Cryptography for Developers Course

Flavio Copes

Updated August 3, 2026

Contents

Use hashes, password hashing, authenticated encryption, signatures, TLS, secure randomness, and key management without inventing cryptographic protocols.

What you'll learn: Choose and review practical cryptographic operations, formats, libraries, keys, failure behavior, and migration for an application.

Cryptographic goals

Choose confidentiality, integrity, authenticity, and established high-level constructions.

Choose the security property

Separate confidentiality, integrity, authenticity, and non-repudiation before selecting a cryptographic operation.

Cryptography solves specific problems. Start by naming the property you need, because the property picks the tool for you.

Confidentiality hides content from anyone without the key. **Integrity** detects that data changed. **Authenticity** helps prove who created or approved a message. **Non-repudiation** goes one step further: the creator cannot later deny producing it, which is what digital signatures add over shared-secret authentication.

These sound similar. They are not, and mixing them up is where real systems break.

Why the property comes first

Here is a failure I want you to remember. A payroll file is encrypted before upload, but nobody verifies who created it. Its contents stay private while an attacker replaces it with different encrypted bytes. Confidentiality was achieved. Integrity and authenticity were never requested, so nobody got them.

Each property maps to a different operation:

```
confidentiality  -> encryption (with a key)
integrity        -> hash comparison against a trusted digest
authenticity     -> MAC (shared secret) or signature (private key)
non-repudiation  -> digital signature only
```

Notice that a MAC gives authenticity between two parties that share a secret, but not non-repudiation. Either party could have produced the MAC, so it proves nothing to a third party.

Some designs need several properties together. An encrypted session cookie usually needs confidentiality and authenticity at the same time, which is why authenticated encryption exists as a single construction.

What cryptography does not give you

Availability and authorization sit outside the cryptographic primitive. Encrypting a document does not decide who is allowed to request decryption, and a valid signature does not mean the signed action is permitted now. Name those needs separately and solve them with access control, not with more encryption.

My advice is to write the property down before touching any API. One line is enough: ”this backup needs confidentiality against whoever steals the disk, and integrity so we notice tampering on restore.”

Practice

Write the required properties for a private note, a software installer, a session cookie, and a payroll upload. For each answer, name the attacker and the evidence that the control produces. Then

change one case so confidentiality is unnecessary but authenticity remains essential, and explain the different choice. The installer is a good candidate: its contents are public, but you care deeply about who built it.

Separate encoding, hashing, and encryption

Distinguish reversible representation, one-way digests, keyed authentication, and encryption so familiar APIs are not used for the wrong job.

Base64 changes how bytes are written. It provides no secrecy. Anyone can reverse it without any key:

```
const encoded = Buffer.from('secret-key').toString('base64')
Buffer.from(encoded, 'base64').toString() //secret-key
```

Encoding is a reversible representation. Base64, hex, and URL encoding exist so bytes survive transport through systems that expect text. That is the whole job.

A real application stored an API key as Base64 and labeled the column `encrypted_key`. Anyone who read the database could decode the value without a secret. The column name promised a security property the code never provided.

The four operations, side by side

A **cryptographic hash** maps data to a fixed-size digest. It is one-way: you cannot recover the input, and you cannot practically find two inputs with the same digest.

```
import { createHash } from 'node:crypto'

createHash('sha256').update('hello').digest('hex')
// 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Run it twice with the same input and you get the same digest. Change one character and the whole digest changes.

A **MAC** authenticates data with a shared secret. Without the key, an attacker cannot produce a valid tag. This matters because a plain hash cannot authenticate a webhook: an attacker who changes the body can calculate a fresh hash for the new body. Only the keyed version stops that.

Encryption makes plaintext recoverable only with a key, and in any modern design it should include authentication too. It is the only operation on this list meant to be reversed, and only by a key holder.

How to pick

The goal chooses the primitive. Need to transport bytes as text? Encode. Need a fingerprint or a change detector? Hash. Need to prove a message came from someone holding a shared secret? MAC. Need to hide content and get it back later? Encrypt.

If you catch yourself hashing something you will need to read back, or encoding something you need to keep secret, stop. The operation does not match the property.

Practice

Classify Base64, SHA-256, HMAC, password hashing, and authenticated encryption by input, key use, reversibility, and security property. Save one small result from each available API. Then modify

or decode every result and record what the operation does not protect. The gaps you write down are the point of the exercise.

Threat-model encrypted data

Decide which attacker, endpoint, storage system, administrator, or physical loss encryption should protect against.

Encryption is not useful in the abstract. Ask two questions before anything else: who has the key, and who are you trying to keep out?

The answers decide the layer where encryption belongs. Each layer blocks a different attacker.

What each layer actually protects

Disk encryption protects a powered-off stolen disk. It does nothing against an attacker controlling the running server, because the operating system decrypts transparently for every process.

Database-level or application-level encryption can protect a database dump or a stolen backup. But the application holds the key, so the application can still decrypt records. Anyone who compromises the application inherits that ability.

TLS protects data moving between machines. It says nothing about what happens at either end.

Here is the failure pattern I see most often. A database field is encrypted, but the decryption key sits in the same application environment:

```
# .env on the same host that stores the data
DATABASE_URL=postgres://app@db.internal/prod
FIELD_ENCRYPTION_KEY=6f1d...c2a9
```

A stolen database backup is protected; remote code execution in the application is not. The attacker reads the key from the environment and decrypts everything, same as the app does.

Map plaintext and keys before choosing

Before picking a layer, trace one piece of data through the system and mark every place plaintext or key material exists:

```
customer address:
  browser input     -> plaintext (TLS in transit)
  app server memory -> plaintext
  request logs      -> plaintext?  <- often forgotten
  database column   -> ciphertext
  database backup   -> ciphertext
  encryption key    -> app environment variable
```

The logs line is where audits find surprises. Encrypting the column while logging the request body protects nothing.

Moving the key to a managed key service reduces direct key exposure, and it gives you an audit log of decrypt calls. But an authorized application may still request decryption whenever it wants. Encryption changes the attack path rather than removing it. That is not a weakness, it is the honest description of what you bought.

Practice

Draw the plaintext and key locations for one customer address from browser input through logs, database, backups, and application memory. Mark which attacker each encryption layer blocks and save the diagram. Then assume the application process is compromised and identify every remaining plaintext and decryption path. If the diagram shows the key next to the data, you have found your first fix.

Never invent cryptography

Use maintained high-level libraries and established constructions instead of designing algorithms, modes, padding, or message formats yourself.

Cryptographic code can look correct and fail catastrophically. It compiles, it round-trips your test data, and it is still broken in ways you cannot see from the output.

This is the one field where ”it works on my machine” means nothing. The attacks that matter exploit structure, timing, and error behavior, not bugs that throw exceptions.

What going wrong looks like

A developer encrypts with AES-CBC and adds a custom checksum. The design looks reasonable, but it accepts modified ciphertext before the checksum is checked and leaks useful error differences. That error difference is the basis of padding-oracle attacks, which recover plaintext without ever touching the key.

Another classic: picking ECB mode because it is the shortest name in the list.

```
// never do this
import { createCipheriv } from 'node:crypto'
const cipher = createCipheriv('aes-256-ecb', key, null)
```

ECB encrypts every 16-byte block independently. Identical plaintext blocks produce identical ciphertext blocks, so patterns in your data survive encryption and stay visible to anyone reading the ciphertext.

Neither mistake is a coding error. Both are design errors, made by choosing primitives instead of a construction.

What to use instead

Prefer APIs that choose secure algorithms, generate nonces, authenticate ciphertext, and define a versioned format. In practice that means an AEAD construction such as AES-GCM or ChaCha20-Poly1305 exposed through a maintained library, not a cipher mode you assembled yourself.

Follow the library's current documentation and run its published test vectors, so you know your integration produces the exact bytes the specification expects:

```
node --test crypto-wrapper.test.js
#   matches NIST GCM test vector (0.8ms)
```

Keep the cryptographic boundary small enough to review and replace. One module in your codebase should own encryption and decryption. Everything else calls it. When guidance changes, you have one file to update.

One more habit: a high-level construction may create a format migration later, so store an explicit

version with every ciphertext from the start. A single byte prefix is enough.

Practice

Inventory every direct cryptographic call in one project and map each call to its security goal. Replace or wrap one low-level composition with a maintained high-level API, then save passing published test vectors. Corrupt the ciphertext, nonce, and tag separately and prove every case fails without returning plaintext.

Check your understanding: cryptographic goals

Check confidentiality, integrity, authenticity, encoding, hashing, encryption, threat models, and trusted libraries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Hashes, passwords, and MACs

Use fingerprints, password hashing, salts, work factors, and keyed message authentication.

Use hashes for fingerprints

Use a modern cryptographic hash to identify bytes and detect accidental or adversarial changes when the expected digest is trusted.

A cryptographic hash turns any input into a fixed-size digest. A small input change produces a very different result, which makes the digest a reliable fingerprint for bytes.

SHA-256 is the standard choice today. It gives you a 32-byte digest, usually written as 64 hex characters:

```
shasum -a 256 app.zip
# 7c2df1a9e3b0f4d6...  app.zip
```

Hashes help identify files, content, and structured messages. Compute the digest on both ends of a transfer, compare, and you know the bytes match. Content-addressed systems like Git are built on exactly this idea.

Verifying a download

The common workflow ships a checksum file next to the artifact:

```
shasum -a 256 -c app.zip.sha256
# app.zip: OK
```

The `OK` means the file matches the digest in `app.zip.sha256`. Change one byte of the file and the same command reports `FAILED`.

The trust problem

Now the part everyone skips. A plain downloaded checksum does not authenticate a file if an attacker can replace both file and checksum.

A download page serves `app.zip` and its SHA-256 checksum from the same compromised server. The attacker replaces both, and the user sees a successful comparison. The math worked perfectly. The trust assumption was wrong.

The digest detects changes only when the expected value has an independent trusted path. That path can be a different server, a value you recorded earlier, or a digest published over a channel the attacker does not control. A signature can supply that path when its verification key is already trusted, which is why serious projects sign their release checksums.

So when you verify a fingerprint, ask where the expected digest came from. If the answer is "the same place as the file", you verified transfer integrity and nothing more. That is still useful against corruption. It is not useful against an attacker.

Inside your own code you can compute digests with `node:crypto`:

```
import { createHash } from 'node:crypto'
import { readFileSync } from 'node:fs'

createHash('sha256').update(readFileSync('app.zip')).digest('hex')
```

Practice

Hash a small file, save the digest through a separate trusted channel, and verify it from a clean copy. Change one byte and capture the failed check. Then replace both the file and its adjacent checksum and explain why the command succeeds but authenticity still fails.

Hash passwords slowly

Store password verifiers with Argon2id or another current password-hashing construction instead of fast hashes or reversible encryption.

Password storage has a special threat: an attacker can guess offline after stealing the database. No rate limit, no lockout, no logs. Just their hardware against your hashing choice.

That is why fast hashes fail here. An attacker steals a user table containing unsalted SHA-256 password hashes. They can test billions of common guesses offline without contacting the application or triggering a rate limit. SHA-256 was designed to be fast, and here speed works entirely for the attacker.

Encrypting passwords is also wrong: whoever holds the key can recover every password. You never need the original password back. You only need to check a guess against a verifier.

Use a password-hashing function

Use a maintained password library and current OWASP parameters. The function should be deliberately expensive in time and memory. **Argon2id** is the first recommendation today; **bcrypt** and **scrypt** remain acceptable. Memory hardness matters because it blunts GPU attacks, which is why Argon2id is preferred over bcrypt for new systems.

```
import argon2 from 'argon2'

const verifier = await argon2.hash('correct horse battery staple')
// $argon2id$v=19$m=65536,t=3,p=4$c29tZXNhbHQ$hashedbytes...
```

```
await argon2.verify(verifier, 'correct horse battery staple') // true
await argon2.verify(verifier, 'wrong guess')                  // false
```

Store the encoded algorithm, parameters, salt, and result, never the original password. The encoded string above already carries all of that, so verification years later still knows exactly what to recompute.

OWASP currently suggests Argon2id with at least 19 MiB of memory, an iteration count of 2, and one degree of parallelism as a baseline. Treat that as a floor, not a target.

Tune the cost on real hardware

Argon2id raises the time and memory cost for every guess. Raising it too far can exhaust login servers, so measure on production-like hardware and keep request-level abuse controls.

```
console.time('hash')
await argon2.hash('benchmark-password', { memoryCost: 65536, timeCost: 3 })
console.timeEnd('hash')
// hash: 92ms
```

Somewhere between 50ms and a few hundred milliseconds per hash is the usual budget. Remember that logins happen concurrently: 20 simultaneous logins at 64 MiB each is over a gigabyte of memory.

Practice

Benchmark a maintained Argon2id implementation with current OWASP guidance on production-like hardware and save latency plus memory settings. Verify a known password and reject a wrong one. Then run several concurrent hashes and show that the chosen cost remains within the service resource budget.

Understand salts and work factors

Use unique salts generated by the password library and tune cost parameters so identical passwords do not share a verifier and guesses remain expensive.

A salt is a random value mixed into each password hash. It is unique per password and does not need to be secret. Its job is to prevent precomputed and shared work.

Why that matters: two users choose the same password. With a shared or missing salt, their stored verifiers match and one cracked password reveals the other immediately. Worse, the attacker can crack the whole table at once, testing each guess against every row, or use precomputed tables built years before your breach.

With unique salts, every row becomes a separate cracking job. Same password, different verifier:

```
import argon2 from 'argon2'

await argon2.hash('hunter2')
// $argon2id$v=19$m=65536,t=3,p=4$Rk9Qb2ZYc1E$...
await argon2.hash('hunter2')
// $argon2id$v=19$m=65536,t=3,p=4$x1VtZ2JkQnc$...  <- different salt, different result
```

Let the library handle the salt

Let the library generate and store the salt inside its encoded result. Look at that encoded string: it carries the algorithm (`argon2id`), the version, the cost parameters (`m`, `t`, `p`), the salt, and the hash, separated by `$`. Verification reads everything it needs from the string itself.

This is why you should never build a `salt` column by hand or invent your own format. Hand-rolled salting is where bugs live: reused salts, truncated salts, salts applied to the wrong input.

Work factors age

The **work factor** is the cost knob: memory and iterations for Argon2id, rounds for bcrypt. Tune memory and time cost to your environment, then increase them as hardware changes. A cost that felt expensive five years ago is cheap on today's GPUs.

Unique salts stop shared precomputation but do not make weak passwords strong. A user whose password is `123456` falls in the first seconds of any attack regardless of parameters.

Old verifiers do not upgrade themselves. Rehash after a successful login when a stored verifier uses an older policy: at that moment you hold the plaintext legitimately, so you can write a new verifier at current cost and let the old one retire naturally.

```
if (await argon2.verify(stored, password) && argon2.needsRehash(stored, currentOptions)) {
  await saveVerifier(userId, await argon2.hash(password, currentOptions))
}
```

Practice

Hash the same password twice through the password library and save the two different encoded results. Verify both, then inspect where the algorithm, salt, and work factors are stored. Lower the policy cost for one test verifier and prove a successful login upgrades it without knowing or storing the plaintext later.

Authenticate messages with a MAC

Use HMAC or a high-level message-authentication API when two trusted parties share a secret and must detect forgery or modification.

A plain hash does not prove who created a message. Anyone can calculate another hash after modifying it.

A **message authentication code** fixes that by combining the message with a secret key. Without the key, an attacker cannot produce a valid tag for modified bytes. **HMAC** is the standard construction, and HMAC-SHA-256 is what most webhook signatures use.

```
import { createHmac } from 'node:crypto'

const secret = process.env.WEBHOOK_SECRET
const tag = createHmac('sha256', secret).update(rawBody).digest('hex')
// e3b1c44298fc1c14...
```

The sender computes the tag and ships it in a header. The receiver recomputes it over the received bytes and compares.

Verify the exact bytes

Verify the exact bytes you received. Here is the failure that bites almost every webhook integration: the sender signs the raw JSON body, but the receiver parses and serializes it before verification. Whitespace and key order change the bytes, so authentic requests fail. Capture the raw request body before any JSON middleware touches it, and feed that to the HMAC.

Compare in constant time

Never compare MACs with `==` or `===`. String comparison returns at the first differing character, so response timing leaks how many leading characters matched, and an attacker can build a valid tag byte by byte. Use a constant-time library comparison:

```
import { timingSafeEqual } from 'node:crypto'

const expected = createHmac('sha256', secret).update(rawBody).digest()
const received = Buffer.from(signatureHeader, 'hex')

const valid = expected.length === received.length &&
  timingSafeEqual(expected, received)
```

`timingSafeEqual` throws when lengths differ, so check the length first.

What a MAC does not do

An HMAC proves that a shared-secret holder created the exact bytes. It does not prevent replay: yesterday's validly signed request verifies again today. Include context such as version, timestamp, and message type in the authenticated data, and reject stale timestamps or repeated event identifiers.

It also gives no non-repudiation. Every verifier holding the secret can also forge messages, so a valid tag cannot prove to a third party which side produced it. Keep keys separate by purpose: the webhook secret authenticates webhooks and nothing else.

Practice

Create an HMAC for the exact raw bytes of a small webhook and save a successful verification. Change one byte and verify with the library constant-time comparison. Replay the original valid request and show why a timestamp or event identifier is still required.

Check your understanding: hashes, passwords, and macs

Check cryptographic hashes, password hashing, salts, work factors, HMAC, KDFs, comparisons, and upgrades.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Symmetric encryption

Use shared keys, authenticated encryption, nonces, associated data, and envelope encryption.

Understand shared-key encryption

Use one secret key for encryption and decryption and recognize that safely distributing and storing that key is the central problem.

Symmetric encryption uses the same secret key to protect and recover data. **AES** is the standard algorithm, and a 256-bit key is the common choice:

```
openssl rand -hex 32
# 9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
```

It is fast, hardware-accelerated on modern CPUs, and works well for stored data and large messages. When one system encrypts data for itself — database fields, backups, files at rest — symmetric encryption is the right tool.

The key is the whole problem

The algorithm is the easy part. Here is the hard part: anyone with the key can decrypt, and usually create valid ciphertext too. Symmetric encryption cannot distinguish between the parties that share a key.

That turns key distribution into the central design question. Every copy of the key is a place your data can leak from. Before shipping, answer three questions:

```
who can read the key?    (developers, CI, every replica, backups?)
where does it live?      (env var, file, secret store, KMS?)
how does it get there?   (deploy pipeline, manual paste, service call?)
```

If every application instance needs the key, each one becomes part of the threat model. A concrete failure: every application replica receives the same master key in an environment variable. One compromised debug endpoint can expose the key and every record protected by it. The encryption was fine. The distribution made it worthless.

Reduce the blast radius

Keep the key separate from the data it protects where possible. A key sitting in the same database as the ciphertext protects against nothing.

Central key storage — a KMS or a vault — can narrow access and record every operation, but each service allowed to decrypt remains trusted. You have not removed trust, you have concentrated and logged it.

Separate keys by environment, tenant, or purpose when the blast-radius reduction justifies the operational cost. A leaked staging key that cannot touch production data is an incident. A single key shared everywhere is a catastrophe.

My advice: design the key's life first — generation, storage, access, rotation — and only then write the encrypt call. The next lessons cover the encrypt call itself.

Practice

Create a key-access map for one encrypted field, including developers, CI, application replicas, backups, and recovery jobs. Save the effective permissions for each identity. Then revoke one test replica and prove it can no longer decrypt while another authorized replica still can.

Use authenticated encryption

Choose a high-level AEAD construction so ciphertext confidentiality and integrity are verified together before plaintext is accepted.

Encryption without authentication can allow controlled ciphertext changes. Attackers do not need to read your data to hurt you; sometimes flipping bits in ciphertext is enough to change what it decrypts to.

The failure looks like this: an application encrypts an account role but does not authenticate the ciphertext. A controlled bit change can alter decrypted data or produce distinguishable errors before the application notices. The data stayed confidential the whole time. It just stopped being trustworthy.

Authenticated encryption (AEAD) solves both problems in one construction. It encrypts and produces an authentication tag over the ciphertext, and decryption verifies the tag before releasing any plaintext. Use it by default. **AES-GCM** and **ChaCha20-Poly1305** are the two standard choices, and a high-level library exposes them through a safe API. Do not select raw cipher modes yourself.

AES-GCM in Node

```
import { createCipheriv, randomBytes } from 'node:crypto'

const key = randomBytes(32)          // 256-bit key
const nonce = randomBytes(12)        // 96-bit nonce, unique per encryption

const cipher = createCipheriv('aes-256-gcm', key, nonce)
const ciphertext = Buffer.concat([cipher.update('role=admin'), cipher.final()])
const tag = cipher.getAuthTag()      // 16 bytes
```

Store the nonce, ciphertext, and tag together. None of them is secret; only the key is.

Decryption verifies before it trusts:

```
import { createDecipheriv } from 'node:crypto'

const decipher = createDecipheriv('aes-256-gcm', key, nonce)
decipher.setAuthTag(tag)
const plaintext = Buffer.concat([decipher.update(ciphertext), decipher.final()])
// throws "Unsupported state or unable to authenticate data"
// if ciphertext, tag, or nonce was modified
```

Flip one byte of the ciphertext or the tag and `decipher.final()` throws. That throw is the security feature.

Fail closed, and fail uniformly

Decryption must fail closed when authentication fails and must not release partial plaintext. In stream-style APIs, do not act on any decrypted output until the final tag check passes.

The application must also treat every authentication failure the same. One generic error, one log path, no fallback decryption with a weaker mode or an older code path. Distinguishable failures are how attackers turn a small crack into an oracle.

Practice

Encrypt a short account record with a maintained AEAD API and save the complete versioned output fields. Decrypt it successfully, then flip one byte in the ciphertext and tag. Prove both failures return no plaintext and follow the same public error path.

Handle nonces and associated data

Follow the library's nonce requirements and authenticate unencrypted context such as record IDs, versions, and content types.

A **nonce** is a "number used once". It is not normally secret — you store it right next to the ciphertext — but reuse can destroy security for common encryption constructions.

For AES-GCM the rule is absolute: never reuse the same nonce with the same key. Two messages encrypted with the same key and nonce leak the XOR of their plaintexts, and nonce reuse can also let an attacker forge authentication tags. One repeated pair can compromise the key's entire authentication guarantee, not just the two messages involved.

How reuse happens in real systems

Nobody reuses a nonce on purpose. It happens through coordination failures. Two workers share an encryption key and each starts a nonce counter at zero after deployment. Their first messages reuse a nonce with the same key, breaking the construction assumptions.

Follow the exact API contract. Prefer library-managed nonces. When you must provide one, generate or count it according to the construction:

```
import { randomBytes } from 'node:crypto'

const nonce = randomBytes(12)   // fresh random 96-bit nonce per encryption
```

Random 12-byte nonces are safe for AES-GCM as long as a single key does not encrypt a very large number of messages; NIST guidance caps random-nonce use at 2^32 encryptions per key, and staying far below that is the comfortable position. Counters avoid collision entirely but demand durable, coordinated state — exactly what the two-workers failure lacked. When in doubt, use random nonces and rotate keys before the volume gets large.

Associated data binds context

AEAD APIs accept a second input: **associated data** (AAD). It remains visible — it is not encrypted — but it is protected from undetected modification, because the authentication tag covers it too.

Use it to bind ciphertext to its context, such as a record ID and schema version:

```
const cipher = createCipheriv('aes-256-gcm', key, nonce)
cipher.setAAD(Buffer.from('user:4821:v2'))
const ciphertext = Buffer.concat([cipher.update(address), cipher.final()])
```

Decryption must present the same AAD or the tag check fails. This prevents moving valid ciphertext to another record even though that context is not encrypted: an attacker copying user 4821's encrypted address onto user 977's row gets an authentication failure, not a silent swap.

Practice

Use a library-managed nonce to encrypt two records and save the nonce, record ID, and schema version with each result. Swap the record IDs during decryption and prove authentication fails. Then simulate two workers or a counter restart and show how the design prevents nonce reuse under the same key.

Use envelope encryption

Encrypt data with a data key and protect that key with a separate key-encryption service so rotation and access can be controlled centrally.

Encrypting every record directly with one long-lived master key makes rotation and access harder. Think about what rotating that key means: one long-lived master key directly encrypts millions of records, so rotating it requires reading and rewriting every record while the old key remains broadly available. Most teams quietly never rotate.

Envelope encryption separates roles so that problem disappears.

Two keys, two jobs

Generate a fresh **data-encryption key** (DEK) for each object or group of records, encrypt the data with it, then encrypt — "wrap" — that key with a **key-encryption key** (KEK) held by a managed KMS or vault. Store the wrapped data key with the ciphertext. The KEK never leaves the key service; the plaintext DEK exists only in application memory during the operation.

```
import { randomBytes, createCipheriv } from 'node:crypto'

const dek = randomBytes(32)                    // fresh per record
const nonce = randomBytes(12)
const cipher = createCipheriv('aes-256-gcm', dek, nonce)
const ciphertext = Buffer.concat([cipher.update(document), cipher.final()])

const wrappedDek = await kms.encrypt({ keyId: 'kek-prod-v2', plaintext: dek })
// the plaintext dek is discarded after this point
```

A stored record then carries everything needed to decrypt later, except the authority to do it:

```
{
  "v": 2,
  "kek_id": "kek-prod-v2",
  "wrapped_dek": "AQIDAHh3...",
  "nonce": "8kQzL1Xb9mPq",
  "tag": "5tR8...",
  "ciphertext": "..."
}
```

To read, the application asks the KMS to unwrap `wrapped_dek`, decrypts locally, and discards the DEK again. The KMS sees one small unwrap call per read, not your data.

Why this structure pays off

Rotating the wrapping key may avoid re-encrypting all data. You unwrap each DEK with the old KEK and rewrap with the new one. The bulk ciphertext bytes never move and the plaintext is

never exposed during rotation — you rewrite a 32-byte field per record instead of the record itself.

Access control moves to one place. The KMS decides who may unwrap keys, and its audit log shows every decryption request. Losing a database backup exposes wrapped keys only, which are useless without the KEK.

One caution: bind each wrapped key to its record with authenticated context, so a valid wrapped DEK cannot be moved onto a different record unnoticed.

Practice

Draw and save one encrypted record containing ciphertext, nonce, wrapped data key, key identifier, version, and authenticated context. Simulate rotating the wrapping key by rewrapping the data key and prove the ciphertext bytes stay unchanged. Then use the wrong record context and capture the expected decryption failure.

Check your understanding: symmetric encryption

Check shared keys, AEAD, nonces, associated data, storage formats, envelope encryption, and key separation.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Public-key cryptography

Understand key pairs, agreement, signatures, certificates, and authenticated TLS.

Understand public and private keys

Use a shareable public key and protected private key while keeping encryption, agreement, and signing purposes distinct.

Public-key cryptography splits key material into a public part and a private part. The two are mathematically linked: what one does, only the other can undo or verify. The private key never becomes public.

This solves the problem symmetric keys cannot: working with parties you have never shared a secret with. You publish the public key to anyone. You guard the private key like the secret it is.

```
import { generateKeyPairSync } from 'node:crypto'

const { publicKey, privateKey } = generateKeyPairSync('ed25519')

publicKey.export({ type: 'spki', format: 'pem' })
// -----BEGIN PUBLIC KEY-----   <- share this freely
privateKey.export({ type: 'pkcs8', format: 'pem' })
// -----BEGIN PRIVATE KEY-----  <- never leaves your control
```

One pair, one purpose

Depending on the construction, a key pair supports key agreement, encryption, or signatures. With signatures, the private key signs and anyone verifies with the public key. With encryption, anyone encrypts to the public key and only the private key decrypts. Some algorithms only do one job at all: Ed25519 keys sign, X25519 keys perform key agreement, and they are not interchangeable.

Do not assume one key pair should serve every purpose even when the math allows it. Keep signing, encryption, and key-agreement keys separate so compromise or misuse of one purpose does not cross into another. A leaked TLS key should never also be your release-signing key.

You will meet these pairs everywhere: your SSH key (`~/.ssh/id_ed25519` and `id_ed25519.pub`), TLS server keys, package and commit signing keys.

Public does not mean trusted

Here is the trap. Public keys still need an authenticated way to bind them to the expected identity.

An application downloads a public key from the same unauthenticated response as the signed update. An attacker replaces the update, signature, and public key together. Every mathematical check passes, because the attacker signed their malware with their own key and handed you the matching verifier.

Public means shareable, not automatically trusted. The binding between a key and an identity has to come from somewhere the attacker does not control: a certificate chain, a key pinned at install time, or a fingerprint verified out of band. That binding problem is what the rest of this module is about.

Practice

Classify the public and private material for an SSH login, TLS site, and package-signing workflow, and save how each public key gains trust. Verify one signed test message with the expected public key. Then substitute a new attacker-controlled key pair and show why mathematical verification alone is insufficient.

Agree on shared keys

Use an established authenticated key-agreement protocol to derive session keys without sending the final shared key across the network.

Key agreement lets two parties derive shared secret material over a public channel, without ever sending the secret itself. Diffie-Hellman is the classic construction; **X25519** is the modern elliptic-curve version you will actually meet.

Each side generates a key pair, they exchange public keys, and each combines its own private key with the other's public key. Both arrive at the same secret. An observer who saw both public keys cannot compute it.

```
import { generateKeyPairSync, diffieHellman } from 'node:crypto'

const alice = generateKeyPairSync('x25519')
const bob = generateKeyPairSync('x25519')

const aliceSecret = diffieHellman({ privateKey: alice.privateKey, publicKey: bob.publicKey })
```

```
const bobSecret = diffieHellman({ privateKey: bob.privateKey, publicKey: alice.publicKey })

aliceSecret.equals(bobSecret) // true
```

The part the math does not cover

Key agreement does not automatically prove who the other party is. You derived a secret with someone. The math never says with whom.

A client and server perform raw Diffie-Hellman over an untrusted network. An active attacker intercepts the exchange and establishes one shared key with the client and another with the server, reading and changing traffic between them. Both victims see a working encrypted channel. This is the textbook man-in-the-middle attack, and unauthenticated key agreement walks straight into it.

Real protocols combine ephemeral key agreement with signatures, certificates, or pre-established identity. In TLS 1.3, the server signs the handshake transcript with the private key its certificate vouches for, so the client knows the agreement happened with the certificate's owner. "Ephemeral" means fresh agreement keys per connection, which gives forward secrecy: a later key compromise does not decrypt recorded traffic.

Derive, never use raw

The raw shared secret is not your encryption key. Feed the result through the protocol's key derivation and use separate keys by purpose:

```
import { hkdfSync } from 'node:crypto'

const clientKey = Buffer.from(hkdfSync('sha256', aliceSecret, salt, 'client write key', 32))
const serverKey = Buffer.from(hkdfSync('sha256', aliceSecret, salt, 'server write key', 32))
```

HKDF stretches one secret into independent keys, each bound to a purpose label.

Do not assemble a handshake from raw primitives. Key agreement needs authenticated identities and a defined key schedule, and getting either subtly wrong is invisible until someone exploits it. Use a protocol such as current TLS instead of composing raw agreement, signatures, and derivation yourself.

Practice

Trace a TLS connection and save where the ephemeral agreement, certificate identity, transcript authentication, and derived traffic keys appear conceptually. Verify the connection with the expected hostname. Then connect through a test endpoint with the wrong identity and capture the authentication failure.

Sign and verify data

Use digital signatures to bind exact bytes to a private-key holder and verify identity, context, format, and freshness before trusting the result.

A digital signature is created with a private key and verified with the public key. It proves more than a checksum, but less than "this action is safe."

Unlike an HMAC, verification needs no secret. Anyone holding the public key can check the signature, and nobody without the private key can forge one. That asymmetry gives you non-repudiation:

the signer cannot claim a verifier forged the message, because verifiers never held signing ability.

Ed25519 is the common modern choice, and Node supports it directly:

```
import { generateKeyPairSync, sign, verify } from 'node:crypto'

const { publicKey, privateKey } = generateKeyPairSync('ed25519')

const manifest = Buffer.from(JSON.stringify({
  project: 'shipd', version: '2.4.1',
  artifact: 'sha256:7c2df1a9...', purpose: 'release', ts: 1754236800
}))

const signature = sign(null, manifest, privateKey)
verify(null, manifest, publicKey, signature) // true
```

Change one byte of the manifest and `verify` returns `false`.

Sign exact bytes, with context

Signatures cover bytes, not meaning. Define a canonical or exact byte representation and verify the bytes you received, before any parsing and re-serialization. JSON with unstable key order is a classic way to sign one thing and verify another.

Include purpose and version inside the signed payload. A signature over bare data can be replayed in a different context; a signature over `purpose: 'release'` cannot be presented as a login token.

A valid signature is not approval

Verify with the expected public key — not a key that arrived with the message. Then keep going, because policy questions remain.

Here is the trap: a deployment manifest has a valid signature but names an old artifact with a known vulnerability. The signature proves who signed those bytes, not that the release is current or approved now. This rollback pattern is a real attack on update systems: the attacker replays your own legitimately signed old release.

So a valid signature from an unexpected signer, or for an old message, should still be rejected by policy. Verification must also enforce the expected signer, project, freshness, and authorization. The cryptography answers ”who signed these exact bytes.” Everything after that is your application's decision.

Practice

Define and save the exact signed bytes for a release manifest containing project, version, artifact digest, purpose, and timestamp. Verify a valid current manifest with the expected public key. Then replay an old valid manifest and prove a freshness or release-policy check rejects it.

Understand certificates and TLS

See how certificate chains bind public keys to names and why hostname, validity, chain, and trust-anchor verification must remain enabled.

A TLS certificate connects a public key to an identity such as a domain name through a chain of

signatures. A certificate authority signs the server's certificate, an already-trusted root signs the CA's certificate, and your system ships with the roots preinstalled. That chain is how your client trusts a public key it has never seen before.

You can watch the whole thing:

```
openssl s_client -connect flaviocopes.com:443 -servername flaviocopes.com
# Certificate chain
#  0 s:CN=flaviocopes.com
#    i:C=US, O=Let's Encrypt, CN=E7
#  1 s:C=US, O=Let's Encrypt, CN=E7
#    i:C=US, O=Internet Security Research Group, CN=ISRG Root X2
# Verify return code: 0 (ok)
```

During the handshake, at a high level: the client sends supported algorithms, the two sides run an ephemeral key agreement, the server presents this chain and proves possession of the private key by signing the handshake transcript. Only then do encrypted application bytes flow.

What verification checks

The client verifies the hostname (the certificate names the domain you asked for), the validity period, key usage, the certificate chain, and that it terminates at a trusted root. Inspect those fields yourself:

```
openssl x509 -in cert.pem -noout -subject -dates -ext subjectAltName
# subject=CN=flaviocopes.com
# notBefore=Jun 12 08:21:33 2026 GMT
# notAfter=Sep 10 08:21:32 2026 GMT
# X509v3 Subject Alternative Name:
#     DNS:flaviocopes.com, DNS:www.flaviocopes.com
```

Every check answers a specific attack. Skip hostname verification and a valid certificate for `attacker.dev` satisfies a connection to your API.

Never disable verification

Here is the incident that repeats everywhere. A developer disables certificate verification to fix a staging connection error. Traffic stays encrypted, but the client will now establish that encrypted connection with an attacker presenting any certificate. Disabling verification turns encryption into a connection with an unknown party — and the flag ships to production, because these flags always do.

Private certificate authorities and local test certificates need deliberate trust configuration instead: add your CA to the trust store, or pass it explicitly (`NODE_EXTRA_CA_CERTS=ca.pem` in Node). A global verification bypass turns one environment problem into a system-wide identity failure.

Certificates expire, so automate renewal and monitor expiry. An expired certificate is a full outage with a known date attached.

Practice

Inspect a real test certificate and save its names, validity period, key usage, issuer chain, and trusted root. Connect with the correct hostname and trust store. Then test a wrong hostname, expired certificate, or untrusted root and prove verification fails without a global bypass.

Check your understanding: public-key cryptography

Check key pairs, key agreement, signatures, verification, certificates, TLS, and trust anchors.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Keys and operations

Generate, store, rotate, revoke, version, migrate, and review cryptographic systems.

Generate keys securely

Use cryptographically secure platform or key-management APIs with the exact size and algorithm required instead of deriving keys from memorable text.

Cryptographic keys need enough unpredictable bits. A 256-bit key is only as strong as 256 bits if every bit came from a source an attacker cannot model. Human-created phrases and ordinary random functions do not provide that.

`Math.random()` is the classic mistake in JavaScript. It exists for shuffling animations, not secrets: it is not a **CSPRNG** (cryptographically secure pseudorandom number generator), and its outputs can be predicted from previous outputs. The same goes for timestamps, process IDs, and anything else an attacker can guess or observe.

Use the platform's secure source

Let a maintained library, operating system, HSM, or KMS generate keys. In Node, `node:crypto` reads from the operating system's CSPRNG:

```
import { randomBytes } from 'node:crypto'

const key = randomBytes(32)
```

That is a full-strength 256-bit key. On the command line, openssl does the same job:

```
openssl rand -hex 32
# 41d1e2c7a80f5b9e6d3c7f2a1b8e4d90c5a6f3e2d1b0a9c8e7f6d5c4b3a29180
```

Match the exact size and algorithm the construction requires: 32 bytes for AES-256, 12 bytes for a GCM nonce, key pairs through `generateKeyPairSync` rather than raw bytes. For random integers and identifiers, use `randomInt()` and `randomUUID()` from the same module, never `Math.random()`.

Why derived-from-text keys fail

A developer derives an encryption key with SHA-256 from a memorable deployment phrase. The output looks perfectly random — 64 hex characters, passes every eyeball test. But an attacker who obtains ciphertext can test likely phrases offline much faster than searching a random key space. The key space collapsed from 2^256 to "phrases a person would pick."

Use a password KDF only when a user password must derive key material — an unavoidable input, like encrypting a local vault with the user's passphrase. Then use a real password KDF with a

salt and current parameters, never a bare hash, and accept that the key is only as strong as the password.

Two operational rules. Never print generated keys while debugging: logs outlive the debugging session and flow into systems with wider access. And if key generation fails, it must fail loudly — a fallback to a timestamp-seeded generator is a silent catastrophe.

Practice

Trace the generation API and random source for every key in one small application, and save algorithm plus length metadata without saving key bytes. Generate two keys and prove they differ and satisfy the library format. Then block or mock the random source and confirm generation fails instead of falling back to a timestamp or ordinary random function.

Store and limit keys

Keep keys out of source and normal logs, use managed key services where appropriate, and grant only required encrypt, decrypt, sign, or verify operations.

A key's storage and permissions are part of the cryptographic design. The strongest algorithm cannot protect an exposed key, and most real-world crypto failures are key-handling failures, not broken math.

Start with the obvious exclusions. Keys never go in source control, and never in normal logs. A key committed once lives in git history forever, so treat any committed key as compromised and rotate it — deleting the file changes nothing:

```
git log -p --all -S 'BEGIN PRIVATE KEY' -- '*'
# any hit here means the key is burned, even if the file is gone
```

Where keys should live

Prefer a KMS, HSM, secure enclave, or protected secret store that can restrict operations and record use. The ladder looks like this: environment variables injected at deploy are better than files in the repo; a secret manager with access control and audit is better than environment variables; a KMS that performs operations without releasing the key is better still.

That last step changes the model. A managed key service can expose only a sign operation and keep raw key material inside the service. Your code sends bytes and receives a signature. Nothing that runs in your process can copy a key that never enters your process. This narrows authority, but access policy and audit logs become part of the cryptographic system — misconfigure the policy and the HSM protects an operation anyone can invoke.

Grant operations, not keys

Permissions should name operations, not just keys. An identity that verifies webhooks needs verify, never sign. A service that encrypts uploads needs encrypt, not decrypt:

```
identity            key            allowed
api-server          webhook-hmac   verify
upload-worker       storage-kek    encrypt
restore-job         storage-kek    decrypt
release-pipeline    signing-v3     sign
```

Here is why the narrow grant matters. A signing key is available as plaintext to the whole CI job, including formatting and test steps. A compromised test dependency can copy the same authority used for releases. Scope the key to the one signing step, or better, keep it in a service that only that step's identity may call.

Separate keys by environment and purpose, and keep data and keys under different access paths when the architecture allows it. A backup containing both ciphertext and its keys protects nothing.

Practice

Build a key inventory with purpose, owner, environment, storage, allowed operations, users, rotation, and revocation. Save the effective policy and one audit event for an authorized test operation. Then attempt a forbidden operation, such as decrypting with a sign-only identity, and capture the denial.

Rotate, version, and revoke keys

Attach key and format versions to protected data, support controlled migration, and prepare immediate revocation for suspected compromise.

Keys and algorithms have lifecycles. Keys leak, employees leave, algorithms age out of guidance. Design the data format so old records can be read while new writes use the current key — because one day you will rotate under pressure, and that is the wrong day to invent a migration.

Version everything you protect

Store a non-secret key identifier and format version with ciphertext or signatures:

```
{
  "v": 2,
  "kid": "field-key-2026-08",
  "nonce": "9mPqL2Xb8kQz",
  "tag": "4sR7...",
  "ciphertext": "..."
}
```

Reads become a lookup instead of a guessing game:

```
const keyring = {
  'field-key-2026-02': oldKey,   // decrypt only
  'field-key-2026-08': currentKey,
}

const key = keyring[record.kid]
if (!key) throw new Error(`unknown key id: ${record.kid}`)
```

Compare that with the failure mode: ciphertext records store no key identifier. After rotation, the application must try every historical key, creating slow failures and making retirement unsafe — you can never prove a key is unused, so you can never delete it.

Rotate in phases

A versioned format supports new writes with the current key and controlled reads with older keys. Rotation becomes routine:

```
phase 1: add key v2 to the keyring        (reads: v1+v2, writes: v1)
phase 2: switch writes to v2              (reads: v1+v2, writes: v2)
phase 3: re-encrypt or expire v1 records
phase 4: retire v1 to decrypt-only, then delete per retention policy
```

Keep old decrypt-only keys during migration, then retire them according to retention needs. Watch a metric of reads per key version; phase 4 starts when v1 reads hit zero.

Revocation is not rotation

Rotation is planned. **Revocation** is what you do when a key may be compromised, and it must be fast: stop trusting the key now, not after a migration.

For signing keys that can mean revoking trust and issuing a clean replacement, plus deciding whether things signed earlier remain acceptable. For encryption keys, immediate revocation can make old data unavailable — records encrypted under the revoked key stop being readable. So compromise response and data recovery need an explicit tradeoff, written down before the incident: which data you would sacrifice, and who makes that call at 3am.

Practice

Write and save a migration plan from key version 1 to version 2 using new-write and old-read phases. Encrypt records under both versions and prove the key identifier selects the correct key without trial decryption. Then revoke version 1 in a test environment and record which reads fail and how the recovery decision is made.

Complete a cryptography review

Review goals, algorithms, libraries, formats, randomness, keys, failure behavior, migration, and recovery before relying on a cryptographic design.

A crypto review asks whether the complete system provides the intended property. Algorithm names alone are not enough — every lesson in this course was a piece of that sentence, and the review is where the pieces meet.

Here is what "the algorithm is fine, the system is broken" looks like. A webhook uses HMAC-SHA-256, so the team marks the design secure. But the same secret is shared across environments, raw bytes are not preserved before verification, replay is allowed, and failures leak which check failed. Four exploitable gaps, zero broken algorithms.

Walk a fixed checklist

Review against a list, not against vibes. Check the threat model, high-level construction, current guidance, nonce rules, authenticated context, key access, error handling, versioning, backup, revocation, and tests. In practice I structure it like this:

```
goal        which property, against which attacker?
construct   maintained AEAD / HMAC / signature API, not raw primitives?
randomness  every key and nonce from a CSPRNG?
nonces      reuse impossible by design, not by discipline?
context     record IDs, versions, purpose in authenticated data?
keys        who can use each key, for which operations, logged where?
failures    fail closed, uniform errors, no partial plaintext?
```

```
format      key id + version stored with every output?
recovery    rotation rehearsed, revocation decision written down?
```

The review is only real if you exercise the failure paths. Verification succeeding proves little; what matters is that tampering, replay, wrong context, an old key version, and malformed input all fail — safely, uniformly, and visibly in your logs:

```
node --test crypto-review.test.js
#   rejects modified ciphertext (1.2ms)
#   rejects replayed webhook delivery (0.9ms)
#   rejects wrong record context (1.1ms)
#   unknown key id fails without trial decryption (0.4ms)
```

Review the surroundings too

The algorithm name is only one input. Review data formats, key access, nonce or replay rules, failure behavior, migration, recovery, and the authorization decision that follows verification. That last one is subtle: a verified webhook still needs an "is this event allowed to do that?" check, and reviews keep finding code that treats "signature valid" as "action approved."

Track standards changes, including post-quantum transitions, without inventing a premature hybrid protocol. Knowing which data would need re-protection, and having versioned formats ready, is preparation enough for most teams today.

Schedule the review to recur. A design that passed in 2024 has aging parameters and possibly deprecated algorithms by 2026.

Practice

Review one encrypted field or signed webhook and save a diagram plus a table of goals, library, format, keys, failures, migration, and remaining assumptions. Run successful verification and preserve the evidence. Then test tampering, replay, wrong context, old key version, and malformed input, and record that every failure is safe and observable.

Check your understanding: keys and operations

Check randomness, key generation, storage, rotation, revocation, versioning, backups, post-quantum planning, and cryptographic review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/cryptography-for-developers/.