be slow and expensive, and KMS APIs limit request sizes. Envelope encryption solves this. The application asks the KMS for a fresh **data encryption key** (DEK); the KMS returns it twice, in plaintext and wrapped (encrypted) under a **key-encryption key** (KEK) that never leaves the HSM. The application encrypts the data locally with the plaintext DEK, discards the DEK from memory and stores the ciphertext with the wrapped DEK. To decrypt, it sends only the small wrapped DEK to the KMS, which checks the caller's permission, logs the request and returns the DEK.



Envelope encryption: bulk data never goes to the KMS, and the master key never leaves it.

```
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

def encrypt_field(plaintext: bytes, kms, kek_id: str, context: bytes) → dict:
    dek = AESGCM.generate_key(bit_length=256)            # fresh data key
    nonce = os.urandom(12)                               # never reuse with this key
    ct = AESGCM(dek).encrypt(nonce, plaintext, context)  # context binds it to its row
    wrapped = kms.encrypt(key_id=kek_id, plaintext=dek)  # KEK stays inside the KMS
    return {"ct": ct, "nonce": nonce, "wrapped_dek": wrapped, "kek": kek_id}

def decrypt_field(rec: dict, kms, context: bytes) → bytes:
    dek = kms.decrypt(key_id=rec["kek"], ciphertext=rec["wrapped_dek"])  # checked,
logged
    return AESGCM(dek).decrypt(rec["nonce"], rec["ct"], context)
```

Envelope encryption of one field with the Python cryptography library; kms stands for any KMS client (real KMS APIs combine the first two steps in one generate-data-key call).

Envelope encryption has several architectural benefits. **Rotation is cheap**: rotating a KEK creates a new key version for new wrapping operations, while old versions remain available for decryption; existing DEKs can be re-wrapped without touching the data, and full re-encryption is needed only if a DEK itself may have been exposed. **Blast radius is limited**: with a DEK per record, file or partition, one leaked DEK exposes little. **Separation of duties** becomes possible: the ordering service may use its KEK to unwrap but not change its policy or delete it, while security administrators can manage keys but cannot decrypt data. **Key hierarchies** map onto the organisation: a KEK per service and data class, or per tenant, so that a large restaurant chain could even supply and control its own key. And destroying a key is a powerful tool:

crypto-shredding makes every copy of the data encrypted under it unreadable, including backups and immutable event logs, which matters for erasure under the GDPR and for event-sourced systems.

The price is a new critical dependency. If the KMS is unavailable, encrypted data cannot be read; if a key is deleted by mistake, the data is gone for good. Cloud KMSs therefore enforce a waiting period before deletion, and architects should treat key deletion and key policy changes as privileged, logged, two-person operations, cache unwrapped DEKs briefly where the latency of KMS calls matters, and include the KMS in disaster recovery plans.

IF YOU ARE NEW TO THIS

Step by step: a key that locks other keys

The idea that needs the most care here is envelope encryption: the data is locked with its own data key, and that key is itself locked with a master key that never leaves the key management service (KMS). Follow one restaurant's payout bank details in and out of storage.

- To store a restaurant's new bank details, the payments service asks the KMS for a fresh data key. The KMS returns it twice: in plain form, and wrapped, meaning locked under the payments master key, which stays inside the KMS's tamper-resistant hardware.
- The service locks the details with the plain data key on its own machine, erases that key from memory, and stores the locked details together with the wrapped key. The details never travel to the KMS.
- Anyone who copies that row holds locked details and a locked key, useless without permission to use the master key.
- When the payout job needs the details, it sends only the small wrapped key to the KMS, which checks that this job may use the master key, logs the request and returns the plain data key.
- Two benefits follow. Rotating the master key need not touch the stored details: the small data keys can simply be re-wrapped. And destroying a key makes every copy of the data encrypted under it unreadable, backups included: crypto-shredding.
- The price is a new critical dependency. If the KMS is unavailable, the details cannot be read; if a key is deleted by mistake, the data is gone for good.
- **The lesson:** keep bulk data local and keys central. Each record can have its own data key, while the master key stays inside the KMS, where every use is checked and logged.

Secrets management

A secret is any credential that grants access: database passwords, API keys for the payment provider, private keys, signing keys, OAuth client secrets, webhook secrets. Secrets management starts from one rule: **secrets never live in source code**, nor in container images, nor in configuration files committed to repositories, because those are copied, cached, forked and read by many more people and systems than the secret should be. Once in a repository's history, a secret stays there after the line is deleted.

Secrets belong in a vault (HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, Google Secret Manager) that stores them encrypted, releases them only to authenticated workloads according to policy, logs every access and supports rotation. Better than rotating long-lived secrets is not having them: a vault can issue **dynamic secrets**, such as a database account created for one service instance with a lease of an hour, and workload identity lets services authenticate to cloud APIs and to each other without any stored secret at all. Secrets injected at run time should be read from files or the vault's API rather than broadly visible environment variables, which tend to end up in crash dumps and debug logs.

Prevention needs automation, because people make mistakes. Pre-commit hooks and pipeline scanners (such as gitleaks and detect-secrets) catch secrets before they are pushed; repository hosts scan for known secret formats, and GitHub's push protection blocks pushes that contain them. Public repositories are scanned by attackers within minutes of a push. The consequences are well documented: in 2016 attackers who gained access to a private GitHub repository used by Uber engineers found an access key for Uber's cloud storage there and downloaded personal data of about 57 million riders and drivers.

Checklist: when a secret has leaked

Treat any secret that reached a repository, log, ticket or chat as compromised, whether or not the repository is private.

- [ ] **Revoke or rotate the secret immediately**
  Do this first. Deleting the commit or message does not help: clones, forks, caches and scanners already have it.
- [ ] **Establish what it could reach**
  List the systems, data and permissions the secret granted, and any other secrets reachable from there.
- [ ] **Review usage since exposure**
  Check the provider's and the vault's access logs for use from unexpected identities, addresses or times since the secret first appeared.
- [ ] **Contain and clean up**
  Rotate dependent credentials, invalidate sessions and tokens issued with it, then remove the secret from history and caches (after rotation, not instead of it).
- [ ] **Decide on notification**
  If personal data may have been accessed, start the breach assessment; under the GDPR the clock for notifying the supervisory authority starts when you become aware.
- [ ] **Fix the cause**
  Move the secret to a vault, replace it with workload identity or a dynamic secret, and turn on secret scanning and push protection so the same mistake is blocked next time.

Tokenisation and reducing PCI DSS scope

Tokenisation replaces a sensitive value with a token that has no mathematical relationship to it; the real value is kept in a separate, tightly protected vault, and only the vault can map the token back. Unlike encryption, a stolen token cannot be reversed with a key, because there is no key, only a lookup in a system the attacker does not control.

Its most important use is card payments. The Payment Card Industry Data Security Standard (PCI DSS, currently version 4.0.1, published in June 2024) applies to every system component that stores, processes or transmits cardholder data, or that is connected to or could affect the security of those components. A platform that lets card numbers pass through its servers brings its API gateway, services, logs, networks and administrators into scope, with hundreds of requirements and an annual assessment. A platform that uses the payment provider's

hosted payment fields or redirect, so that card numbers go straight from the customer's device to the provider, receives only a token and perhaps the last four digits, and its PCI DSS scope shrinks dramatically: a smaller merchant may qualify for the shortest self-assessment questionnaire, and even the largest merchants, which still need an annual assessment by a qualified assessor, have far fewer systems to assess. This is the design in the checkout threat model: the cheapest card data to protect is card data you never hold. Encrypting card numbers yourself does not achieve the same reduction, because whoever can decrypt them remains in scope.

The same idea works for other sensitive identifiers. An internal tokenisation service can replace national insurance numbers or passport numbers with tokens throughout the platform, so that only one small, heavily protected service ever sees the real values.

Data classification

Encryption, tokenisation, logging, retention and access decisions all depend on knowing which data is how sensitive. A **data classification** scheme gives a few levels with handling rules attached, so that teams do not decide each case from scratch. Four levels are common; what matters is that every data set in the platform's data catalogue or schema carries a level, and that the rules are enforced by templates and checks rather than by memory.

A data classification scheme for the food-delivery platform

Level	Examples	Handling rules
Public	Menus, prices, restaurant opening hours	Integrity protection only; may be cached anywhere.
Internal	Aggregated sales figures, service configuration without secrets	Staff and services only; not in public logs.
Confidential	Customer names, addresses, phone numbers, order histories, courier locations	TLS and storage encryption; least-privilege access; masked in support tools; minimised in analytics; retention limits.
Restricted	Payout bank details, courier identity documents, secrets, payment tokens	All of the above plus field-level encryption with a dedicated key, access by named services only, every access logged.

Field-level encryption

Field-level encryption, the application-level layer from the first table, encrypts individual fields before they are written, typically with envelope encryption and a KEK per data class. Its strength is that everything downstream of the application holds only ciphertext: database

administrators, read replicas, backups and the data warehouse fed by change data capture. A breach of any of them exposes nothing readable, and access to the plaintext is reduced to the services that can call the KMS with that key, each call logged.

Its costs are real. The database can no longer index, search, sort or join on the encrypted field. Equality lookups can be restored with a **blind index**, a separate column holding a keyed hash (HMAC) of the normalised value, at the price of revealing which rows share a value. Deterministic encryption has the same leak, while range queries need specialised schemes or should simply not be done on such fields. Some databases offer client-side field-level or queryable encryption, which packages these techniques. Because of these costs, field-level encryption should be reserved for restricted data, whose fields are rarely queried anyway.

WORKED EXAMPLE

Worked example: what to encrypt, where, and with which keys

The food-delivery platform's architect reviews the main data sets against the classification scheme. Every service already uses TLS (mutual TLS internally) and encrypted storage with a KMS key per service, so the question is what needs more.

Card numbers	Never held. Hosted payment fields send them to the payment provider; Payments stores the provider's token, the card brand and the last four digits. PCI DSS scope shrinks to the checkout integration with the provider.
Customer passwords	Not stored by product services. The identity provider offers passkeys; remaining passwords are hashed with Argon2id at OWASP's parameters, with a pepper held in an HSM.
Addresses and phone numbers (confidential)	Storage encryption with Orders' KEK; least-privilege accounts; masked in the support tool; the warehouse receives postcode districts only, not full addresses.
Restaurant payout bank details (restricted)	Field-level envelope encryption under a dedicated Payments KEK; only the payout job may unwrap; changes require re-authentication and notify the restaurant owner, protecting integrity as well as confidentiality.
Courier identity documents (restricted)	Stored as objects, each with its own DEK wrapped under an onboarding KEK; only the verification service can decrypt; every access is logged and reviewed monthly; deleted after the legal retention period.
Personal data in events and the event log	Events carry customer IDs rather than addresses; the few personal fields that must travel are encrypted with a per-customer key, so that destroying the key erases them from the retained log.
Backups	Encrypted with a separate KEK held in a separate cloud account, so that compromising production credentials does not expose backups; restores are tested quarterly.
Service secrets	In the vault; databases use dynamic credentials with one-hour leases; the payment provider's API key is readable only by the Payments workload identity.

About ten KEKs, each owned by one service and rotated automatically every year, cover the whole platform; the leverage is in the decisions, not in the number of keys.

Two decisions remove data rather than protect it (card numbers and full addresses in analytics), which is always the strongest option.

The design is recorded as one ADR per restricted data set and as the data protection concept in the architecture description.

CHECK YOUR UNDERSTANDING

▼ Which parties does transparent database encryption not protect against?

Anyone who can query the database through a valid account, including the application, an attacker using SQL injection and database administrators, because the database decrypts data for them.

▼ Why is rotating a key-encryption key cheap in envelope encryption?

Only the small wrapped data keys need re-wrapping under the new key version; the bulk data stays encrypted under its unchanged data keys.

▼ A developer proposes storing passwords encrypted with AES so that support staff can tell users their password. What is wrong?

Passwords must be stored with a one-way, slow, salted password hash so that nobody, including staff and anyone who steals the key, can recover them; recovery should reset, not reveal.

▼ Why does using a payment provider's hosted fields reduce PCI DSS scope more than encrypting card numbers yourself?

With hosted fields, card numbers never reach the platform's systems, so they are not in scope; encrypted card numbers that the platform can decrypt keep its systems in scope.

▼ Try it: The platform already uses TLS and encrypted storage. Decide what more to do for customer card numbers, customer passwords and customers' allergy notes. For each, give the protection and the threat it addresses. Assume the classification rates allergy notes as sensitive.

Card numbers: never hold them; use hosted payment fields and the provider's token, which shrinks PCI DSS scope. Passwords: no encryption, but a slow salted hash such as Argon2id, or passkeys through an identity provider, so a stolen table is of little use. Allergy notes: field-level encryption with envelope keys in a KMS, protecting against administrators and downstream copies. It costs searchability and adds a dependency on the KMS, so reserve it for sensitive fields.

SOURCES

- OWASP Password Storage Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- OWASP Secrets Management Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

- NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final
- Google Cloud KMS: Envelope encryption: https://cloud.google.com/kms/docs/envelope-encryption
- PCI Security Standards Council: https://www.pcisecuritystandards.org/

SECTION 8.5 OF 8.8 · ADVANCED · 34 MIN READ · LAST REVIEWED NOVEMBER 2026

Secure communication and zero trust

BEFORE YOU READ THIS

This section builds on:

- 0.4 How computers talk
- 8.3 Identity and access
- 8.4 Protecting data

Terms it uses from earlier: TLS and HTTPS, OAuth 2.0, Workload identity, Fallacies of distributed computing, API gateway

THE IDEA IN BRIEF

Imagine a hotel where the front desk checks your identity once and then hands you a key that opens every room in the building. If a thief gets past the desk, every room is open. A better hotel gives you a key card that opens only your room, only for the nights you booked, and records every door it opens. Zero trust applies the second idea to software: no request is trusted because it comes from inside the company network. Every person and every program must prove who it is and be allowed to do that particular thing, every time.

WHAT YOU NEED TO KNOW

- The perimeter model trusts anything inside the network, so one stolen credential or compromised machine allows lateral movement; cloud, mobile and remote work have also removed any single perimeter to defend.
- NIST SP 800-207 (August 2020) defines zero trust as granting no implicit trust based on network location or ownership, with seven tenets including securing all communication, per-session access and dynamic, strictly enforced policy.
- A policy enforcement point in front of each resource asks a policy decision point (policy engine plus policy administrator), which decides using identity, device or workload posture, threat data and policy.
- Google's BeyondCorp (2014) applied zero trust to staff access with device inventory, an access proxy and trust tiers, and removed the privileged office network and the VPN.
- Mutual TLS gives every service call encryption and a verified identity at both ends; SPIFFE standardises short-lived workload identities, and service meshes automate mTLS and per-call policy at a real operational cost.

- Certificates must be issued and renewed automatically (ACME, RFC 8555), especially as public TLS certificate lifetimes fall to 200 days in 2026, 100 days in 2027 and 47 days in 2029.
- API gateways authenticate tokens, rate-limit and validate input, but object-level authorisation must still happen in the service that owns the data.

Why the network perimeter stopped working

For decades most organisations protected their systems like a castle with a moat. A firewall separated the trusted inside from the untrusted internet, staff connected to the inside through a virtual private network (VPN), and anything already inside was treated as friendly. Services in the data centre called each other over plain HTTP with no authentication, because the network itself was the credential. Chapter 6 listed "the network is secure" among the fallacies of distributed computing; the perimeter model is that fallacy turned into an architecture.

Three things broke it. First, the inside became too large to defend: laptops travel, staff work from home, partners connect, and a single phishing e-mail or stolen VPN password puts an attacker inside the moat. Once there, the attacker moves sideways from machine to machine, which is called lateral movement. In the 2013 breach at the US retailer Target, attackers first used network credentials stolen from a heating and ventilation contractor and from there reached the payment terminals, taking data on about 40 million cards. Second, the inside dissolved: workloads now run across several cloud accounts, SaaS products and third-party APIs, so there is no single wall to put a firewall on. Third, the inside became dynamic: containers appear and disappear every minute and IP addresses are reused, so rules written as *allow 10.0.4.0/24 to reach 10.0.7.12* describe nothing stable.

The idea that the perimeter should be abandoned is older than the name. The Jericho Forum, a group of corporate security officers, argued for *de-perimeterisation* from 2004; John Kindervag, then an analyst at Forrester Research, coined *zero trust* around 2010; and Google's BeyondCorp papers from 2014 showed it working at scale. The slogan is *never trust, always verify*: no request is trusted because of where it comes from.

IF YOU ARE NEW TO THIS

If you are new to this

Think of an office building. The old approach checks your badge at the front door and then lets you into every room. Zero trust checks your badge, and asks whether you should be there right now, at every door you open. The building no longer has a single trusted inside, so a thief who slips in behind someone at the front door still cannot get into the finance office.

The rest of this section uses some terms from earlier sections: TLS and certificates (Section 0.4), tokens, OAuth 2.0 and workload identity (Section 8.3) and encryption and keys (Section 8.4).

Zero trust as NIST defines it

The reference text is NIST Special Publication 800-207, *Zero Trust Architecture*, published in August 2020 (Rose, Borchert, Mitchell and Connelly). It describes zero trust as a set of principles that move defences from static, network-based perimeters to users, assets and resources, and a zero trust architecture as an organisation's plan for applying them. Its central claim is that no implicit trust is granted to an asset or user account based only on its physical or network location or on who owns it. Authentication and authorisation are discrete functions performed before a session to a resource is established.

The publication lists seven tenets. They are worth knowing in their own words because many products claim to be zero trust while meeting only one or two of them.

The seven tenets of NIST SP 800-207, with the food-delivery platform

Tenet (paraphrased)	What it means	Food-delivery example
1. All data sources and computing services are resources	A laptop, a container, a SaaS tool and a database are all things to protect, not only servers in the data centre.	The courier app's backend, the restaurant tablet API and the analytics bucket are each resources with their own policy.
2. All communication is secured regardless of network location	Traffic inside the data centre gets the same encryption and authentication as traffic over the internet.	Orders calls Payments over mutual TLS even though both run in the same cluster.
3. Access is granted per session	Trust is evaluated for each session with the least privilege needed, and granting one resource grants nothing else.	A support agent's session can read one customer's order history; it cannot also reach the refunds API.
4. Access is decided by dynamic policy	The decision uses the identity, the state of the device or workload and other attributes such as time, location and behaviour.	A refund over a set amount from an unmanaged laptop is refused; the same request from a managed device with fresh MFA is allowed.
5. The enterprise monitors the integrity and posture of all assets	No device or workload is trusted by default; its patch level, configuration and integrity are measured continuously.	A container image without a valid signature, or with a critical unpatched CVE, is not admitted to the cluster.
6. Authentication and authorisation are dynamic and strictly enforced before access	Identity is checked before every access and re-checked during long sessions; credentials are short-lived and rotated.	Service certificates live for hours and are renewed automatically; user sessions are re-evaluated when risk changes.
7. The enterprise collects as much information as possible about the state of assets, network and communications	Telemetry feeds back into the policy so that decisions improve over time.	Access logs from the gateway, the mesh and the identity provider flow to the security monitoring platform.

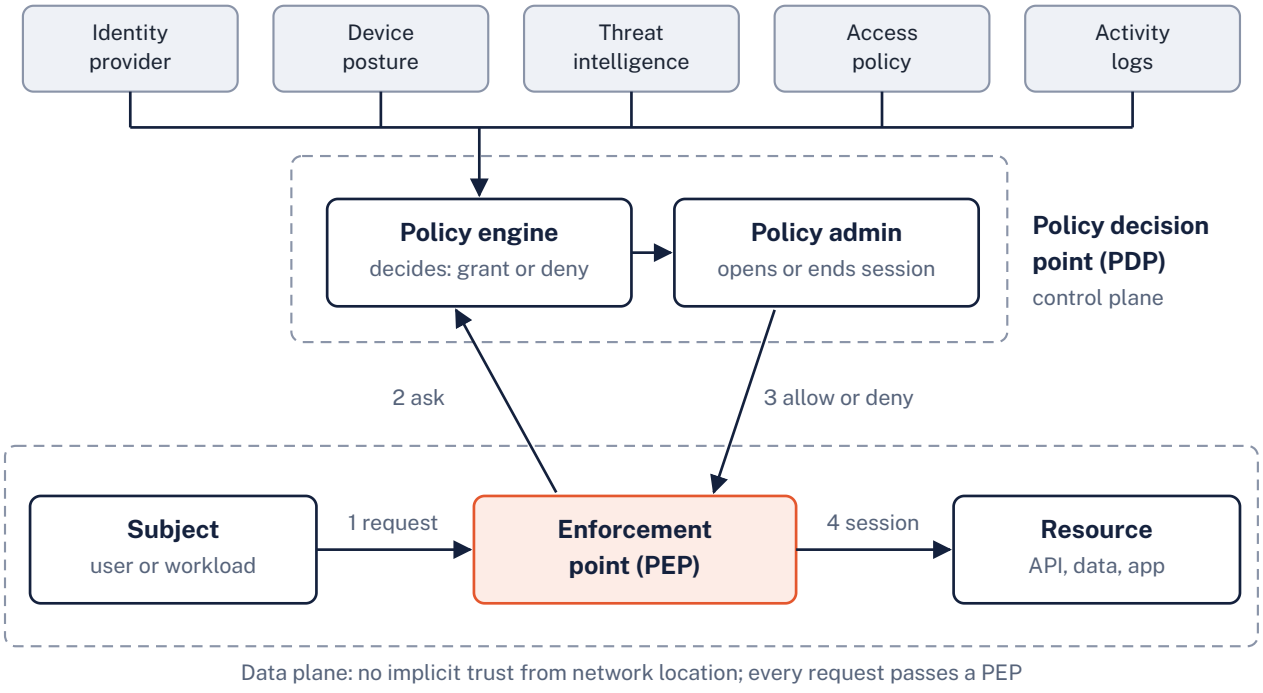
Policy decision and enforcement points

SP 800-207 describes the machinery with a small set of logical components. A policy enforcement point (PEP) sits in front of a resource and lets traffic through only when told to: in practice an identity-aware proxy, an API gateway, a service mesh sidecar or a database proxy. The policy decision point (PDP) makes the decision. NIST splits it into a policy engine, which evaluates the request against policy and inputs and decides, and a policy administrator, which carries out the decision by opening or closing the path, for example by issuing a short-lived

session token or configuring the PEP. The inputs include the identity provider, device and workload posture (patch level, configuration, integrity), threat intelligence, the access policies themselves and activity logs.

Separating decision from enforcement is the same move as separating policy from mechanism in operating systems, and it pays off for the same reason: policies can be written once, reviewed, tested and changed centrally, while enforcement is spread to every place traffic flows. The policy engines of Section 8.3, such as Open Policy Agent, are often the engine behind a PDP. The design question for an architect is where to put the enforcement points so that there is no path around them, and how to keep the decision fast and available, because a PDP on the critical path of every request is now a dependency with its own availability and latency budget. Common answers are to cache decisions for seconds, to push compiled policies to the PEP so that it can decide locally, and to decide in advance what happens if the PDP is unreachable (normally deny, with a small allow-list for health checks).

The logical components of a zero trust architecture



Requests travel only through an enforcement point; the decision is made in a separate control plane fed by identity, posture, threat and policy data.

BeyondCorp: zero trust for people

Google began moving away from its privileged corporate network after the 2009 intrusion known as Operation Aurora and described the result in *BeyondCorp: A New Approach to Enterprise Security* (Rory Ward and Betsy Beyer, ;login:, December 2014), followed by several further papers. The design has three main parts. Every device is enrolled in a device inventory

and identified by a certificate. An access proxy, reachable from the internet, fronts every internal application; employees reach applications through it from any network, including the office network, which gets no special privileges. A trust inference component combines who the user is with the state of the device to assign a trust tier, and each application declares the minimum tier it requires.

The consequence that surprised people was the removal of the VPN: an employee in a café and an employee at a desk in the office go through the same checks. The idea now ships as products under the names identity-aware proxy and zero trust network access (ZTNA). The US Cybersecurity and Infrastructure Security Agency's Zero Trust Maturity Model (version 2.0, April 2023) is a useful self-assessment tool: it scores five pillars (identity, devices, networks, applications and workloads, and data) from traditional to optimal, with visibility, automation and governance across all of them.

Mutual TLS, certificates and workload identity

Zero trust between services needs two things on every call: encryption, and a strong identity for both ends. Transport Layer Security does both. TLS 1.3 (RFC 8446, August 2018) completes its handshake in one round trip: the client sends its supported algorithms and a key share, the server answers with its key share and its certificate, and both derive session keys with ephemeral Diffie-Hellman, so every session has forward secrecy (a stolen server key cannot decrypt recorded past traffic). TLS 1.3 removed the weak options of earlier versions, including RSA key exchange, CBC-mode ciphers, RC4, SHA-1 signatures, compression and renegotiation; TLS 1.0 and 1.1 were formally deprecated by RFC 8996 in 2021. An optional zero round-trip (0-RTT) mode lets a returning client send data immediately, but that early data can be replayed by an attacker, so it must be limited to idempotent requests. Major browsers and cloud providers have also begun using hybrid post-quantum key exchange that combines X25519 with ML-KEM (standardised by NIST as FIPS 203 in 2024), which protects today's recorded traffic against a future quantum computer.

A certificate binds a public key to a name and is signed by a certificate authority (CA); the client checks the chain of signatures up to a root it already trusts. This system of CAs, certificates, trust stores and revocation is a public key infrastructure (PKI). Public certificates are also logged in Certificate Transparency logs, so a mis-issued certificate for your domain can be spotted. Ordinary HTTPS authenticates only the server. In mutual TLS (mTLS) the client also presents a certificate, so the server learns which workload is calling. Internal certificates come from a private CA that the organisation runs, usually inside its secrets or mesh platform, because public CAs will not issue certificates for internal names.

What name goes in the certificate? Hostnames and IP addresses are poor identities for containers. SPIFFE (Secure Production Identity Framework for Everyone, a graduated project of the Cloud Native Computing Foundation) standardises workload identity: each workload gets a URI such as `spiffe://fooddelivery.example/ns/prod/sa/orders` and a short-lived SPIFFE Verifiable Identity Document, either an X.509 certificate for mTLS or a JWT. Its reference implementation, SPIRE, attests a workload (checks which Kubernetes service account or cloud instance it really is) before issuing the document, and rotates it automatically, typically every hour or so. Authorisation rules can then say *orders may call payments* instead of naming addresses.

Service meshes

Implementing mTLS, certificate rotation, retries and per-call authorisation inside every service, in every language, is expensive and inconsistent. A service mesh moves that work into infrastructure. In the classic design each service instance gets a sidecar proxy (Envoy in Istio, a Rust micro-proxy in Linkerd) that intercepts its traffic; a control plane issues identities, distributes policies and collects telemetry. Newer sidecar-less designs, such as Istio's ambient mode, put a shared per-node proxy in charge of mTLS and add a separate proxy only where HTTP-level policy is needed. NIST SP 800-207A (September 2023) describes this pattern as the way to apply zero trust to cloud-native applications: identity-based segmentation, with policies enforced by the mesh at every call.

A mesh is a significant platform decision. It adds latency per hop (usually small, but measurable at the p99), memory per pod, a control plane that must itself be highly available and secured, and a learning curve for every team. It also only protects traffic that goes through it: a job that connects directly to the database bypasses it. Small systems with a handful of services often do better with mTLS provided by a library or the platform's load balancers. The deciding questions are how many services and languages you have and whether a platform team exists to run it (see Section 10.7).

Automating certificates

Certificates expire, and expired certificates are one of the most common causes of avoidable outages. On 6 December 2018 an expired certificate in Ericsson software used by the UK mobile network O2 left about 32 million customers without data service for most of a day. The cure is automation. The ACME protocol (Automatic Certificate Management Environment, RFC 8555, 2019), developed for the Let's Encrypt certificate authority, lets software prove control of a domain (by serving a token over HTTP, adding a DNS record or answering a TLS challenge), obtain a certificate and renew it without a person involved. Let's Encrypt, a free, non-profit CA

that began issuing certificates in 2015, made short-lived, automatically renewed certificates the norm; tools such as cert-manager do the same inside Kubernetes, and every major cloud load balancer can manage certificates for you.

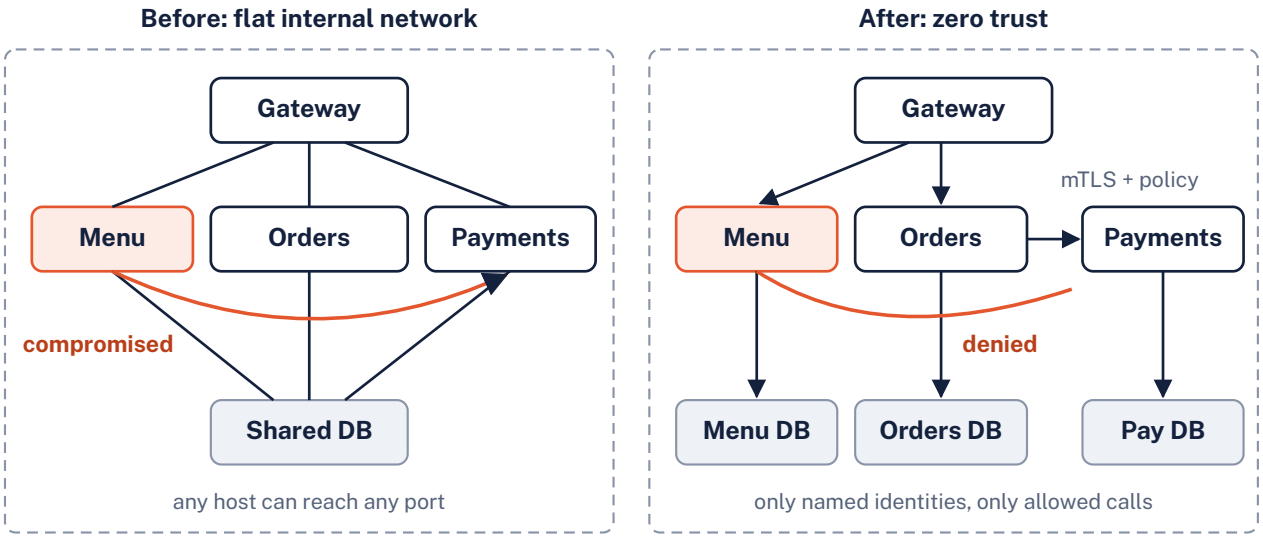
Automation is no longer optional. Under CA/Browser Forum ballot SC-081, the maximum lifetime of publicly trusted TLS certificates fell to 200 days for certificates issued from 15 March 2026, and falls to 100 days from 15 March 2027 and 47 days from 15 March 2029. Short lifetimes limit the damage of a stolen key and reduce reliance on revocation, which has always worked poorly. Architecturally this means every certificate, public or internal, needs an owner, automated renewal, monitoring of expiry dates with alerts well before the deadline, and a rehearsed manual fallback.

Segmentation and microsegmentation

Zero trust does not mean the network no longer matters; it means the network is no longer the only control. Network segmentation divides infrastructure into zones with controlled paths between them: public subnets for load balancers, private subnets for services, isolated subnets for databases, and separate cloud accounts for production, staging and security tooling. Microsegmentation shrinks the zones to single workloads, so each service may receive traffic only from the callers it needs and may send traffic only to its declared dependencies.

In Kubernetes, all pods can reach all other pods unless NetworkPolicy resources restrict them (and such policies take effect only if the cluster's network plugin enforces them), so the first step is a default-deny policy per namespace followed by explicit allows. Egress (outbound) control is as important as ingress: a compromised service that cannot open connections to the internet cannot easily send stolen data out or download further tools. Cloud security groups, Kubernetes network policies and mesh authorisation policies work at different layers (addresses and ports, pods and labels, workload identities and HTTP methods) and are best used together: the outer layers give coarse protection that survives a mesh misconfiguration, the inner layer expresses intent precisely. This is defence in depth (Section 8.1) applied to the network.

From a flat network to identity-based segmentation



Zero trust does not stop a service being compromised; it limits what the compromised service can reach.

Securing the API gateway

The API gateway is the main enforcement point between the internet and the services, so it carries several duties. **Authentication:** terminate TLS, then validate every access token (signature with a pinned algorithm, issuer, audience, expiry and not-before time), reject requests without one, and pass a verified identity, never the raw client claims, to the services. **Rate limiting:** limit requests per client, per user and per route, usually with a token bucket, answering HTTP 429 with a `Retry-After` header; separate, stricter limits protect login, password reset and voucher redemption from credential stuffing and enumeration. **Input validation:** enforce maximum request sizes and header counts, accept only the declared content types, and validate bodies against the OpenAPI schema so malformed requests never reach a service. A web application firewall can add generic attack signatures, but it is a supplement, not a substitute for correct code.

Know what a gateway cannot do. It sees the route and the token, but not whether order 81723 belongs to the customer asking for it. Broken object level authorisation, where a user changes an identifier in the URL and reads someone else's data, has been first in the OWASP API Security Top 10 in both its 2019 and 2023 editions, and only the service that owns the data can prevent it. Services must therefore authorise every request themselves, using the verified identity the gateway forwards. When a service calls another on a user's behalf, it should pass a token that names both the user and the calling service, for example obtained through OAuth 2.0 token exchange (RFC 8693), rather than its own all-powerful credential; otherwise a bug in one service turns it into a confused deputy that can act for anyone.

WORKED EXAMPLE

Worked example: moving the food-delivery platform from a flat network to zero trust

The platform runs about twenty services in one Kubernetes cluster and one cloud account. Services call each other over plain HTTP, a shared database user is used by six services, staff reach the admin console over a VPN, and the threat model in Section 8.2 flagged lateral movement after any single compromise as the top risk. A big-bang switch would break traffic, so the migration is staged and each step is reversible.

1. Inventory and observe (weeks 1 to 3)	List every service, its owner and its callers. Install the mesh in permissive mode, which accepts both plain and mTLS traffic, and use its telemetry to build the real call graph. Surprise found: the analytics job calls the Payments database directly.
2. Give every workload an identity (weeks 3 to 5)	One Kubernetes service account per service; the mesh issues SPIFFE certificates valid for 24 hours, rotated automatically. The shared database user is replaced by one credential per service, issued by the secrets manager for an hour at a time.
3. Encrypt everything (weeks 5 to 8)	Switch namespaces one at a time to strict mTLS, starting with low-risk ones. Dashboards show any remaining plain-text callers; each is fixed before its namespace flips. Rollback is one configuration change per namespace.
4. Authorise every call (weeks 8 to 12)	Write allow-policies from the observed call graph: Orders may call Payments on `POST /authorisations`; Menu may call nothing in Payments. Run them in audit mode for two weeks, review the denials, then enforce. Add default-deny Kubernetes NetworkPolicies and egress rules as an outer layer.
5. Replace the VPN for staff (weeks 10 to 16)	Put the admin console and internal tools behind an identity-aware proxy that requires single sign-on, phishing-resistant MFA and a managed, patched device. Refunds above a threshold need a fresh MFA step. The VPN is switched off once nothing uses it.
6. Harden the edge (in parallel)	The gateway validates tokens, applies per-route rate limits and schema validation, and forwards a verified identity; each service still checks that the caller may see the specific order or account.
7. Measure and keep it true	Fitness functions fail the build if a new service has no authorisation policy, and alerts fire on denied calls and on certificates within seven days of expiry. Every new dependency is a reviewed policy change.

The order matters: observe before enforcing, identity before policy, audit mode before deny. Each step can be rolled back on its own, which keeps the risk of the migration lower than the risk it removes.

The result is not that nothing can be breached. A compromised Menu service can still misuse its own data and the calls it is allowed, but it can no longer reach payments or other services' databases, and its attempts are logged and alerted on.

WATCH OUT

Common misconceptions about zero trust

- **It is not a product.** NIST describes zero trust as principles and an architecture; vendors sell components (proxies, identity providers, meshes) that implement parts of it.
- **It does not remove the network controls.** Segmentation and firewalls stay as outer layers; they are no longer the only thing between an attacker and the data.
- **mTLS alone is not zero trust.** Encryption with identity but an allow-all policy still lets any compromised service call anything; per-call authorisation is the point.
- **It is not finished in one project.** CISA's maturity model exists because organisations move pillar by pillar over years.
- **It must not make security unusable.** Saltzer and Schroeder's psychological acceptability still applies: frequent prompts teach people to approve anything, so step-up checks should be reserved for risky actions.

Questions people ask

Do we still need a VPN?

For people reaching web applications, an identity-aware proxy usually replaces it. VPNs or private links remain reasonable for site-to-site connections and for protocols a proxy cannot handle, but they should not grant broad trust to whatever is on the other side.

Is a service mesh required for zero trust?

No. It is one convenient way to get mTLS, workload identity and per-call policy everywhere. Platform load balancers, libraries and cloud identity features can achieve the same for a small number of services.

Does zero trust make the system slower?

Each check adds a little latency, typically well under a millisecond for a local mesh proxy and a few milliseconds when a remote decision is needed. Caching decisions, pushing policy to the enforcement point and reusing connections keep it small; measure it at p99 on critical paths.

Browser security: the same-origin policy, CORS and CSRF

So far the callers have been services and apps that hold their own credentials. A browser is different: it runs code from many sites side by side and attaches a user's cookies automatically. The platform's customer website and restaurant portal therefore have a trust boundary inside the customer's browser, where no gateway can reach. The starting point is the **same-origin policy**. Two URLs share an origin only if scheme, host and port all match, and script from one origin is restricted in how it can touch resources from another. The restriction falls mainly on *reading*: cross-origin reads are typically blocked, while cross-origin writes such as form submissions are typically allowed.

Cross-Origin Resource Sharing (CORS) is the controlled relaxation of that rule. If the site at `app.example.com` calls `api.example.com`, those are different origins, so the API must reply with `Access-Control-Allow-Origin` naming the site. For anything beyond a simple request (a `PUT` or `DELETE`, a custom header, a JSON body) the browser first asks permission with an `OPTIONS` preflight. Three points are often misunderstood. The browser enforces CORS, and the server only declares a policy, so tools such as `curl` ignore it and the API must still authenticate and authorise every call. CORS only ever loosens the same-origin policy; it adds no protection. And a simple request still reaches the server and may take effect: only the response is withheld. A wildcard origin is refused for requests with credentials, and echoing back whatever `Origin` arrives defeats the point, so compare it with an allowlist.

Cross-site request forgery (CSRF) exploits exactly that gap. A page on another site makes the browser send a request to the platform, the customer's cookie may go with it, and the server sees what looks like the customer changing a delivery address. OWASP's main defences are an unpredictable token checked on every state-changing request (per session where the server keeps state, a signed double-submit cookie where it does not), or a check of the `Sec-Fetch-Site` header with a fallback to verifying `Origin`. The `SameSite` cookie attribute from Section 8.3 is useful defence in depth but not a replacement: `Lax` still sends cookies on top-level cross-site `GET` navigations, so a `GET` must never change state. A token sent by script in an `Authorization` header is not attached automatically, so it is not exposed in this way, but a backend for frontend brings cookies back and needs CSRF protection. One warning: cross-site scripting defeats all of these defences, which leads to the next control.

Content Security Policy and security headers

Content Security Policy (CSP) is a response header that tells the browser what a page may load and run. Its main use is limiting the damage of cross-site scripting: with a policy in place, injected inline scripts, inline event handlers and `eval()` are blocked unless the policy explicitly allows them. The W3C specification is clear that CSP is best used as defence in

depth: it reduces the harm of an injection but does not replace input validation and output encoding. CSP Level 3 is still a working draft, so check the current version and browser support.

Allowlists of hosts are often bypassable and hard to maintain, so current guidance favours a **strict CSP** based on nonces or hashes, for example `script-src 'nonce-RANDOM'; object-src 'none'; base-uri 'none'`. A nonce must be a new unpredictable value on every response, which has an architectural consequence: pages have to be generated per response, whereas cached static pages suit hashes, which must be recalculated whenever a script changes. Every third-party script, such as a payment widget or an analytics tag, has to be inventoried and allowed. The safe rollout is `Content-Security-Policy-Report-Only`: the policy reports violations to an endpoint without blocking anything, and is enforced once the reports are clean. The directive `frame-ancestors 'none'` also stops other sites framing the checkout page, a defence against clickjacking.

A few other headers are cheap and worth setting everywhere. `Strict-Transport-Security` (HSTS, RFC 6797) tells the browser to use only HTTPS for the site; because the very first visit is unprotected, sites can also seek inclusion in the browsers' preload list. `X-Content-Type-Options: nosniff` stops the browser guessing content types. `Referrer-Policy: strict-origin-when-cross-origin` keeps paths and order references out of the referrer sent to other sites. `Permissions-Policy` switches off browser features a page does not need; the courier page may need geolocation, the menu page does not. OWASP advises against relying on the old `X-XSS-Protection` header and recommends setting it to `0`; check its cheat sheet for current values.

The architect's concern is where these live. If each team sets them by hand, one service will forget. Set them once at the edge or gateway, or in a shared template, and test the deployed responses in the pipeline as a fitness function (Section 11.3).

CHECK YOUR UNDERSTANDING

▼ Why does mutual TLS between services, with every service allowed to call every other, fall short of zero trust?

It provides encryption and identity, but no authorisation. A compromised service can still call anything; zero trust needs a per-call policy that allows only the calls each identity actually requires.

▼ What is the difference between the policy engine and the policy enforcement point?

The policy engine decides whether a request should be allowed, using policy and inputs such as identity and posture; the enforcement point sits in the traffic path and applies that decision by allowing or blocking the request.

▼ Why are short-lived certificates attractive, and what do they demand?

A stolen key is useful only until the certificate expires, which reduces reliance on revocation. They demand fully automated issuance and renewal, plus monitoring of expiry.

▼ A customer changes the order number in a URL and sees another customer's order. Which component should have prevented it?

The service that owns orders, by checking that the authenticated customer owns that specific order. The gateway authenticated the user correctly but cannot know who owns each object.

▼ Try it: A team says: "The API gateway validates every token, and our services sit on a private network, so services need no further checks." What could go wrong, and what would you add under zero-trust thinking?

One compromised service or stolen credential can move laterally and call everything, because location is being treated as trust. A valid token also does not show that this customer may see this order. Give each service a verified identity with mutual TLS, ideally short-lived and issued automatically, and allow only the calls it needs. Each service must make the object-level check on data it owns. The gateway remains useful for tokens, rate limits and input validation.

SOURCES

- NIST SP 800-207: Zero Trust Architecture (2020): https://csrc.nist.gov/pubs/sp/800/207/final
- Ward and Beyer: BeyondCorp, A New Approach to Enterprise Security (2014): https://research.google/pubs/beyondcorp-a-new-approach-to-enterprise-security/
- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: https://www.rfc-editor.org/rfc/rfc8446
- RFC 8555: Automatic Certificate Management Environment (ACME): https://www.rfc-editor.org/rfc/rfc8555
- SPIFFE: overview of the SPIFFE standards: https://spiffe.io/docs/latest/spiffe-about/overview/
- MDN: Same-origin policy: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

- MDN: Cross-Origin Resource Sharing (CORS): https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS
- OWASP Cross-Site Request Forgery Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
- MDN: Content Security Policy (CSP): https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP
- W3C: Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- OWASP HTTP Headers Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html
- RFC 6797: HTTP Strict Transport Security (HSTS): https://www.rfc-editor.org/rfc/rfc6797.html

Software supply-chain security

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 0.8 How software gets built
- 8.1 Security principles

Terms it uses from earlier: Dependency, CI/CD pipeline, Container image, Attack surface, Least privilege

THE IDEA IN BRIEF

A restaurant does not grow its own flour or make its own ovens; it trusts suppliers. If a supplier's flour is contaminated, every restaurant that bought it is affected, however clean its own kitchen is. Good restaurants keep a list of every ingredient and where it came from, buy from suppliers who seal and label their deliveries, and check the seals on arrival. Software is built the same way, from hundreds of components written by other people and assembled by build machines. Supply-chain security means knowing what is inside your software, where it came from, and that nobody tampered with it on the way.

WHAT YOU NEED TO KNOW

- Most code in a modern service comes from dependencies, build tools, CI actions and base images; every one of them, and everyone who can change them, is part of the attack surface.
- event-stream (2018) abused maintainer trust, SolarWinds (2020) a compromised build system, Log4Shell (2021) a vulnerable library buried deep in dependency trees, and xz utils (2024) a patient takeover of a release process.
- An SBOM in SPDX or CycloneDX format lists components with exact identifiers so that exposure to a new vulnerability can be found by query; US Executive Order 14028 (2021) and the EU Cyber Resilience Act made SBOMs mainstream.
- SLSA (version 1.2) defines Build levels L0 to L3, ending with hardened, isolated builds, and Source levels up to two-party review; provenance records who built an artefact from what.
- Sigstore's keyless signing ties a short-lived certificate to a CI identity and records signatures in a public transparency log; admission checks verify signatures and provenance before deployment.

- Lockfiles with hashes, pinning CI actions and images by digest, small automated updates with a short delay for new releases, prioritised scanning and ephemeral, hermetic builds reduce the risk in daily work.
- The CRA's reporting obligations apply from 11 September 2026 (24-hour early warning, 72-hour notification) and most other obligations from 11 December 2027.

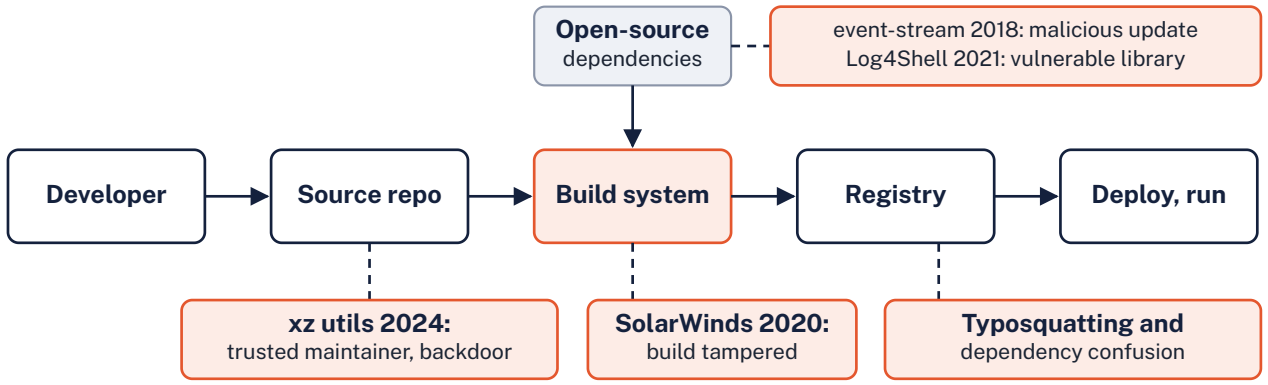
Why dependencies are an attack surface

Most of the code in a modern service was not written by the team that runs it. A small web service may declare twenty direct dependencies and, through their dependencies in turn, install several hundred packages; the container it ships in adds an operating-system userland, and the pipeline that builds it runs compilers, plug-ins and CI actions written by still more people. Section 0.3 introduced dependencies as a saving in effort with a maintenance cost. This section looks at their security cost: every one of those components, and every person and system able to change them, is part of your attack surface.

A software supply-chain attack compromises something you consume in order to reach you, usually many victims at once. It can arrive in four broad ways. **Malicious code from a trusted source:** a maintainer's account is phished, a maintainer turns hostile, or a project is handed to a stranger. **Malicious look-alikes:** typosquatting publishes `reqeusts` next to `requests`; *dependency confusion*, demonstrated by Alex Birsan in February 2021, publishes a package on the public registry with the same name as a company's internal package and a higher version number, which some build tools then prefer. **Tampering in the pipeline:** the source is fine but the build system, the signing keys or the registry are compromised. **Honest vulnerabilities:** no attacker was involved in writing the code, but a serious bug in a widely used transitive dependency exposes everyone who ships it.

Recent years have shown how fast the first kind spreads. In March 2025 the version tags of a popular GitHub Action, tj-actions/changed-files, were moved to a malicious commit that printed CI secrets into build logs, affecting thousands of repositories that referred to the action by tag. In September 2025 a maintainer of widely used npm packages such as `chalk` and `debug` was phished through a fake npm support site, and malicious versions were published and removed within hours; later the same month a self-propagating worm, nicknamed Shai-Hulud, stole npm tokens and used them to publish infected versions of further packages.

The supply chain and the trust relationships attackers abuse



Every arrow is a trust relationship; each famous incident abused a different one

Source, build, registry and dependencies are separate trust relationships, and each famous incident attacked a different one.

Four incidents, told accurately

event-stream (2018). event-stream was a popular npm package, downloaded about two million times a week, whose original author no longer had time for it. In September 2018 he handed publishing rights to a volunteer who had offered to help. The new maintainer added a dependency, `flatmap-stream`, whose published package contained an encrypted payload that activated only inside the build of Copay, a bitcoin wallet application, and tried to steal wallets with large balances. A developer noticed the suspicious code in November 2018. Nothing in npm's model had been broken: a trusted name had simply changed hands.

SolarWinds (2020). Attackers, later attributed by the US government to Russia's foreign intelligence service, gained access to the build environment of SolarWinds, the maker of the Orion network management product. An implant (named SUNSPOT by investigators) watched for Orion builds and swapped one source file for a backdoored version during compilation, then put the original back. The resulting updates, published between March and June 2020, were built by SolarWinds' own pipeline and signed with its genuine certificate. About 18,000 customers installed them; the attackers then chose a much smaller number of government agencies and companies for follow-on intrusion. FireEye discovered the campaign in December 2020 while investigating its own breach. The lesson for architects: a signature proves who built something, not that the build was clean.

Log4Shell (2021). CVE-2021-44228, disclosed on 9 December 2021 and scored 10.0 under CVSS, was a flaw in Log4j 2, the dominant Java logging library. Log4j resolved lookups written inside log messages, including JNDI lookups that could load a class from a remote server, so an attacker who could get a string such as a username or a request header logged could run code on the server. No attacker had touched the supply chain; the damage came from how widely,

and how deeply, the library was embedded. Organisations spent weeks finding where they used it, several follow-up releases were needed to close related flaws, and those with an accurate inventory answered the question in hours.

xz utils (2024). xz is a compression library used, through other libraries, by the OpenSSH server on several Linux distributions. Over about two years an account called Jia Tan contributed useful work and, helped by other accounts pressing the overworked original maintainer to accept help, became a co-maintainer. Versions 5.6.0 and 5.6.1, released in February and March 2024, contained a backdoor that would let the holder of a specific private key run commands through SSH. The malicious code was hidden in binary test files and activated by a modified build script that existed only in the release tarballs, not in the public repository. On 29 March 2024 Andres Freund, a developer at Microsoft, disclosed it after investigating why SSH logins on a test system used unusual CPU and took about half a second longer; it was tracked as CVE-2024-3094. Because it was caught early, it had reached mainly development and rolling-release distributions rather than stable ones.

What each incident teaches

Incident	What happened	Link attacked	Controls that would have helped
event-stream (npm), 2018	A new volunteer maintainer, handed the package by its tired original author, added a dependency (flatmap-stream) carrying an encrypted payload aimed at one bitcoin wallet application.	Maintainer trust and transitive dependencies	Lockfiles and a delay before adopting new versions; review of new transitive dependencies; funding and succession for critical packages.
SolarWinds Orion, 2020	Attackers inside the vendor's network implanted code that swapped a source file during the build, so legitimately signed updates contained the SUNBURST backdoor.	Build system	Isolated, ephemeral, monitored builds; provenance; reproducible builds to compare outputs; least privilege for the network management tool itself.
Log4Shell (Log4j 2), 2021	A long-standing feature that resolved JNDI lookups in logged text let any attacker who could get a string logged run code on the server.	A vulnerable, deeply nested dependency	An SBOM to find every affected system in minutes; egress filtering; fast, rehearsed patching.
xz utils, 2024	After about two years of patient contributions, a co-maintainer shipped release tarballs with a hidden backdoor that targeted OpenSSH on some Linux distributions.	Source and release process	Building from the reviewed repository, not from hand-made tarballs; reproducible builds; scrutiny of binary test files and build scripts.

Software bills of materials

A software bill of materials (SBOM) is a machine-readable list of the components in a piece of software, with versions, identifiers and the relationships between them. It does not make software safer by itself; it makes questions answerable. When the next Log4Shell is announced, a searchable store of SBOMs for every deployed artefact turns “are we affected, and where?” into a query.

Two formats dominate. SPDX, from the Linux Foundation, began as a licence-compliance format; version 2.2.1 is the international standard ISO/IEC 5962:2021, and SPDX 3.0 (2024) adds profiles for security, builds and AI. CycloneDX, an OWASP project designed for security use cases, is standardised by Ecma International as ECMA-424; version 1.7 was released in October 2025. Both can identify components with a package URL (purl), such as

`pkg:npm/express@4.21.2`, which is what vulnerability databases match against. The companion idea of VEX (Vulnerability Exploitability eXchange) lets a supplier state that a listed vulnerable component is not exploitable in its product, for example because the vulnerable function is never called, which cuts the noise for customers.

Policy pushed SBOMs into the mainstream. US Executive Order 14028, *Improving the Nation's Cybersecurity*, signed on 12 May 2021, set out software supply-chain measures for software sold to the federal government, including providing purchasers with an SBOM; the National Telecommunications and Information Administration then published the minimum elements of an SBOM (supplier, component name, version, other unique identifiers, dependency relationships, author of the SBOM data and a timestamp). US policy has since moved away from uniform mandates: in January 2026 the Office of Management and Budget's memorandum M-26-05 rescinded the earlier requirement for suppliers to attest to secure development practices, leaving agencies to set risk-based requirements and to ask for SBOMs where they choose. In the EU, the Cyber Resilience Act described below makes an SBOM part of the technical documentation for products with digital elements.

Generate SBOMs during the build, from what was actually resolved and packaged, rather than from the manifest a developer wrote; store them alongside the artefact, keyed by its digest; and feed them to a service that re-checks every stored SBOM when a new vulnerability is published.

```
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.7",
  "metadata": {"timestamp": "2026-10-20T09:14:02Z",
               "component": {"bom-ref": "orders-service", "type": "application",
                             "name": "orders-service", "version": "4.12.0"}},
  "components": [
    {"bom-ref": "log4j-core", "type": "library", "name": "log4j-core",
     "group": "org.apache.logging.log4j", "version": "2.24.3",
     "purl": "pkg:maven/org.apache.logging.log4j/log4j-core@2.24.3",
     "hashes": [{"alg": "SHA-256", "content": "9f2c..."}]},
    {"bom-ref": "jackson-databind", "type": "library", "name": "jackson-databind",
     "version": "2.18.2",
     "purl": "pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.18.2"}
  ],
  "dependencies": [
    {"ref": "orders-service", "dependsOn": ["log4j-core", "jackson-databind"]}
  ]
}
```

An abridged CycloneDX SBOM for the food-delivery orders service. The purl and hash identify each component exactly, so a scanner can match it against vulnerability databases.

SLSA: levels for build integrity

SLSA (Supply-chain Levels for Software Artifacts, pronounced *salsa*) is a framework, now maintained under the Open Source Security Foundation, that describes incremental levels of protection against tampering. Version 1.0 (2023) defined a Build track; version 1.2 restructured the specification into several tracks and added a Source track. The central artefact is provenance: a signed statement, usually in the in-toto attestation format, of which build platform produced an artefact, from which source revision, with which build definition and parameters. A consumer verifies provenance against expectations such as *built by our CI from the main branch of github.com/fooddelivery/orders*.

The Source track has four levels: version control (L1), continuous, immutable history with provenance for each revision (L2), continuously enforced technical controls such as protected branches (L3), and two-party review of every change to protected branches (L4). For most organisations the realistic target is Build L3 on a shared, well-run build platform plus Source L3 or L4 for important repositories; the levels say what a platform must guarantee, and a platform team can meet them once for every service.

SLSA Build track levels

Level	Name	What it guarantees
Build L0	No guarantees	No requirements: the absence of SLSA.
Build L1	Provenance exists	The build produces provenance describing how the artefact was built. It catches mistakes but is easy to forge.
Build L2	Hosted build platform	A hosted build service generates and signs the provenance, protecting against tampering after the build.
Build L3	Hardened builds	The build platform isolates builds from each other and keeps signing material out of reach of user-defined build steps, protecting against tampering during the build.

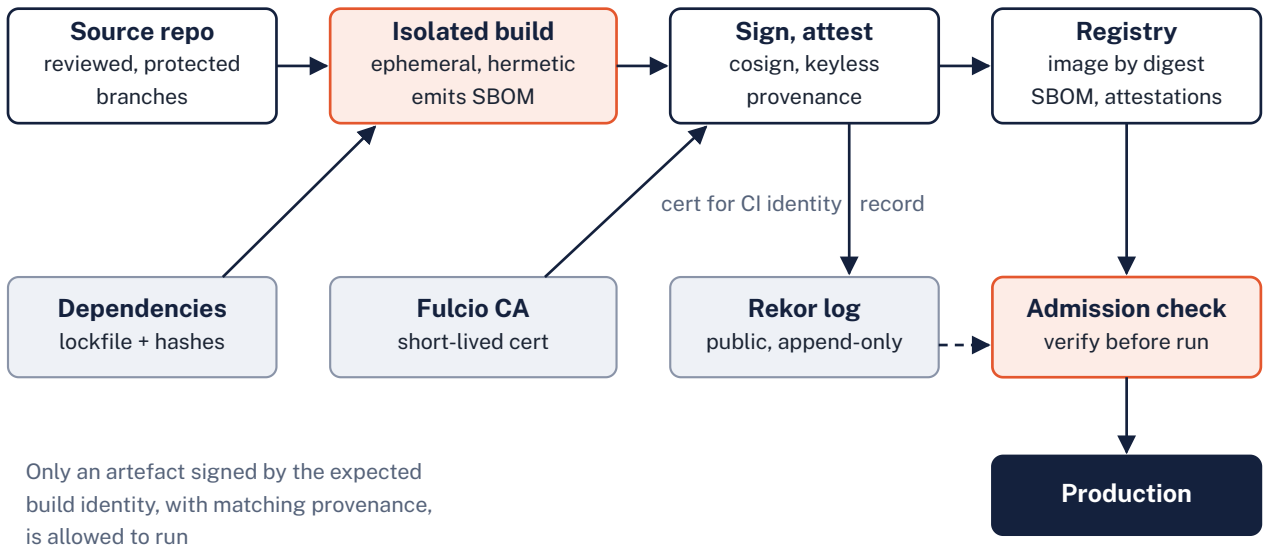
Signing and provenance with Sigstore

Signing artefacts has long been possible, but managing long-lived signing keys was so burdensome that few open-source projects did it, and stolen keys were themselves a risk. Sigstore, launched in 2021 and now part of the Open Source Security Foundation, makes signing cheap through *keyless* signing. A CI job proves its identity to Fulcio, Sigstore's certificate authority, with an OpenID Connect token; Fulcio issues a certificate valid for a few minutes that names that identity, such as a specific workflow in a specific repository; the tool

signs the artefact and records the signature in Rekor, a public append-only transparency log; the ephemeral private key is then discarded. Verifiers check the signature, the identity in the certificate and the log entry, instead of trusting a key that might have leaked.

The cosign tool applies this to container images and other artefacts, and package ecosystems have adopted the same ideas: npm can publish packages with provenance statements linking them to the source repository and build (2023), and PyPI supports trusted publishing from CI without long-lived tokens (2023) and attestations based on Sigstore (2024). At deployment, an admission controller in Kubernetes, such as Sigstore's policy controller or Kyverno, can refuse any image that is not signed by the expected identity or lacks the expected provenance. Remember SolarWinds: signing and provenance prove where something came from; only build isolation, review and detection make what came out trustworthy.

A hardened pipeline from commit to production



Each stage produces evidence the next can verify, so production runs only what the expected pipeline built from reviewed source.

Lockfiles, pinning and automated updates

A manifest such as `package.json` or `pyproject.toml` states acceptable version ranges; a lockfile (`package-lock.json`, `poetry.lock`, `uv.lock`, `go.sum`, `Cargo.lock`) records the exact versions resolved, usually with a cryptographic hash of each. Committing the lockfile and installing from it in CI (`npm ci`, `pip install --require-hashes`) means that every build uses the same reviewed bytes, and that a package republished with different content fails the hash check. Pinning applies to build tooling too: refer to CI actions by full commit hash, not by a tag that its owner, or an attacker, can move, which is exactly what made the tj-actions incident spread; refer to base images by digest.

Pinning without updating creates the opposite risk: old versions accumulate known vulnerabilities and upgrades grow too large to attempt. Update bots such as Dependabot and Renovate open small, frequent pull requests that CI tests automatically. A useful refinement is a minimum release age, waiting a few days before adopting a routine new version, because most malicious releases are detected and pulled within hours or days; security fixes are exempt and fast-tracked.

Software composition analysis (SCA) tools, such as OWASP Dependency-Check, Trivy, Grype or commercial services, match your dependencies against vulnerability databases. Their weakness is noise: hundreds of findings, most in code paths you never call. Prioritise with exploit data (is it in the CISA Known Exploited Vulnerabilities catalogue, what is its EPSS score?), with reachability analysis where your tools offer it, and with exposure (is the service internet-facing?). Section 8.8 covers vulnerability management as a process.

IF YOU ARE NEW TO THIS

Step by step: same code, different build

The step most people trip over here is that the same source code can produce different software on different days unless the exact dependencies are locked. Follow one library through two days of builds of the orders service.

- The orders service's manifest asks for a library at 'version 2.18 or any later 2.x'. On Monday the build fetches the newest match, 2.18.2, and the tests pass.
- On Tuesday version 2.18.3 appears, published by the maintainer or perhaps by an attacker who has phished the maintainer's account. Nobody on the team changes a line, yet Tuesday's build fetches 2.18.3, because it also matches the range.
- The same can happen in the library's own transitive dependencies, which the team never chose.
- A lockfile stops this. On Monday the package manager records the exact version of every package it resolved, transitive ones included, usually with a cryptographic hash: a fingerprint of the package's exact contents. The team commits the file, and CI installs only what it lists.
- If 2.18.2 is ever republished with different contents, the fingerprint no longer matches and the build fails. Build tools need the same care: a CI action named by a tag can be moved to other code, as in the tj-actions incident, while a full commit hash always names the same code.
- Locking is not the end. Old versions gather known vulnerabilities, so an update bot proposes small upgrades for CI to test. Waiting a few days before adopting routine releases lets most malicious ones be caught first; security fixes go through quickly.
- **The lesson:** decide deliberately when your dependencies change. Lock exact versions so that every build uses the same reviewed bytes, then update in small, tested steps.

Containers and build integrity

A container image carries an operating-system userland as well as your code, and every package in it is part of the supply chain. Start from minimal base images (distroless, Alpine, or slim distribution images), which contain fewer packages to be vulnerable; rebuild images regularly so that base-image fixes arrive even when your code has not changed; scan images in CI and again continuously in the registry, because new vulnerabilities are published against old images every day; and deploy images by digest so that what was scanned is what runs.

The build system is the most valuable target in the chain because it touches everything, as SolarWinds showed. Run builds on ephemeral runners that are created for one job and destroyed afterwards, so that an attacker cannot persist between builds. Make builds hermetic, with all inputs declared and fetched by hash and no network access during compilation, so that undeclared inputs cannot slip in. Keep long-lived secrets out of CI by using OIDC federation to obtain short-lived cloud credentials for each job, and give each pipeline the least privilege it needs, so that a pull request from a fork cannot reach the deployment credentials. Protect the release path with branch protection and two-person review. Finally, reproducible builds, where the same source always produces bit-for-bit identical output, let independent parties rebuild and compare; the Reproducible Builds project and distributions such as Debian have made most of their packages reproducible, and a mismatch is strong evidence of tampering.

The EU Cyber Resilience Act

The Cyber Resilience Act (CRA), Regulation (EU) 2024/2847, sets horizontal cybersecurity requirements for *products with digital elements* placed on the EU market: hardware and software products, including their remote data processing solutions. It entered into force on 10 December 2024 and applies in stages. The rules on notifying conformity assessment bodies apply from 11 June 2026. The reporting obligations in Article 14 apply from 11 September 2026: a manufacturer must report an actively exploited vulnerability or a severe incident affecting the security of its product with an early warning within 24 hours of becoming aware, a fuller notification within 72 hours and a final report later (for a vulnerability, within 14 days after a corrective measure is available; for a severe incident, within one month), through the single reporting platform run by ENISA, the EU cybersecurity agency. Most other obligations apply from 11 December 2027.

The essential requirements in Annex I are architectural as much as procedural: products must be designed with an appropriate level of security, delivered with a secure default configuration, limit attack surfaces, protect the confidentiality and integrity of data, minimise data, and be able to log security events and receive security updates. Manufacturers must identify and document components, including by drawing up an SBOM covering at least top-level dependencies, handle vulnerabilities through a coordinated disclosure policy, and provide security updates for a support period that should normally be at least five years unless the product is expected to be used for less. Fines for breaching the essential requirements or the manufacturers' obligations can reach EUR 15 million or 2.5% of worldwide annual turnover, whichever is higher.

Scope matters. Free and open-source software developed or supplied outside a commercial activity is outside the CRA, and a lighter regime applies to *open-source software stewards* that support such software for commercial use. Pure software as a service is generally outside the CRA unless it is the remote data processing solution of a product; cloud and other digital

service providers may instead fall under the NIS2 Directive. Whether a particular product is in scope, and which conformity route applies, is a legal question for each organisation; the architectural consequence is that SBOMs, vulnerability handling, update mechanisms and security logging become evidence that has to exist, not good intentions.

> **NOTE**
>
> ### Information, not legal advice
>
> The descriptions of the Executive Order, the OMB memorandum and the Cyber Resilience Act summarise how these measures affect software architecture. They are not legal advice; obligations depend on the product, the market and the role of the organisation, and should be confirmed with the official texts and qualified advisers.

Checklist: supply-chain security for a service team

Most of these are platform features once and habits afterwards.

- [ ] **Keep an inventory**
  Generate an SBOM for every build and store it with the artefact, so that "are we affected?" is a query, not a project.
- [ ] **Lock and verify dependencies**
  Commit lockfiles, install with hash checking in CI, and fail the build if the lockfile and manifest disagree.
- [ ] **Choose dependencies deliberately**
  Before adding one, check maintenance, number of maintainers, OpenSSF Scorecard results, licence and how much code it pulls in transitively.
- [ ] **Update continuously, but not instantly**
  Let a bot propose small updates every week; apply a short minimum release age for routine updates; fast-track security fixes.
- [ ] **Scan, then triage**
  Run software composition analysis and image scanning in CI and continuously in the registry; prioritise by exploitability and reachability, not by count.
- [ ] **Pin build tools as well as libraries**
  Pin CI actions and plug-ins to full commit hashes and base images to digests; review changes to them like code.
- [ ] **Isolate and harden the build**
  Ephemeral runners, no long-lived secrets (use OIDC federation), least-privilege tokens, protected branches and two-person review for release paths.
- [ ] **Sign and record provenance**
  Sign artefacts with Sigstore or an equivalent, publish SLSA provenance, and aim for Build L3 on the platform.
- [ ] **Verify at deployment**
  Admission policies reject images that are unsigned, signed by the wrong identity or lack provenance from the expected repository.
- [ ] **Plan for the next Log4Shell**
  Rehearse finding and patching one library across every service within a day, and know your CRA, contractual and customer reporting duties.

Questions people ask

Should we vendor (copy) all dependencies into our repository?
Vendoring gives control and offline builds but makes updates harder and hides upstream fixes. A lockfile with hashes plus a private registry mirror that caches approved packages gives most of the control with less maintenance.

Is a private registry enough to stop dependency confusion?
Only if builds are configured to resolve internal names exclusively from it. Use scoped or namespaced package names, register your internal names on public registries, and configure the package manager so that a public package can never override an internal one.

Who in the organisation owns supply-chain security?
Usually shared: a platform or security team provides the hardened pipeline, signing, scanning and policies once; each service team keeps its dependencies few, current and justified.

CHECK YOUR UNDERSTANDING

▼ **Why did SolarWinds customers' signature checks not protect them?**

The backdoor was inserted during SolarWinds' own build, so the trojanised update was genuinely signed by SolarWinds. A signature shows who produced an artefact, not that the build was clean.

▼ **What does a lockfile add to a manifest?**

The manifest states acceptable version ranges; the lockfile records the exact versions resolved and usually their hashes, so every build installs the same bytes and a changed package fails verification.

▼ **Why is referring to a CI action by tag riskier than by commit hash?**

A tag can be moved by whoever controls the repository, including an attacker, so the same reference can silently run different code; a full commit hash identifies immutable content.

▼ What question does an SBOM answer that a vulnerability scanner alone does not answer quickly across an estate?

Which deployed artefacts, in which services and environments, contain a given component and version, so that exposure to a newly announced vulnerability can be found everywhere at once.

▼ Try it: A CI pipeline uses build-action@main, a base image tagged latest, and installs packages with no lockfile. Nobody can say which services use a newly vulnerable library. Rank the fixes you would make, giving a reason for each.

First, produce an SBOM per build, so a new vulnerability can be found by query in minutes. Next, commit lockfiles with hashes and pin the CI action and base image by digest, as tags can move and anyone who can change them can reach your build. Then automate small dependency updates, with a short delay for new releases. Finally, sign artefacts with provenance and check them at deployment. Each step limits what a compromised upstream can do.

SOURCES

- SLSA specification v1.2: https://slsa.dev/spec/v1.2/
- Sigstore documentation: https://docs.sigstore.dev/
- Regulation (EU) 2024/2847 (Cyber Resilience Act), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2024/2847/oj
- NVD: CVE-2024-3094 (xz utils backdoor): https://nvd.nist.gov/vuln/detail/CVE-2024-3094
- CycloneDX specification: https://cyclonedx.org/

SECTION 8.7 OF 8.8 · INTERMEDIATE · 34 MIN READ · LAST REVIEWED NOVEMBER 2026

Privacy by design and the GDPR

BEFORE YOU READ THIS

This section builds on:

- 7.5 CQRS and event sourcing
- 8.2 Threat modelling
- 8.4 Protecting data

Terms it uses from earlier: Crypto-shredding, Data residency, Data-flow diagram (DFD), Time to live (TTL)

THE IDEA IN BRIEF

Imagine lending your diary to a friend. You expect them to read only the page you showed them, not to photocopy it, not to keep it for ever, and to give it back when you ask. Privacy by design builds those expectations into software from the start: collect only what is needed, use it only for the reason given, keep it only as long as necessary, and make it possible to show people what is held about them and to delete it on request. Laws such as the GDPR turn these expectations into duties, and many of them can only be met if the system was designed for them.

WHAT YOU NEED TO KNOW

- Privacy asks whether data should be collected and used at all, and what control people keep; a system can be secure and still violate privacy.
- The GDPR (Regulation (EU) 2016/679) and the UK GDPR with the Data Protection Act 2018, amended by the Data (Use and Access) Act 2025, share the Article 5 principles: lawfulness, fairness and transparency, purpose limitation, minimisation, accuracy, storage limitation, integrity and confidentiality, and accountability.
- Article 25 requires data protection by design (appropriate measures such as pseudonymisation) and by default (only necessary data processed, kept and made accessible), drawing on Ann Cavoukian's seven principles.
- Rights of access, rectification, erasure, restriction, portability and objection need a data map, per-service export and erasure handlers, restriction flags and suppression lists, with responses normally within one month.
- A DPIA is required before likely high-risk processing (Article 35) and works best alongside the threat model over the same data-flow diagram.

- Pseudonymised data remains personal data for whoever can re-identify it; anonymisation must withstand all means reasonably likely to be used, and is hard to achieve.
- Erasure must reach caches, search indexes, analytics, event logs (often by crypto-shredding), processors and backups (put beyond use until they expire); this section is information about how the law affects architecture, not legal advice.

NOTE

Information, not legal advice

This section explains how data protection law affects software architecture. It summarises the law accurately at the time of writing but is not legal advice: whether and how a rule applies depends on the facts, and organisations should take advice from qualified data protection lawyers and their data protection officer.

Privacy is a design property

Security and privacy overlap but are not the same. Security asks whether data is protected from people who should not have it. Privacy also asks whether the organisation itself should collect the data at all, what it may do with it, for how long, and what control the person it describes keeps. A system can be free of breaches and still invade privacy, for example by keeping every courier's location trail for ever or by quietly reusing order histories to train a pricing model.

The food-delivery platform is a good example of why privacy is architectural. It holds names, phone numbers and home addresses; order histories that can reveal religion or health through dietary choices; payment references; and a continuous stream of courier locations. These data flow from the apps through dozens of services, events, caches, logs, analytics stores and backups, and to processors such as the payment provider, the SMS gateway and the support-desk SaaS. Decisions about where personal data is stored, how it is keyed and how it can be found and removed are expensive to change later, which is exactly the test for architectural significance.

The law in brief: the GDPR and the UK GDPR

The General Data Protection Regulation, Regulation (EU) 2016/679, has applied since 25 May 2018. It protects personal data, which Article 4(1) defines as any information relating to an identified or identifiable natural person, directly or indirectly, for example by a name, an identification number, location data or an online identifier. The controller decides why and how data is processed and carries primary responsibility; a processor processes on the controller's

behalf under a contract (Article 28), as the platform's SMS gateway does. The GDPR applies to organisations established in the EU and, under Article 3, to others that offer goods or services to, or monitor the behaviour of, people in the EU.

In the United Kingdom the GDPR was retained after Brexit as the UK GDPR, which operates alongside the Data Protection Act 2018. The Data (Use and Access) Act 2025 amended both; most of its data protection changes came into force on 5 February 2026, including changes to lawful bases, automated decision-making and international transfers, and a requirement for controllers to handle data protection complaints follows from 19 June 2026. The European Commission renewed its adequacy decisions for the UK in December 2025, so personal data can continue to flow from the EU to the UK without extra safeguards. For architects, the two regimes share the same principles, rights and design duties; the differences are matters of detail for lawyers.

Maximum fines for the most serious infringements are EUR 20 million or 4% of worldwide annual turnover under the GDPR (GBP 17.5 million or 4% under the UK GDPR), whichever is higher, though the more common consequences are orders to stop processing, reputational damage and the cost of re-engineering. In November 2025 the European Commission proposed amendments to the GDPR as part of a *Digital Omnibus* package; at the time of writing these are still going through the legislative process and change nothing yet.

The Article 5 principles and what they mean for systems

Principle	In short	Architectural consequence
Lawfulness, fairness and transparency (5(1)(a))	Process only with a lawful basis, in ways people would reasonably expect, and tell them what you do.	Record a lawful basis per purpose; generate privacy notices from the same data map the systems use; no hidden tracking.
Purpose limitation (5(1)(b))	Collect for specified, explicit purposes and do not reuse data for incompatible ones.	Tag data with its purposes; check the tag before a new consumer (such as a machine-learning pipeline) is allowed to read it.
Data minimisation (5(1)(c))	Adequate, relevant and limited to what is necessary.	Do not collect date of birth if you only need an over-18 flag; store the courier's location trail only while it is needed.
Accuracy (5(1)(d))	Accurate and, where necessary, kept up to date.	One owner per fact; let people correct their data and propagate corrections to copies.
Storage limitation (5(1)(e))	Kept in identifiable form no longer than necessary.	A retention period for every data set, enforced automatically by TTLs, partition drops and expiry jobs, including in logs and backups.
Integrity and confidentiality (5(1)(f))	Appropriate security, including against unauthorised processing, loss and damage.	Everything in Sections 8.1 to 8.6: access control, encryption, key management, zero trust, supply-chain controls.
Accountability (5(2))	The controller is responsible for compliance and must be able to demonstrate it.	Records of processing, DPIAs, consent records, erasure ledgers and audit logs are evidence the architecture must produce.

Lawful bases, briefly

Article 6 permits processing only on one of six lawful bases: consent, performance of a contract with the person, a legal obligation, vital interests, a task in the public interest, or legitimate interests that are not overridden by the person's interests and rights. Special categories of data, such as health or religious belief, need an additional condition under Article 9. The basis is chosen per purpose, not per system, and it changes what the architecture must do. On the food-delivery platform, the delivery address is processed to perform the contract; invoices are kept to meet a legal obligation under tax law; fraud screening typically relies on legitimate interests; and marketing e-mails normally need consent, which electronic marketing rules also govern.

Consent has the strictest mechanics. Under Article 7 the controller must be able to demonstrate that consent was given, it must be specific to a purpose, and withdrawing it must be as easy as giving it. That means consent cannot be a single boolean column: the system needs an append-only record of what the person agreed to, when, through which channel and in response to which wording, and every consumer must honour a withdrawal promptly.

```
{
  "event": "ConsentGiven",
  "subjectRef": "cust_4411",
  "purpose": "marketing_email",
  "noticeVersion": "privacy-notice-2026-09",
  "wording": "Send me offers from restaurants near me",
  "channel": "ios-app 7.3.0",
  "method": "unticked checkbox, user ticked",
  "at": "2026-10-03T18:22:41Z"
}
{
  "event": "ConsentWithdrawn",
  "subjectRef": "cust_4411",
  "purpose": "marketing_email",
  "channel": "email unsubscribe link",
  "at": "2026-10-19T07:05:12Z"
}
```

Consent as an append-only record of events: the current state is derived from the latest event per purpose, and the history is the evidence Article 7 requires.

Data protection by design and by default

Article 25 turns privacy by design into a legal duty. Article 25(1), data protection by design, requires the controller, both when deciding how to process and during processing, to implement appropriate technical and organisational measures, such as pseudonymisation, that put the principles into effect, taking into account the state of the art, the cost, the nature and purposes of processing and the risks. Article 25(2), by default, requires that only the data necessary for each specific purpose is processed by default, covering the amount collected, the extent of processing, the storage period and accessibility; in particular, data must not be made accessible to an unlimited number of people without the person's intervention. The European Data Protection Board's Guidelines 4/2019 on Article 25 give practical examples.

The law borrowed the idea from Ann Cavoukian, Information and Privacy Commissioner of Ontario, who developed privacy by design in the 1990s and published its seven foundational principles in 2009; the international conference of data protection commissioners endorsed them in 2010. They remain a useful design review checklist.

Cavoukian's seven foundational principles of privacy by design

Principle	What it asks of a design
1. Proactive not reactive; preventive not remedial	Anticipate privacy harms at design time instead of fixing them after an incident.
2. Privacy as the default setting	The person has to do nothing to be protected; optional sharing is off until they turn it on.
3. Privacy embedded into design	Privacy is part of the architecture and business processes, not a bolt-on.
4. Full functionality: positive-sum, not zero-sum	Seek designs that achieve both privacy and the business goal instead of trading one away.
5. End-to-end security: full lifecycle protection	Protect data from collection to secure destruction.
6. Visibility and transparency	Practices are open and verifiable by users and independent reviewers.
7. Respect for user privacy	Keep it user-centric: strong defaults, clear notice and easy-to-use choices.

Rights that the architecture must support

Data subjects' rights are where privacy law most directly becomes system requirements. A controller must normally respond within one month of a request, extendable by up to two further months for complex or numerous requests (Article 12(3)), and must be able to find the person's data wherever it lives. In a system of twenty services, five SaaS processors and a data warehouse, that is impossible without a maintained data map: the record of processing activities that Article 30 already requires, kept close enough to the code to be accurate. Under the UK's 2025 amendments, a controller responding to an access request is required only to make reasonable and proportionate searches, which a good data map makes quick.

Data subjects' rights and what each requires of systems

Right	What the person can ask	What systems need
Access (Art. 15)	Confirmation that data is processed, a copy, and information such as purposes, recipients, retention and source.	A data map of every store holding personal data; a per-service export API keyed by a stable customer ID; identity verification; a way to assemble and deliver the result within the deadline.
Rectification (Art. 16)	Correction of inaccurate data and completion of incomplete data.	One authoritative owner per field; corrections propagated to copies and caches by events; recipients told (Art. 19).
Erasure (Art. 17)	Deletion where, among other grounds, the data is no longer needed or consent is withdrawn, subject to exceptions such as legal obligations.	An erasure orchestrator, per-store erasure rules (delete, anonymise, retain with reason), crypto-shredding for immutable stores, and a ledger proving completion.
Restriction (Art. 18)	Data kept but not otherwise used, for example while accuracy is disputed.	A restriction flag that every consumer checks, so data is stored but excluded from processing such as marketing or model training.
Portability (Art. 20)	Data the person provided, processed by automated means on the basis of consent or contract, in a structured, commonly used, machine-readable format.	An export in JSON or CSV with documented fields; optionally direct transmission to another controller where technically feasible.
Objection (Art. 21)	Stopping processing based on legitimate interests or public task unless there are compelling grounds; absolute for direct marketing.	Per-purpose switches; a suppression list (keeping the minimum needed, such as a hashed e-mail address) so the person is not re-added by a later import.
Automated decisions (Art. 22)	Safeguards against decisions based solely on automated processing with legal or similarly significant effects.	For example, automated deactivation of a courier: a route to human review, an explanation and a way to contest it.

Data protection impact assessments

Article 35 requires a data protection impact assessment (DPIA) before processing that is likely to result in a high risk to people's rights and freedoms, in particular systematic and extensive profiling with significant effects, large-scale processing of special-category or criminal-offence data, and systematic monitoring of a publicly accessible area on a large scale. Supervisory authorities publish lists of further cases; continuous tracking of couriers' locations

and algorithmic management of their work are typical candidates. A DPIA describes the processing and its purposes, assesses necessity and proportionality, assesses the risks to people, and records the measures that address them; if high risk remains, the controller must consult the supervisory authority before starting (Article 36).

Architects should treat a DPIA as the privacy counterpart of the threat model in Section 8.2, done at the same time and over the same data-flow diagram. The LINDDUN framework, developed at KU Leuven, does for privacy threats what STRIDE does for security threats, with categories such as linking, identifying, detecting, data disclosure and unawareness. The DPIA's conclusions become architectural requirements, such as *location trails are deleted 30 days after the delivery*, which can be tested.

International transfers and data residency

Chapter V of the GDPR restricts transfers of personal data to countries outside the European Economic Area. A transfer is lawful if the Commission has found the country adequate (Article 45), if appropriate safeguards such as the Commission's standard contractual clauses are in place (Article 46), or under narrow derogations (Article 49). After the Court of Justice invalidated the EU-US Privacy Shield in the Schrems II judgment of July 2020, transfers under contractual clauses need an assessment of the destination's laws; the EU-US Data Privacy Framework, found adequate in July 2023, covers transfers to US companies that certify under it, and survived a first challenge in the EU General Court in September 2025 (*Latombe v Commission*), although further litigation, including any appeal to the Court of Justice, could still change the position.

Two architectural points follow. First, a transfer is not only a database in another region: remote access by a support team abroad, a SaaS processor's sub-processors and a logging service hosted elsewhere all count, so the data map must record where data is stored and from where it is accessed. Second, data residency requirements, whether legal, contractual or chosen, are easiest to meet with regional deployment cells: each region holds its customers' personal data and keys, global services see only pseudonymous references, and replication across regions is explicit and reviewed. Retrofitting residency into a single global database is one of the most expensive changes a platform can face, which is why it belongs among the constraints captured early.

Compliance outside the EU and UK: SOC 2, ISO 27001, HIPAA and the CCPA

As the platform sells to large restaurant chains or enters new countries, other names appear. This passage is information, not legal advice: whether a law applies depends on the facts, and the rules change, so check the current position with qualified advisers. The names fall into two

groups: assurance schemes that customers ask for, and laws that bind you.

SOC 2 is an attestation report, not a certification or a law. Under American Institute of CPAs (AICPA) standards, an independent auditor examines a service organisation's controls against criteria in five categories: security, availability, processing integrity, confidentiality and privacy. Security is required in every report, the organisation adds the other categories that fit its commitments, and no fixed list of controls is prescribed. A **Type 1** report looks at the design of controls at a point in time; a **Type 2** report also tests that they operated effectively over a period, commonly six to twelve months, and is the more common of the two. **ISO/IEC 27001:2022** (amended in 2024 for climate action) is an international standard of requirements for an information security management system: a risk-driven way of running security, with 93 reference controls in Annex A. Unlike SOC 2, it leads to a certificate, which is only as useful as its stated scope.

HIPAA is a US federal law, and its Security Rule (45 CFR Parts 160 and 164) requires administrative, physical and technical safeguards for electronic protected health information. It binds covered entities (health plans, clearinghouses and many healthcare providers) and their business associates, not everyone who holds health data. The platform would ordinarily be outside it, but delivering prescriptions for pharmacies would be a reason to take advice on whether it had become a business associate. HHS proposed tougher rules at the end of 2024, including mandatory encryption and multi-factor authentication; check whether they have been finalised.

The **California Consumer Privacy Act** (CCPA), as amended by the California Privacy Rights Act, covers for-profit businesses that do business in California and meet a threshold: broadly, gross annual revenue above a figure set at $25 million and adjusted for inflation every odd-numbered year ($26,625,000 from January 2025; check the current figure), or buying, selling or sharing the personal information of 100,000 or more California residents or households, or earning half or more of revenue from selling or sharing it. Residents can know, delete and correct their data, opt out of its sale or sharing for cross-context advertising, and limit the use of sensitive information. Regulations in force from January 2026 add risk assessments and, in phases, cybersecurity audits. Other US states have their own privacy laws.

The architectural lesson is that the demands overlap: least privilege, encryption, logging, tested recovery and an inventory of data recur in each. Build one set of controls and map it to each scheme, rather than a separate programme for each, and shrink scope through data classification and segmentation, as Section 8.4 showed for PCI DSS.

Continuous compliance evidence

A Type 2 report asks whether a control worked all year, so screenshots gathered the week before the audit are weak and expensive. It is better to collect evidence as a by-product of normal work: pipeline logs showing every release passed review and scanning, infrastructure definitions, access reviews, the audit trail (Section 8.8). Each item is tagged with the controls it supports, and a control that fails raises an alert when it fails, not months later. Section 11.4 describes the policy-as-code side. NIST's Open Security Controls Assessment Language (OSCAL) offers machine-readable XML, JSON and YAML formats for control catalogues, implementation details and assessment results, replacing documents with structured data. Automation gathers evidence; it does not replace the auditor's judgement or the people accountable for the controls.

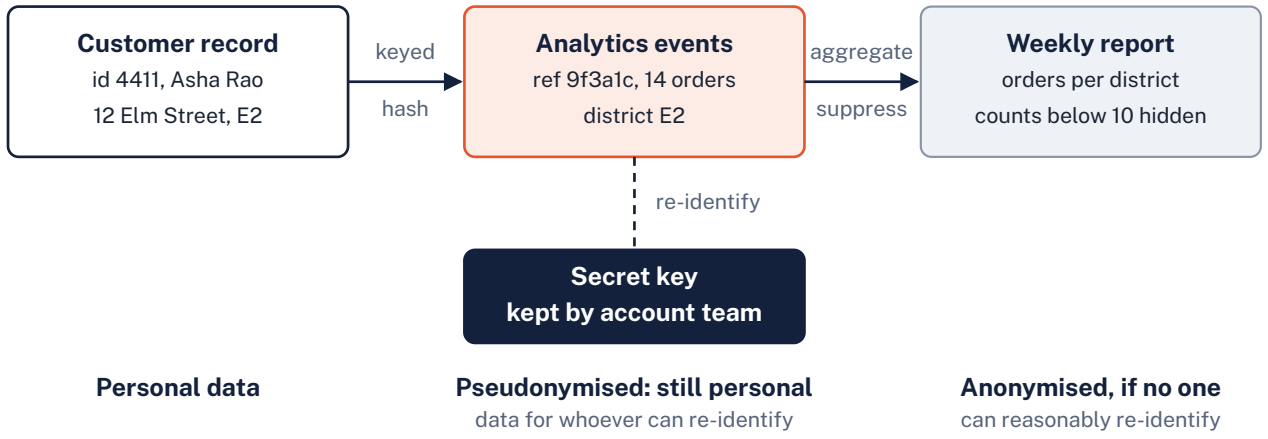
Minimisation, pseudonymisation and anonymisation

The cheapest personal data to protect is the data you never collect. Minimisation shows up in small decisions: an over-18 flag instead of a date of birth, a postcode district instead of a full address for analytics, a masked phone number relayed through the platform instead of revealing the customer's number to the courier, and location sampling only while a delivery is in progress.

Pseudonymisation (Article 4(5)) processes data so that it can no longer be attributed to a person without additional information, such as a key, that is kept separately and protected. A keyed hash (HMAC) of the customer ID, with the key held by the account team, lets the analytics team count orders per customer without seeing who the customers are. It is a valuable safeguard that Article 25 names explicitly, but Recital 26 is clear that pseudonymised data which could be attributed to a person by the use of additional information is still personal data. In *EDPS v SRB* (Case C-413/23 P, 4 September 2025) the Court of Justice added a nuance: whether data is personal is assessed from the perspective of whoever holds it, so pseudonymised data may not be personal data for a recipient that has no reasonable means of re-identifying anyone, although it remains personal data for the controller that holds the key, and the controller must still tell people about such recipients. The EDPB's Guidelines 01/2025 on pseudonymisation describe how to do it well.

Anonymisation goes further: the data no longer relates to an identifiable person, taking into account all the means reasonably likely to be used to identify someone, and the GDPR then no longer applies. True anonymisation is hard. Removing names is not enough when a combination of postcode, birth date and sex identifies most people, or when a single courier's location trail reveals where they live. Aggregation with small counts suppressed, generalisation and, for statistics, differential privacy are the usual techniques, and the result should be tested against realistic re-identification attempts.

Personal, pseudonymised and anonymised data



Pseudonymisation reduces risk but keeps the data personal for whoever can reverse it; anonymisation must withstand any reasonably likely means of re-identification.

IF YOU ARE NEW TO THIS

Step by step: hidden names are not anonymous

This section's hardest idea is the difference between pseudonymised and anonymised data. Both hide names, but, as the section explains, the GDPR treats them very differently. Follow one courier safety study.

- A courier record holds an ID, a name and a phone number: plainly personal data.
- For a road-safety study, the platform replaces the courier ID with a keyed hash: a reference code made with a secret key that the courier operations team keeps separately. Researchers see the code, a delivery count and the city zone, not names or phone numbers.
- Anyone holding the key can turn a courier ID into the same code and find that courier's rows. The data is pseudonymised: names are hidden, but people can still be picked out. As the platform holds the key, the section's rule applies: the data remains personal data for the platform, its researchers included.
- It is still worth doing: researchers can compare workloads without seeing who anyone is.
- Next, the platform builds a monthly report of deliveries per city zone, leaving out zones with fewer than ten couriers. If nobody could identify anyone from it by any means reasonably likely to be used, the report is anonymised, and, as the section explains, the GDPR then no longer applies to it.
- Removing names alone is often not enough. Postcode, birth date and sex together identify most people, and one courier's location trail can reveal their home.
- **The lesson:** a hidden name is not an anonymous person. Pseudonymisation reduces risk, but the data stays personal for whoever can re-identify people; call data anonymous only when no one could reasonably re-identify anyone from it, and test that against realistic attempts.

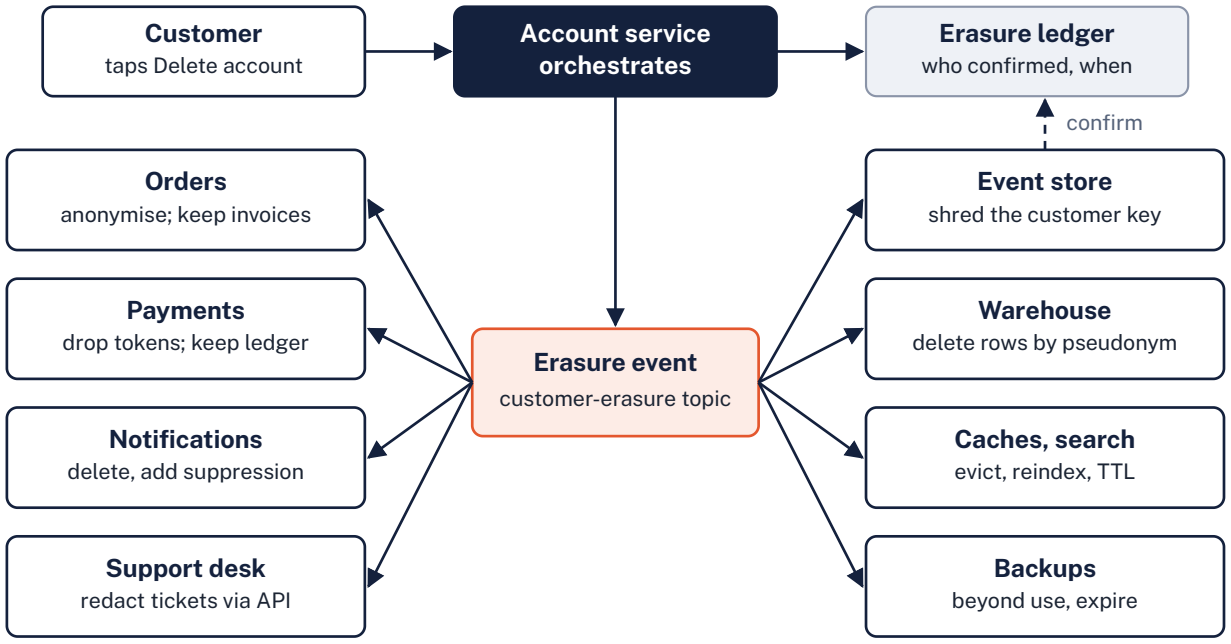
Retention and erasure across every copy

Storage limitation needs a retention schedule: for each category of data, how long it is kept, why, and what happens at the end (deletion, anonymisation or archiving). Schedules only work when systems enforce them automatically: TTLs on cache and key-value entries, time-partitioned tables whose old partitions are dropped, object-storage lifecycle rules, log retention settings and scheduled jobs whose success is monitored.

Erasure is hardest in the places personal data leaks into. **Caches and search indexes** need explicit eviction and re-indexing, with TTLs as a backstop. **Analytics** stores should hold pseudonymous references so that rows can be found and deleted, or the data should be

anonymised on the way in. **Event logs and event-sourced stores** are immutable by design; the usual technique is crypto-shredding, encrypting each person's fields with a per-person key and destroying the key, as described in Section 7.5, or keeping personal data out of events and storing only references. **Application logs** should not contain personal data that is not needed, which Section 8.8 returns to. **Backups** usually cannot be edited; the UK Information Commissioner's Office accepts that data may remain in backups until they are overwritten on an established schedule, provided it is put beyond use meanwhile and the person is told clearly what will happen, and if a backup is ever restored, erasures must be re-applied, which requires keeping the list of erased IDs. Finally, **processors** must be instructed to delete, since they act only on the controller's instructions (Article 28), and other **recipients** must be told (Article 19), so erasure must reach the SMS gateway, the support desk and the marketing platform through their APIs.

Erasure fan-out in the food-delivery platform



Every store that holds the customer's data must act, and say that it has

One request, many stores: each owner applies its own rule (delete, anonymise, shred or retain with a reason) and confirms completion.

WORKED EXAMPLE

Worked example: designing "delete my account" for the food-delivery platform

A customer in London asks to delete their account. The design must erase what can be erased, keep what the law requires with a stated reason, finish well within one month, survive failures part-way through, and produce evidence. Retention periods below are illustrative choices for this example, not legal requirements for any particular business.

1. Verify and confirm	The request arrives in the app from a logged-in session; the account service requires a fresh authentication, explains what will be deleted and what will be kept, and offers a data export first. A short cooling-off period (for example 14 days) allows the customer to cancel; outstanding orders and refunds must complete first.
2. Record the request	The account service writes an erasure ledger entry with a request ID, the customer ID and the list of stores expected to respond, taken from the data map. The ledger stores no more personal data than the ID needed to process the request.
3. Fan out	It publishes `CustomerErasureRequested` through the outbox to a topic. Each consumer is idempotent, because delivery is at least once, and handles the event even if the customer was already partly deleted.
4. Apply per-store rules	Orders replace name, phone and address with a tombstone but keep order lines and invoices for the tax retention period, with the legal obligation recorded as the reason. Payments ask the payment provider to delete stored card tokens and keep the transaction ledger. Notifications delete contact details and add a hashed e-mail to the suppression list. The event store destroys the customer's data key. The warehouse deletes rows by pseudonym in its next daily run. Caches are evicted and the search index updated.
5. Reach processors	Connectors call the deletion APIs of the support-desk SaaS, the SMS gateway and the marketing platform, recording each response.
6. Confirm and chase	Each consumer writes a confirmation to the ledger. A daily job raises an alert for any store that has not confirmed within seven days, well inside the one-month limit.
7. Handle backups	Database backups are kept for 35 days and are not used for anything except disaster recovery. The erased customer IDs are kept in a list so that a restore re-applies all erasures before the system is reopened.
8. Tell the customer	A final message confirms completion, says what has been kept, why and until when, and that backup copies will expire within 35 days.

The data map is the critical artefact: a store missing from it is a store that keeps the data. A fitness function can fail the build if a new service declares a personal-data field without registering an erasure handler.

Retention with a reason is not a failure of erasure. Article 17(3) lists exceptions, including compliance with a legal obligation; the design keeps the minimum data needed for that obligation and deletes it when the obligation ends.

Questions people ask

Is encrypting personal data enough to satisfy the GDPR?
No. Encryption is an important security measure, and encrypted data that leaks may not need to be reported to individuals, but the principles, lawful bases, rights and retention rules apply to encrypted data as to any other.

Can we keep everything in an anonymised form instead of deleting it?
Only if it is truly anonymous, meaning no one can reasonably re-identify people from it. Hashing an ID or dropping names usually produces pseudonymised data, which is still personal data for whoever holds the means to re-identify.

Who decides retention periods: architects or lawyers?
Lawyers and the business decide the periods and reasons; architects make sure every store has a period, that it is enforced automatically and that the system can prove it.

CHECK YOUR UNDERSTANDING

▼ Why is a single `marketing_consent` boolean column a weak design?

It cannot show when, how and to what wording consent was given, which Article 7 requires the controller to demonstrate, and it loses the history of withdrawals. An append-only record of consent events provides that evidence.

▼ The analytics team receives order data keyed by an HMAC of the customer ID. Is it personal data?

For the platform, which holds the key, yes: it is pseudonymised data. For a recipient with no reasonable means of re-identification it may not be, following EDPS v SRB, but the controller's obligations remain.

▼ **Why does a restored backup threaten erasures, and what prevents it?**

The restore brings back data that was erased after the backup was taken. Keeping a list of erased IDs and re-applying the erasures before the restored system is used prevents it.

▼ **What does Article 25(2) mean by data protection by default?**

Without any action by the person, only the data necessary for each purpose is collected, processed, stored and made accessible; for example, optional profile sharing is off by default.

▼ **Try it: Marketing wants to keep every courier's full GPS trail forever "for analytics", with couriers' names attached. Apply privacy by design: what would you challenge and what design would you propose? This is about architecture, not legal advice.**

Challenge the purpose first: what question needs this? Storage limitation and minimisation argue against forever. Keep precise trails only as long as dispatch and disputes need them, then delete them automatically. For analytics, use aggregated data such as speeds per zone, with names removed. Replacing names with IDs is only pseudonymisation and remains personal data. Restrict access by default, and run a DPIA alongside the threat model if the tracking is likely high risk.

SOURCES

- Regulation (EU) 2016/679 (GDPR), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2016/679/oj
- EDPB Guidelines 4/2019 on Article 25 Data Protection by Design and by Default: https://www.edpb.europa.eu/our-work-tools/our-documents/guidelines/guidelines-42019-article-25-data-protection-design-and_en
- ICO: Right to erasure: https://ico.org.uk/for-organisations/uk-gdpr-guidance-and-resources/individual-rights/individual-rights/right-to-erasure/
- SOC 2 services and report types (Schellman): https://www.schellman.com/soc-2
- An Introduction to SOC 2 (Accounting Today): https://www.accountingtoday.com/news/an-introduction-to-soc-2
- AICPA & CIMA: SOC suite of services: https://www.aicpa-cima.com/resources/landing/system-and-organization-controls-soc-suite-of-services
- ISO/IEC 27001:2022/Amd 1:2024: https://www.iso.org/standard/88435.html
- KPMG: ISO/IEC 27001:2022 overview: https://assets.kpmg.com/content/dam/kpmg/cn/pdf/en/2023/02/iso-iec-27001-2022.pdf
- HHS: Summary of the HIPAA Security Rule: https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html

- HHS: HIPAA Security Rule NPRM fact sheet: https://www.hhs.gov/hipaa/for-professionals/security/hipaa-security-rule-nprm/factsheet/index.html
- California Attorney General: CCPA: https://oag.ca.gov/privacy/ccpa
- CPPA: Updated monetary thresholds in CCPA: https://www.cppa.ca.gov/regulations/cpi_adjustment.html
- CPPA: California finalizes regulations (23 September 2025): https://www.cppa.ca.gov/announcements/2025/20250923.html
- NIST OSCAL: https://pages.nist.gov/OSCAL/

SECTION 8.8 OF 8.8 · INTERMEDIATE · 32 MIN READ · LAST REVIEWED NOVEMBER 2026

Security in operation: logging, detection and response

BEFORE YOU READ THIS

This section builds on:

- 2.7 Security, usability and more
- 8.2 Threat modelling
- 8.7 Privacy by design

Terms it uses from earlier: Personal data, Recovery time objective (RTO), Recovery point objective (RPO), Transactional outbox, Software bill of materials (SBOM)

THE IDEA IN BRIEF

A bank does not rely only on strong doors. It also has cameras that record who came in, alarms that ring when something unusual happens, a plan for what staff do during a robbery, and spare cash locked away elsewhere in case the main vault is lost. Running software securely works the same way. Systems keep careful records of important actions, watch those records for signs of attack, fix newly discovered weaknesses quickly, follow a rehearsed plan when something goes wrong, keep backups that attackers cannot reach, and tell regulators and customers in time when personal data is affected.

WHAT YOU NEED TO KNOW

- Assume breaches will happen: design to detect, contain and recover, with security scenarios that set measurable response times.
- Log security events (authentication, authorisation failures, admin and high-risk actions) with who, what, when and where; never log passwords, tokens, keys, card data or unnecessary personal data; protect logs with separate, write-once storage.
- An audit trail is a complete, immutable, attributable record of defined actions, emitted by the owning service when the action commits.
- A SIEM correlates telemetry and alerts on rules derived from the threat model and MITRE ATT&CK; rules should be versioned, tested and tuned to avoid alert fatigue.
- Prioritise vulnerabilities by known exploitation (KEV), likelihood (EPSS), reachability and exposure, not by CVSS base score alone, and agree remediation targets in advance.
- NIST SP 800-61 Revision 3 (April 2025) frames incident response through the CSF 2.0 functions; ransomware resilience depends on immutable, separately credentialed backups and tested restores.

> - Under GDPR Article 33, personal data breaches must be notified to the supervisory authority without undue delay and, where feasible, within 72 hours of awareness, and to individuals under Article 34 when the risk is high.

Security does not end at deployment

The earlier sections of this chapter aim to prevent attacks. No prevention is perfect, so mature organisations also design on the assumption that a breach will happen: they want to notice it quickly, limit its damage, recover cleanly and learn from it. Chapter 2 described the security tactics as resisting, detecting, reacting to and recovering from attacks, and showed how to write security scenarios with measurable responses such as *a leaked credential is detected and revoked within 15 minutes*. This section is about the architecture and practices that make such responses achievable.

Much of it builds on the operational foundations of observability and incident management in Chapter 9. The difference is the adversary: a failing disk does not hide its tracks, delete its logs or come back next week through a door it left open.

Security logging: what to log and what never to log

Security logs exist to answer, after the fact, who did what, to which resource, when, from where and with what result. The OWASP Logging Cheat Sheet is the standard practitioner reference. It recommends recording events such as authentication successes and failures, authorisation failures, session management failures, input validation failures, application errors, high-risk functions such as administrative actions and data exports, and the use of security controls themselves. Each event should carry when (a synchronised timestamp), where (application, service, host, environment), who (the authenticated identity, the source address) and what (the action, the object, the result and the reason).

Equally important is what must stay out. Logs are copied widely, kept for a long time and read by many people and tools, so they are a favourite place for secrets and personal data to leak from. Never log passwords, session identifiers or access tokens (log a hash or a key ID if you need to correlate), encryption keys and connection strings, payment card data or special-category personal data. Treat log data as personal data where it identifies people, and apply the minimisation and retention rules of Section 8.7. Redaction is best enforced centrally in the log pipeline as well as in code, because one forgotten debug statement can undo it.

Log these, never log those (after the OWASP Logging Cheat Sheet)

Record	Exclude, mask or hash
Authentication successes and failures, MFA challenges, password resets	Passwords, even wrong ones (a mistyped password is often a near-correct one)
Authorisation failures (access denied), especially repeated ones	Session identifiers, access tokens, refresh tokens, API keys (log a hash or the key ID instead)
Input validation failures and requests rejected by the gateway	Encryption keys, database connection strings and other secrets
Administrative actions: role changes, policy changes, user creation, configuration changes	Full payment card numbers, card verification codes, bank account details
High-risk business actions: refunds, payouts, data exports, account deletion	Health data, government identifiers and other special-category personal data
Security-relevant errors and exceptions; start-up and shutdown of security controls	Personal data beyond what investigation needs, and anything a user opted out of
Use of break-glass or emergency access	Source code, and data classified above what the logging system is cleared to hold

```
{
  "ts": "2026-10-21T19:42:07.318Z",
  "event": "authz.denied",
  "service": "orders",
  "env": "prod",
  "actor": {"type": "user", "id": "cust_4411", "session": "sha256:7c1e..."},
  "client": {"ip": "203.0.113.24", "ua_family": "iOS app 7.3"},
  "action": "GET /orders/81723",
  "resource_owner": "cust_9902",
  "reason": "not_owner",
  "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736"
}
```

A structured security event from the orders service. It shows who tried what and why it was refused, with a trace ID for correlation, but no token and no customer name.

Logs must also be protected from attackers, whose first step after gaining access is often to erase the evidence. Ship security logs promptly to a separate account or service that the application's own credentials cannot modify; store them with write-once retention; restrict and log access to them; and keep clocks synchronised so events from different systems line up. Treat log content as untrusted input: an attacker who can put a newline into a logged field can forge fake entries (log injection), so encode or reject control characters, and remember that Log4Shell was triggered by logging attacker-controlled text.

Audit trails

An audit trail is a narrower, stronger kind of log: a complete, chronological, tamper-evident record of defined actions, kept to establish accountability. Debug logs are sampled, rotated and changed whenever developers like; an audit trail must be complete for its scope (every refund, every role change, every access to a customer's support record), immutable, attributable to an authenticated identity and retained for a defined period. Regulations and standards often require one; PCI DSS, for example, requires audit log history to be retained for at least twelve months, with the most recent three months immediately available for analysis.

Architecturally, design the audit trail as a product of the domain, not a by-product of logging. Emit audit events from the service that owns the action, at the point the action commits (the transactional outbox guarantees both happen or neither does), into an append-only store with restricted write access. Tamper evidence can be added by hash-chaining, where each record includes a hash of the previous one, or by periodically anchoring a signed digest elsewhere. Event-sourced systems get much of this for free, as Section 7.5 showed. Cloud providers' control-plane audit logs, which record every API call made to the cloud account, are part of the same picture and must be enabled in every account and region.

Detection: SIEM, rules and response automation

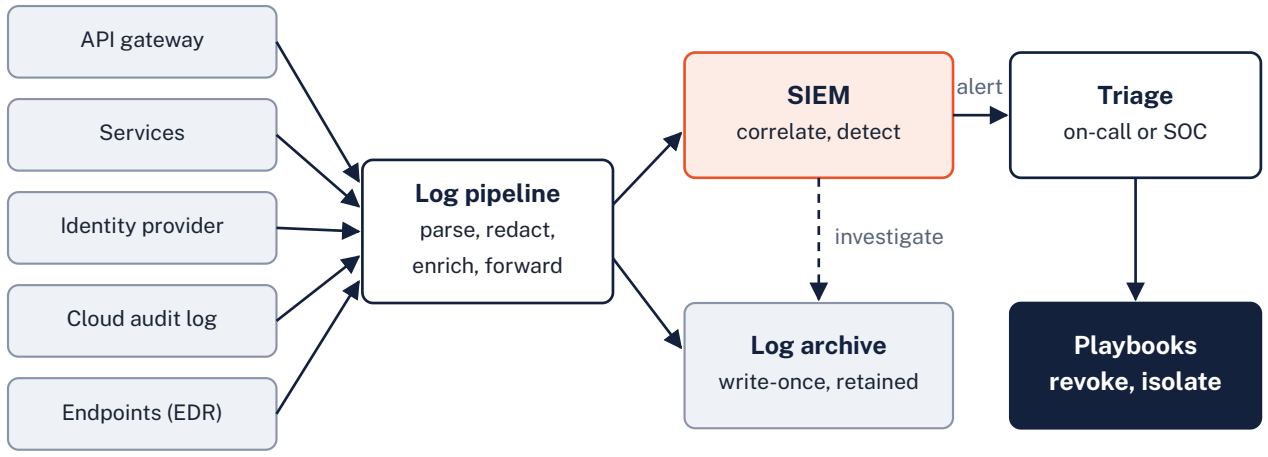
A security information and event management (SIEM) system collects security telemetry from across the estate, normalises it, correlates events and raises alerts when patterns match detection rules: many failed logins followed by a success, a service account used from a new country, a role granted administrator rights outside a change window, a large export at three in the morning. Typical sources are the API gateway, application security events, the identity provider, cloud audit logs, Kubernetes audit logs and endpoint detection and response (EDR) agents on staff laptops and servers.

Good detection is engineered, not bought. Derive rules from the threat model and from MITRE ATT&CK, a public knowledge base of real attackers' tactics and techniques maintained by MITRE since 2015; write them as code in a shared format such as Sigma, version them, and test them by simulating the attack they should catch. Measure alert quality: a rule that fires fifty

times a day and is always benign trains people to ignore it, and alert fatigue is how real attacks are missed. Cheap, high-signal tripwires help: honeytokens, such as a fake access key or a fake customer record that no legitimate process ever touches, produce almost no false alarms. Security orchestration, automation and response (SOAR) tools then run playbooks for well-understood alerts, such as disabling a user, revoking a key or isolating a container, so that people handle the judgement calls.

SIEM ingestion is often priced by volume, which tempts teams to drop logs. A tiered design helps: high-value security events go to the SIEM for real-time detection; bulky logs go to cheaper write-once storage where they can be searched during an investigation.

A security detection pipeline



Detection rules are code: versioned, tested against sample attacks and tuned so every alert is worth a human

Telemetry is collected and redacted once, then used both for real-time detection and, from a write-once archive, for investigation.

Vulnerability management

New vulnerabilities in the software you run are published every day. A CVE (Common Vulnerabilities and Exposures) identifier, such as CVE-2021-44228, gives each publicly known vulnerability a unique name; the programme, started by MITRE in 1999, assigns identifiers through hundreds of CVE Numbering Authorities, including major vendors and open-source foundations. The US National Vulnerability Database adds analysis such as affected product identifiers and scores, although a backlog in that enrichment from 2024, and a brief scare in April 2025 when the contract funding the CVE programme nearly lapsed, showed the risk of depending on a single feed; most organisations now combine several sources, including GitHub's advisory database and the open-source OSV database.

CVSS (the Common Vulnerability Scoring System, maintained by FIRST; version 4.0 was published in November 2023) rates severity from 0 to 10, with ratings of low (0.1 to 3.9), medium (4.0 to 6.9), high (7.0 to 8.9) and critical (9.0 to 10.0). The published base score describes the vulnerability in general; version 4.0's threat and environmental metrics adjust it for exploit activity and for your own environment. A base score is not a measure of your risk. Two signals improve prioritisation considerably: the Exploit Prediction Scoring System (EPSS, also from FIRST), which estimates the probability that a vulnerability will be exploited in the next 30 days, and the US Cybersecurity and Infrastructure Security Agency's Known Exploited Vulnerabilities (KEV) catalogue, which lists vulnerabilities with evidence of real exploitation. Combine them with reachability (is the vulnerable function used?) and exposure (is the service internet-facing?), and most of a scanner's hundreds of findings fall into the routine queue while a handful demand action today.

Agree remediation targets in advance, so that each finding does not become a negotiation, and measure the time to remediate. The figures below are typical illustrations, not a standard; regulated environments may impose their own, for example PCI DSS requires critical security patches to be installed within one month of release.

Example remediation targets

Finding	Target
Known exploited (for example in the CISA KEV catalogue), internet-facing	Mitigate within 24 to 72 hours; patch as soon as a fix exists
Critical severity, reachable, no known exploitation	Within 7 to 14 days
High severity	Within 30 days
Medium severity	Within 90 days, or the next routine update
Low severity, or not reachable	Routine updates; record the reason if deferred

DECISION GUIDE

Decision guide: triaging a newly published vulnerability

Run this when a scanner, advisory or news report names a vulnerability in something you might use.

Q1 **Does any deployed artefact contain the vulnerable component and version (check the SBOM store)?**

- **Yes** Continue to the next question, and record every affected service and owner.
- **No** Record "not affected" with the evidence; keep watching for new affected versions.

Q2 **Is it known to be exploited in the wild (for example listed in the CISA KEV catalogue) or is working exploit code public?**

- **Yes** Emergency path: mitigate within hours to days (patch, disable the feature, or block at the gateway or WAF), then patch properly.
- **No** Continue to the next question.

Q3 **Is the vulnerable code reachable in your use, and is the service internet-facing or handling sensitive data?**

- **Yes** Fix within the SLA for its severity, adjusted with the CVSS environmental metrics and EPSS.
- **No** Continue to the next question.

Q4 **Is there a compensating control that fully blocks exploitation (feature disabled, no network path)?**

- **Yes** Document it (a VEX statement suits this) and patch in the routine update cycle.
- **No** Patch in the routine cycle and revisit if exploitation is reported.

IF YOU ARE NEW TO THIS

Step by step: the highest score is not always first

The idea that needs the most care here is ranking vulnerabilities by your own risk rather than by score. Section 8.2 introduced CVSS: its base score describes a flaw in general, not your risk from it. Follow two findings that arrive on the same morning.

- Finding A is a critical flaw, base score 9.1, in a library bundled into the orders service. Finding B is a high one, base score 7.5, in software the API gateway runs. Sorted by score, A comes first.
- The first check is whether the component is actually deployed. The SBOM store shows that both are, and the team records every affected service and its owner.
- Next comes known exploitation. B is listed in the Known Exploited Vulnerabilities catalogue kept by CISA, a US government agency: there is evidence that attackers are using it. Nothing suggests that anyone is exploiting A.
- Then reachability and exposure. The gateway faces the internet, so attackers can reach B directly. A sits in a part of the library that the orders service never calls.
- So B takes the emergency path: mitigate within hours to days, for example by disabling the affected feature or blocking the attack at the edge, then patch properly. A is recorded as not reachable, with the evidence, and patched in the routine update cycle, to be revisited if exploitation is reported.
- For the many findings that are neither listed nor clearly unreachable, EPSS helps to rank them: it estimates how likely each is to be exploited in the next 30 days.
- **The lesson:** rank vulnerabilities by known exploitation, likelihood, reachability and exposure, not by base score alone, and agree remediation targets in advance so that each finding is not a fresh negotiation.

Penetration testing, bug bounties and disclosure

A penetration test is an authorised, time-boxed attempt by skilled testers to break into a system, ideally with knowledge of its architecture and with a scope agreed in writing. It finds classes of problems that scanners miss, such as broken object-level authorisation and abuse of business logic (redeeming the same voucher twice through a race condition). Commission one before a major launch, after significant architectural change and periodically; a red team exercise goes further, testing detection and response by simulating a real adversary over weeks.

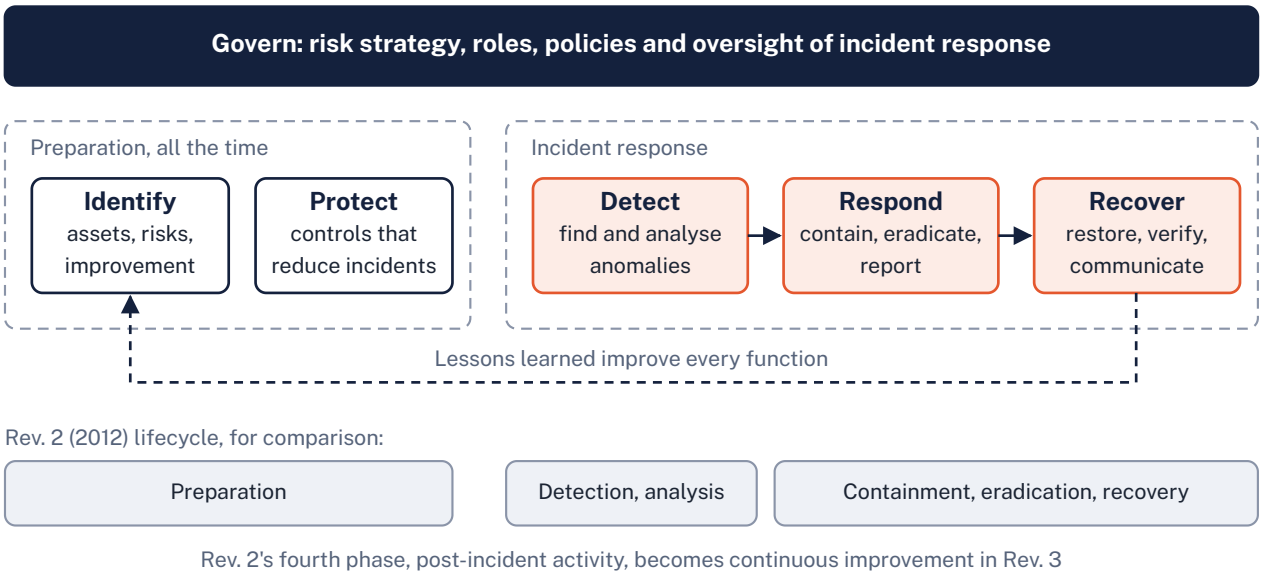
Outside researchers will find problems too, and they need a safe way to report them. A vulnerability disclosure policy states how to report, what testing is permitted and that good-faith research will not be met with legal threats; a `security.txt` file at `/.well-known/security.txt` (RFC 9116, 2022) tells researchers where to send reports. A bug bounty adds payment for valid findings, often through a platform that triages submissions. ISO/IEC 29147 (disclosure) and ISO/IEC 30111 (handling) describe the processes, and the Cyber Resilience Act requires manufacturers of products in its scope to have a coordinated vulnerability disclosure policy. A bounty is not a substitute for internal security work: launched on an untested system, it tends to produce a flood of reports the team cannot handle.

Incident response

Incident response is the organised handling of a security incident from detection to recovery and learning. For more than a decade the standard reference was NIST SP 800-61 Revision 2 (2012), with its four-phase lifecycle: preparation; detection and analysis; containment, eradication and recovery; and post-incident activity. Revision 3, published in April 2025 as *Incident Response Recommendations and Considerations for Cybersecurity Risk Management: A CSF 2.0 Community Profile*, replaces that lifecycle with the six functions of the NIST Cybersecurity Framework 2.0. Govern, Identify and Protect are continuous preparation that makes incidents less likely and response more effective; Detect, Respond and Recover are the response itself; and lessons learned feed improvement across all of them rather than being a final phase.

In practice a good response needs named roles (an incident commander who coordinates and decides, technical leads who investigate and act, a communications lead, a scribe who keeps the timeline, and quick access to legal counsel and the data protection officer), severity levels that set who is woken up, and an out-of-band channel, because an attacker in your e-mail or chat system can read the response. Containment is a trade-off: cutting an attacker off quickly limits damage but can destroy evidence and warn them, so preserve evidence (snapshots, logs, memory captures) before rebuilding, and be sure you have found all their footholds before you announce they are gone. Afterwards, a blameless review asks how the system and its processes allowed the incident, and produces architectural actions.

Incident response in NIST SP 800-61 Revision 3



Revision 3 maps incident response onto the CSF 2.0 functions instead of a four-phase lifecycle.

Ransomware resilience

Ransomware encrypts an organisation's systems and demands payment for the key; most groups now also steal data first and threaten to publish it. The architectural defences are those of this whole chapter (MFA, least privilege, segmentation, patching, detection), but recovery depends on one thing above all: backups the attacker cannot reach. Attackers with administrator rights look for backups first and delete or encrypt them.

Keep at least one copy of critical data immutable or offline: object storage with write-once retention locks that even an administrator cannot shorten, or media physically disconnected from the network. Put backups in a separate account or tenancy with separate credentials and MFA, so that compromising production identities does not reach them. The traditional 3-2-1 rule (three copies, on two kinds of media, one off-site) is a reasonable baseline. Above all, test restores regularly against the RTO and RPO the business has agreed, including the order of recovery: identity services and DNS come first, because nothing else can be rebuilt without them. In 2017 the shipping company Maersk rebuilt its global network after the NotPetya malware destroyed its Windows domain controllers; the recovery depended on the one domain controller that survived, in Ghana, because a power cut had taken it offline during the attack.

Personal data breaches and notification deadlines

A personal data breach is, under GDPR Article 4(12), a breach of security leading to the accidental or unlawful destruction, loss, alteration, unauthorised disclosure of, or access to, personal data. It is broader than theft: a ransomware attack that makes data unavailable, or an

e-mail sent to the wrong recipients, can qualify. Under Article 33 the controller must notify the supervisory authority without undue delay and, where feasible, within 72 hours of becoming aware of it, unless it is unlikely to result in a risk to people's rights and freedoms; later notifications must give reasons for the delay. Under Article 34 the controller must also tell affected people without undue delay if the breach is likely to result in a high risk to them. Article 33(5) requires every breach, notified or not, to be documented. The UK GDPR has the same rules, with notifications going to the Information Commissioner's Office.

Other regimes may run in parallel, with their own clocks, recipients and thresholds, and contracts often add duties to customers and partners. The table summarises the main EU and UK deadlines. The European Commission's November 2025 Digital Omnibus proposal would extend the GDPR deadline to 96 hours and limit authority notification to high-risk breaches, but at the time of writing it has not been adopted and the 72-hour rule applies.

The architectural lesson is that a deadline counted in hours can be met only if the systems can answer, quickly and with evidence, which data was affected, whose it was and whether it was actually accessed. That requires the data map of Section 8.7, data-access logs on stores of personal data, and encryption that may make a leaked copy unintelligible. As with the rest of this chapter, this is information about how the law affects systems, not legal advice; the decision to notify belongs to the controller, advised by its DPO and lawyers.

Notification clocks that commonly apply

Rule	Who tells whom	Deadline
GDPR / UK GDPR Art. 33	Controller to supervisory authority (in the UK, the ICO)	Without undue delay and, where feasible, within 72 hours of becoming aware, unless the breach is unlikely to result in a risk to people. Information may be provided in phases.
GDPR / UK GDPR Art. 33(2)	Processor to controller	Without undue delay after becoming aware.
GDPR / UK GDPR Art. 34	Controller to affected individuals	Without undue delay when the breach is likely to result in a high risk, unless, for example, the data was protected by encryption that makes it unintelligible.
NIS2 Directive (EU) 2022/2555	Essential and important entities to CSIRT or authority	Early warning within 24 hours of becoming aware of a significant incident, notification within 72 hours, final report within one month.
Cyber Resilience Act Art. 14	Manufacturers, via ENISA's single reporting platform	For actively exploited vulnerabilities and severe incidents: early warning within 24 hours, notification within 72 hours (applies from 11 September 2026).

Runbooks

A runbook is a written, tested procedure for a specific situation: revoke and rotate a leaked credential, isolate a compromised container, restore the orders database from an immutable backup, respond to a suspected personal data breach. A good runbook lists its trigger, the people to involve, the exact steps and commands, how to verify each step worked, how to roll back, and what evidence to keep. Runbooks are code in spirit: keep them in version control next to the systems they describe, automate the parts that are mechanical, and rehearse them in game days and tabletop exercises, because a procedure that has never been run usually fails at step three. Every detection rule should link to the runbook for its alert.

Worked example: the first 24 hours of a credential-leak incident at the food-delivery platform

Time	What happens	Why it matters
09:12	An engineer pushes a commit to a public repository that contains a long-lived cloud access key for the production deployment role.	The usual cause: a debugging script, not malice. Automated scanners watch public commits, so exposure must be assumed immediately.
09:14	The secret-scanning alert reaches the security on-call engineer; at 09:16 the SIEM shows API calls with the key from an unfamiliar address.	Detection in minutes depends on having secret scanning and cloud audit logs flowing into the SIEM before the incident.
09:20	Incident declared at the highest severity. An incident commander, a scribe and a technical lead are named; a dedicated channel is opened.	Clear roles stop five people from doing the same thing and nobody doing the urgent thing.
09:25	Contain: the key is deactivated, sessions issued from it are revoked, and a deny-all policy is attached to the role until it is rebuilt.	Deactivation, not deletion, keeps the key ID for log searches. Removing the commit is not containment: the key has already been copied.
09:45	Scope: the team queries the cloud audit log for every call made with the key ID, back to the moment it was created, not just since the push.	The key may have leaked earlier through another route; the log answers what was done, from where and when.
10:30	Findings: the attacker listed storage buckets, started 40 large compute instances in an unused region and created a new user with its own access key.	Attackers establish persistence first; cleaning up only the original key would leave them inside.
11:00	Eradicate: instances terminated, the new user and key deleted, recently changed policies reviewed and reverted; snapshots taken first for evidence.	Preserve evidence before destroying it; it is needed for the review, for insurers and possibly for regulators.
11:45	Data question: object-level access logs show successful downloads from the bucket holding customer exports. The DPO and legal counsel join.	Without object-level logs the team could not prove data was not read and would have to assume it was.
12:30	The DPO assesses a personal data breach likely to result in risk; the 72-hour clock is treated as running from the moment of awareness. A first notification is drafted.	Article 33 allows information in phases; late notification needs reasons. Whether individuals must be told depends on the level of risk.

Time	What happens	Why it matters
15:00	Recover: deployment role rebuilt with short-lived credentials via OIDC federation from CI; billing alarms and region restrictions added; the provider is contacted about fraudulent charges.	Fix the class of problem, not only the instance: no long-lived keys means nothing to leak.
18:00	Push protection for secrets enabled for every repository; all other secrets that were readable by the same engineer's machine rotated as a precaution.	Assume related credentials may also be exposed until proven otherwise.
Next day 09:00	Status update to leadership; notification to the supervisory authority submitted within the 72 hours with what is known; blameless review scheduled within five working days.	The review asks why a long-lived key existed and could be committed, not who committed it.

The timeline shows why operation is part of architecture. Every step that went well depended on a design decision made months earlier: secret scanning and audit logs feeding the SIEM, object-level access logs on the export bucket, a data map that told the DPO whose data was in it, and an identity design that let the deployment role be rebuilt without long-lived keys. Every step that went badly (a long-lived key existing at all, a bucket of customer exports that nobody needed any more) was also an architectural choice.

Checklist: is the system ready to be attacked?

- [ ] **Security events are logged, structured and centralised**
  Authentication, authorisation failures, admin and high-risk actions, with who, what, when, where and a trace ID; no secrets or unnecessary personal data.
- [ ] **Logs are protected and retained**
  Separate account or storage with write-once retention, restricted access, synchronised clocks and a defined retention period.
- [ ] **Cloud audit and data-access logs are enabled**
  Including object-level access for stores of personal or payment data, so that scope can be proven.
- [ ] **Detections exist for your top threats**
  Rules derived from the threat model, tested with simulated attacks, with an owner and a runbook for each alert.
- [ ] **Vulnerability SLAs are agreed and measured**
  Prioritised by exploitation, reachability and exposure; the SBOM store answers "where is it?" in minutes.
- [ ] **A disclosure route exists**
  A vulnerability disclosure policy and a security.txt file; penetration tests before major launches and after major change.
- [ ] **The incident response plan is rehearsed**
  Roles, severity levels, an out-of-band communication channel and contacts for legal, DPO, insurers and providers; exercises at least yearly.
- [ ] **Backups survive an attacker with admin rights**
  Immutable or offline copies in a separate account with separate credentials; restores tested against RTO and RPO.
- [ ] **Notification duties are mapped**
  Which regulators, customers and partners must be told, by whom and within which deadlines, with templates prepared.

CHECK YOUR UNDERSTANDING

▼ Why should a log record a hash or key ID instead of the access token itself?

Logs are widely copied and read, so a logged token can be stolen and replayed. A hash or key ID still lets investigators correlate events without exposing a usable credential.

▼ A scanner reports a CVSS 9.8 vulnerability in a library whose vulnerable function you never call, and a CVSS 7.5 one listed in the KEV catalogue in your internet-facing gateway. Which first?

The KEV-listed gateway vulnerability: it is known to be exploited and exposed. The unreachable 9.8 can follow routine patching, with the reasoning recorded.

▼ Why must backups live in a separate account with separate credentials?

Attackers who gain administrator rights in production look for backups and destroy them; separation means compromising production identities does not reach the recovery copy.

▼ When does the GDPR's 72-hour clock start?

When the controller becomes aware of the personal data breach, meaning it has a reasonable degree of certainty that a security incident has compromised personal data, not when the investigation is complete.

▼ Try it: Vulnerability A has CVSS 9.8, in a library on an internal batch server, and its vulnerable function is never called. Vulnerability B has CVSS 7.5, on your public login service, and is known to be actively exploited. Which first?

B first. CVSS base score alone is a poor guide. B is reachable from the internet and on the known-exploited list, so an attack is likely and could hit a critical service. Patch or shield it now, and check logs for signs of past exploitation. A's score is higher, but it is internal and its code is not reachable, so schedule it within the agreed target. Record the reasoning, then re-check if exposure changes.

SOURCES

- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- NIST SP 800-61 Rev. 3: Incident Response Recommendations and Considerations (2025): https://csrc.nist.gov/pubs/sp/800/61/r3/final
- FIRST: Common Vulnerability Scoring System v4.0: https://www.first.org/cvss/v4-0/
- FIRST: Exploit Prediction Scoring System (EPSS): https://www.first.org/epss/

- RFC 9116: A File Format to Aid in Security Vulnerability Disclosure: https://www.rfc-editor.org/rfc/rfc9116

Further reading

- *Threat Modeling: Designing for Security*, Adam Shostack (2014), Wiley. The standard practical book on threat modelling: the four questions, data-flow diagrams, STRIDE, attack trees and how to make threat modelling part of everyday engineering.
- *Security Engineering: A Guide to Building Dependable Distributed Systems (3rd edition)*, Ross Anderson (2020), Wiley. A broad, deeply researched account of how real systems fail, from protocols and access control to banking, payments and the economics of security.
- *Designing Secure Software: A Guide for Developers*, Loren Kohnfelder (2021), No Starch Press. Security-by-design principles, threat modelling and secure design reviews from a co-inventor of STRIDE, written for working developers and architects.
- *API Security in Action*, Neil Madden (2020), Manning. Authentication, tokens, OAuth 2.0, capability-based security and securing service-to-service calls, with working code.
- *OAuth 2 in Action*, Justin Richer and Antonio Sanso (2017), Manning. Builds clients, authorisation servers and resource servers step by step, which makes the flows and their attacks concrete.
- *The Protection of Information in Computer Systems*, Jerome H. Saltzer and Michael D. Schroeder (1975), Proceedings of the IEEE, vol. 63, no. 9. The origin of least privilege, fail-safe defaults, complete mediation and the other design principles still quoted today.
  https://doi.org/10.1109/PROC.1975.9939
- *SP 800-207: Zero Trust Architecture*, Scott Rose, Oliver Borchert, Stu Mitchell and Sean Connelly (NIST) (2020), National Institute of Standards and Technology. The reference definition of zero trust, its tenets and its logical components (policy engine, administrator and enforcement point).
  https://csrc.nist.gov/pubs/sp/800/207/final
- *Regulation (EU) 2016/679 (General Data Protection Regulation)*, European Parliament and Council of the European Union (2016), Official Journal of the European Union (EUR-Lex). The primary text behind the privacy section; Articles 5, 25, 32 to 35 matter most to architects.
  https://eur-lex.europa.eu/eli/reg/2016/679/oj

Glossary

Anonymisation
Processing data so that it no longer relates to an identifiable person by any means reasonably likely to be used; truly anonymous data falls outside the GDPR.

Asymmetric (public-key) cryptography
Cryptography with a key pair: a public key that anyone may hold and a private key kept secret; used for key exchange, digital signatures and certificates.

Attack surface
The sum of the points where an attacker can try to enter data into, extract data from or otherwise interact with a system: endpoints, ports, files, queues, admin tools and the people and code it trusts.

Attack tree
A tree whose root is an attacker's goal and whose children are the alternative (OR) or combined (AND) ways of reaching it, popularised by Bruce Schneier in 1999.

Attribute-based access control (ABAC)
Deciding access by evaluating a policy over attributes of the user, the resource, the action and the context, such as country, data classification or time of day.

Audit trail
A complete, chronological and tamper-evident record of defined actions, showing who did what and when, kept to establish accountability.

Authentication
Verifying that a user or system is who or what it claims to be, for example by a password, a passkey or a certificate.

Authorisation
Deciding whether an authenticated user or system is allowed to perform a particular action on a particular resource.

Bug bounty
A programme that pays outside researchers for reporting valid security vulnerabilities through an agreed process.

Certificate authority (CA)
An organisation or system that issues digital certificates by signing them, vouching that a public key belongs to a named server, person or workload.

Controller
The organisation that decides why and how personal data is processed and carries primary responsibility for complying with data protection law.

CVE
Common Vulnerabilities and Exposures: the programme that gives each publicly known vulnerability a unique identifier such as CVE-2021-44228.

CVSS
The Common Vulnerability Scoring System, maintained by FIRST, which rates the severity of a known vulnerability from 0 to 10; version 4.0 was published in 2023.

Cyber Resilience Act (CRA)
Regulation (EU) 2024/2847, which sets cybersecurity requirements, vulnerability handling and reporting duties for hardware and software products sold in the EU.

Data protection by design and by default
The GDPR Article 25 duty to build data protection principles into systems through appropriate measures, and to process only the data necessary for each purpose by default.

Data protection impact assessment (DPIA)
An assessment required by GDPR Article 35 before processing likely to result in a high risk to people, describing the processing, its necessity, the risks and the measures that address them.

Data-flow diagram (DFD)
A diagram of external entities, processes, data stores and the data flows between them, with trust boundaries drawn across the flows; the usual model for threat modelling.

Defence in depth
Layering several independent controls so that an attacker who defeats one still faces others, for example a gateway check, a service check and encryption of the data itself.

Envelope encryption
Encrypting data with a data key and then encrypting (wrapping) that data key with a key-encryption key held in a KMS, so bulk data never travels to the KMS and keys can be rotated cheaply.

EPSS
The Exploit Prediction Scoring System from FIRST, which estimates the probability that a vulnerability will be exploited in the next 30 days.

Hardware security module (HSM)
Tamper-resistant hardware that generates and uses keys inside itself and never releases them in plaintext; KMS services are usually built on HSMs.

Immutable backup
A backup copy that cannot be changed or deleted before its retention period ends, even by an administrator, protecting it from ransomware and malicious insiders.

Incident response
The organised handling of a security incident: detecting and analysing it, containing it, removing the cause, recovering and learning from it.

JSON Web Token (JWT)
A compact, signed (and optionally encrypted) token carrying claims such as issuer, subject, audience and expiry, defined in RFC 7519; widely used for access tokens and ID tokens.

Key management service (KMS)
A managed service that creates, stores, rotates and controls use of cryptographic keys, performing operations with them so that applications never see the master keys.

Lateral movement
An attacker's progress from the first machine or account they compromise to other systems inside the same network or organisation.

Lawful basis
One of the six grounds in GDPR Article 6 (consent, contract, legal obligation, vital interests, public task, legitimate interests) on which processing of personal data must rest.

Least privilege
The principle that every user, program and service gets only the permissions it needs for its job, for only as long as it needs them, so that a mistake or compromise does limited damage.

Lockfile
A file generated by a package manager that records the exact versions, and usually hashes, of every dependency resolved, so that every install uses the same packages.

Microsegmentation
Dividing a network into very small zones, down to single workloads, so that each one accepts traffic only from the callers it needs.

Multi-factor authentication (MFA)
Authentication that requires two or more different kinds of factor: something you know, something you have and something you are.

Mutual TLS (mTLS)
TLS in which both sides present and verify certificates, so that a service knows which service is calling it as well as the caller knowing the server.

OAuth 2.0
The delegated authorisation framework of RFC 6749, in which an authorisation server issues access tokens that let a client call a resource server on a resource owner's behalf, without seeing their password.

OpenID Connect (OIDC)
An identity layer on top of OAuth 2.0 that adds an ID token, a signed JWT telling the client who the user is and how and when they authenticated.

Passkey
A FIDO2/WebAuthn credential: a key pair created for one website, whose private key stays on the user's device or password manager, which makes login phishing-resistant and leaves no shared secret on the server.

Password hashing
Storing a password as the output of a deliberately slow, salted function such as Argon2id, scrypt or bcrypt, so that a stolen database cannot be cheaply turned back into passwords.

Perimeter security
The traditional model that protects a trusted internal network with firewalls and VPNs and treats anything already inside as friendly; sometimes called castle-and-moat security.

Personal data
Under the GDPR, any information relating to an identified or identifiable natural person (a living individual), such as a name, an ID number, location data or an online identifier.

Personal data breach
Under GDPR Article 4(12), a security breach leading to accidental or unlawful destruction, loss, alteration, unauthorised disclosure of, or access to, personal data.

PKCE
Proof Key for Code Exchange (RFC 7636): the client sends a hash of a one-time secret when it starts the authorisation code flow and the secret itself when it redeems the code, so a stolen code is useless.

Policy decision point (PDP)
The component that evaluates access policy and answers allow or deny; a policy enforcement point (PEP) asks it and then enforces the answer.

Policy enforcement point (PEP)
The component in the traffic path, such as a proxy, gateway or sidecar, that allows or blocks each request according to a policy decision.

Processor
An organisation that processes personal data on behalf of a controller, under its instructions and a contract, such as a cloud SMS or e-mail provider.

Provenance
Verifiable information about where an artefact came from: which build platform built it, from which source revision, with which inputs and parameters.

Pseudonymisation
Processing personal data so that it can no longer be linked to a person without additional information kept separately; under the GDPR, pseudonymised data is still personal data.

Public key infrastructure (PKI)
The certificate authorities, certificates, trust stores, policies and revocation mechanisms that together let parties verify each other's public keys.

Relationship-based access control (ReBAC)
Deciding access from relationships between users and objects (owner of, member of, works at), stored as a graph; Google's Zanzibar (2019) is the best-known example.

Reproducible build
A build that produces bit-for-bit identical output from the same source and inputs, so that anyone can rebuild and check that a published artefact was not tampered with.

Retention schedule
A documented list of how long each category of data is kept, why, and what happens to it at the end of that period.

Role-based access control (RBAC)
Granting permissions to roles and roles to users, so that access is managed by assigning people to roles such as support agent or restaurant manager.

Runbook
A written, tested procedure for handling a specific operational or security situation step by step, including how to verify and roll back each step.

Secrets management
Storing, distributing, rotating and auditing credentials such as passwords, API keys and private keys in a dedicated vault rather than in code or configuration files.

Secure by default
Shipping a product whose out-of-the-box configuration is the safe one, so that users must deliberately choose to weaken protection rather than remember to switch it on.

Secure Software Development Framework (SSDF)
NIST SP 800-218: a set of secure development practices grouped as prepare the organisation, protect the software, produce well-secured software and respond to vulnerabilities.

Shift left
Moving security (or testing) activities earlier in the development lifecycle, into requirements, design and coding, where problems are cheaper to fix than in production.

SIEM
Security information and event management: a system that collects security logs from many sources, correlates them and raises alerts when they match detection rules.

Sigstore
An open-source project for signing and verifying software artefacts with short-lived certificates tied to an identity and a public transparency log, avoiding long-lived signing keys.

SLSA
Supply-chain Levels for Software Artifacts: a framework of increasing levels of assurance that an artefact was built from the expected source by a trustworthy, tamper-resistant build.

Software bill of materials (SBOM)
A machine-readable inventory of the components and dependencies in a piece of software, in a standard format such as SPDX or CycloneDX.

Software composition analysis (SCA)
Tools and practices that identify the open-source components in software and match them against databases of known vulnerabilities and licences.

Software supply-chain attack
An attack that compromises something an organisation consumes, such as a dependency, build tool, update or registry, in order to reach that organisation and others downstream.

STRIDE
A mnemonic for six kinds of threat (spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege), created at Microsoft by Loren Kohnfelder and Praerit Garg in 1999.

Symmetric encryption
Encryption in which the same secret key encrypts and decrypts, such as AES; fast, so it is used for bulk data.

Threat modelling
A structured way of asking, about a design, what we are building, what can go wrong, what we will do about it and whether we did a good enough job.

Tokenisation
Replacing a sensitive value such as a card number with a meaningless token, while the real value is kept in a separate, tightly protected vault, often run by a payment provider.

Transitive dependency
A package your software uses indirectly because one of your dependencies, or one of theirs, depends on it.

Trust boundary
A line in a system where data or control passes between parts with different levels of trust or privilege, such as from the internet into an API, so input must be checked there.

Workload identity
A verifiable identity for a running service rather than a person, issued by the platform (for example a SPIFFE ID in a short-lived certificate), used instead of shared secrets.

Zero trust
A security model that grants no implicit trust based on network location: every request is authenticated, authorised and encrypted, per resource and per session (NIST SP 800-207).