A04:2025 – Cryptographic Failures

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

- Introduction
- Key Concepts
- Complex Attack Scenarios
- Prevention
- Conclusion

Introduction

What is a Cryptographic Failure?

- **Definition (OWASP)**
  - “Failures related to the lack of cryptography, insufficiently strong cryptography, leaking of cryptographic keys, and related errors”
- **It is about protecting data, not just “using crypto”**
  - Data **in transit**, data **at rest**, and secrets (keys)
  - The failure is usually in *how* crypto is used, not in the maths
- **Formerly “Sensitive Data Exposure”**
  - That old name described the *symptom*; this one names the *cause*
- **What is at stake**
  - Passwords, credit-card numbers, health records, personal data, business secrets – often under GDPR, PCI DSS, HIPAA obligations

A04:2025 – Down from #2 to #4

- **Was #2 in 2021**, moved down two places
- **Many top weaknesses are about randomness**
  - Insufficient entropy, predictable RNG algorithms, weak PRNGs – next to broken/risky algorithms
- **Transport encryption is now easy**
  - Hardware acceleration + free, automated certificates removed the old excuses
- High occurrence count across the industry

CWEs mapped	32
Max incidence rate	13.77%
Avg incidence rate	3.80%
Max coverage	100%
Avg coverage	47.74%
Avg weighted exploit	7.23
Avg weighted impact	3.90
Total occurrences	1 665 348
Total CVEs	2 185

Common Cryptographic Failures (OWASP)

- **No encryption** where it is needed, or not *enforced* (plaintext HTTP, FTP, SMTP for confidential data)
- **Outdated or weak algorithms**: MD_5, SHA-1, DES, RC_4, CBC mode, PKCS#1 v1.5
- **Weak key generation / management**; **hard-coded keys** in repositories
- **Improper certificate validation** (accepting any certificate)
- **Insecure modes and IVs**: ECB mode, reusing an IV/nonce with the same key
- **Passwords used directly as cryptographic keys**
- **Insufficient randomness**: predictable tokens, weak PRNG seeding
- **Deprecated hashes** for passwords; **unsalted** or fast hashes
- **Informative crypto error messages** (padding oracles)

Mapped CWEs (32 total – a selection)

- **CWE-319** Cleartext Transmission
- **CWE-327** Broken/Risky Crypto Algorithm
- **CWE-328** Use of Weak Hash
- **CWE-326** Inadequate Encryption Strength
- **CWE-916** Password Hash w/o Sufficient Effort
- **CWE-759/760** Hash without (unique) Salt
- **CWE-347** Improper Verification of Signature
- **CWE-330** Insufficiently Random Values
- **CWE-331** Insufficient Entropy
- **CWE-338** Crypto Weak PRNG
- **CWE-1241** Predictable Algorithm in RNG
- **CWE-329** Predictable IV with CBC Mode
- **CWE-321** Hard-coded Cryptographic Key
- **CWE-296** Improper Cert Chain Validation

Source: `https://top10.owasp.org/2025/A04_2025-Cryptographic_Failures/`

Why Cryptographic Failures Persist

- **Crypto is easy to use *wrongly***
  - The code “works” and produces ciphertext – but offers little security
  - Mistakes are silent: no error, no crash, no failing test
- **Defaults and old tutorials**
  - Copy-pasted examples using ECB, MD5, a fixed IV, a static key
- **Secrets are hard to manage**
  - Keys end up in source, configs, images – see A02 (Misconfiguration) and A03 (Supply Chain)
- **Harvest now, decrypt later**
  - Captured ciphertext can be broken *years* later – weak crypto or a future quantum computer; plan the migration now

Key Concepts

Five Concepts to Understand Cryptographic Failures

1. **Data in transit**
   - TLS, downgrade attacks, HSTS, certificate validation
2. **Data at rest and key management**
   - Encrypting stored data; where keys live; hard-coded keys
3. **Password storage**
   - Salting and slow, adaptive hashing – not encryption
4. **Randomness and entropy**
   - Unpredictable tokens, keys and IVs; CSPRNGs
5. **Choosing primitives and modes**
   - Strong algorithms, authenticated encryption, no ECB, unique IVs

Concept 1: Protecting Data in Transit

- **Encrypt everything at the transport layer**
  - TLS 1.2 or 1.3 only, forward-secrecy ciphers; drop CBC and legacy protocols (SSLv3, TLS 1.0/1.1)
- **Encryption must be *enforced*, not merely offered**
  - Redirect HTTP → HTTPS and set **HSTS** so the browser refuses plaintext
- **Validate certificates properly**
  - Check the chain, hostname and expiry; never “trust all certificates” (a frequent bug in mobile apps and API clients)
- **Avoid plaintext protocols** for sensitive data
  - FTP, Telnet, plain SMTP; also mind internal service-to-service traffic

Example: HTTPS Downgrade and Cookie Theft

- **OWASP Scenario #1**: a site that does not enforce TLS



- The attacker keeps the victim on **HTTP**, reads the **session cookie**, and **hijacks the authenticated session**
- Historically trivial on open Wi-Fi (the *Firesheep* extension, 2010)
- **Fixes**
  - HTTPS everywhere + `Strict-Transport-Security` (with preload)
  - `Secure`, `HttpOnly`, `SameSite` on cookies (see A02)

Example: Disabled Certificate Validation

```python
import requests
# "It threw an SSL error, so I turned verification off." -- a very common bug
requests.get("https://api.bank.example/transfer", verify=False)
```

```java
// Android/Java: trust manager that accepts ANY certificate
public void checkServerTrusted(X509Certificate[] c, String a) { /* nothing */ }
```

- TLS still “works” – but any machine-in-the-middle can present its own certificate and read/modify the traffic
- The encryption is intact; the **authentication** of the endpoint is gone
- **Fix**: validate the chain, hostname and expiry; fix the *root cause* of the error (bad CA bundle, wrong hostname) instead of disabling checks; consider certificate pinning for high-value clients

Concept 2: Data at Rest and Key Management

- **Decide what must be encrypted**
  - Classify data; apply controls matching the classification
  - Minimise retention: tokenize or truncate (e.g. PCI DSS) what you do not need to keep
- **The key is the secret, not the algorithm**
  - “The security of the system should depend only on the key” (Kerckhoffs’s principle)
  - A **hard-coded key** (CWE-321) in the repo protects nothing
- **Key management is the hard part**
  - Generate keys with a CSPRNG; store them in an HSM or KMS
  - Separate keys from the data they protect; rotate; control access; audit

Example: Hard-coded Key and Separation

Broken – key lives next to the ciphertext, in the source

```
KEY = b"0123456789abcdef"            # committed to Git
token = AES_encrypt(KEY, user_data)  # anyone with the repo can decrypt
```

- If the database leaks *and* the repo leaks, both fall together – and they often leak together (backups, images, CI logs)

Better – key from a managed store, never in code

```
key = kms.get_data_key(key_id="app/records")   # HSM/KMS, access-controlled
ct  = aead_encrypt(key, nonce, user_data, aad)  # authenticated encryption
```

- Envelope encryption: a per-record data key wrapped by a master key in the KMS; the application never sees the master key
- Rotate keys; log access; the breach of the data store alone is not enough

Concept 3: Storing Passwords

- **Do not encrypt passwords – hash them**
  - Encryption is reversible; you never need the original password back
- **Three properties a password hash needs**
  - **Salt** (CWE-759/760): unique per user ⇒ defeats rainbow tables and reveals nothing about equal passwords
  - **Slow / adaptive** (CWE-916): a tunable work factor ⇒ makes mass guessing expensive
  - (modern choices also add memory-hardness against GPUs/ASICs)
- **Use a vetted algorithm**
  - Argon2id, scrypt, bcrypt, or PBKDF2-HMAC-SHA-512 with a high iteration count
  - **Never** MD5/SHA-1/SHA-256 *alone* – they are fast by design

Example: From Fast Hash to Adaptive Hash

Broken – fast, unsalted hash

```
import hashlib
stored = hashlib.sha256(password.encode()).hexdigest()
# identical passwords -> identical hashes; GPUs try billions/sec; rainbow tables
```

Correct – salted, slow, adaptive (the library handles salt + parameters)

```
from argon2 import PasswordHasher
ph = PasswordHasher()                # Argon2id, sensible defaults
stored = ph.hash(password)           # embeds algorithm, salt, work factors
ph.verify(stored, password)          # constant-time verification
```

- **Why the work factor matters**: an unsalted SHA-256 database can be cracked at enormous speed; a well-tuned Argon2 makes each guess thousands of times more expensive
- Re-hash on login when parameters are increased; add a server-side “pepper” kept in a secret store for defence in depth

Concept 4: Randomness and Entropy

- **Security often rests on “unguessable”**
  - Session IDs, password-reset tokens, API keys, CSRF tokens, nonces, IVs, and the keys themselves
- **Not all randomness is equal**
  - `rand()`, `Math.random()`, `java.util.Random` are **predictable** (CWE-338): observe a few outputs, recover the state, predict the rest
  - Low entropy / bad seeding (time, PID) ⇒ guessable values (CWE-330/331)
- **Use a CSPRNG**
  - `secrets` (Python), `crypto.randomBytes` (Node), `SecureRandom` (Java), `/dev/urandom`
- **IVs/nonces**: must be unique per message; **never reuse an IV with the same key** (CWE-323); for CBC also unpredictable (CWE-329)

Example: Predictable Token

Broken – a reset token from a non-crypto PRNG

```
// predictable: the xorshift128+ state can be recovered from a few outputs
const token = Math.random().toString(36).slice(2);
sendResetLink(user, token);
```

- An attacker who sees a few tokens can **predict the next one** and take over accounts without ever receiving the e-mail

Correct – a CSPRNG with enough entropy

```
const token = require('crypto').randomBytes(32).toString('hex'); // 256 bits
```

- **IV reuse** is the same family of bug: with a stream cipher or CTR/GCM mode, reusing an IV with the same key leaks plaintext relationships (and breaks GCM's authentication entirely)
- Rule: fresh, random (or strictly counter-based) IV per encryption

Concept 5: Primitives and Modes

- **Prefer authenticated encryption (AEAD)**
  - AES-GCM, ChaCha20-Poly1305 – confidentiality *and* integrity
  - Encryption without integrity invites tampering and padding-oracle attacks
- **Never use ECB mode**
  - Identical plaintext blocks ⇒ identical ciphertext blocks: structure leaks straight through
- **Avoid deprecated building blocks**
  - MD5, SHA-1 (collisions), DES/3DES, RC4, CBC without integrity, PKCS#1 v1.5
- **Do not invent your own crypto**
  - Use vetted libraries and high-level, misuse-resistant APIs (libsodium, Tink)
  - Have specialists review the choices and configuration

Example: Why ECB Leaks

```
# Broken: ECB encrypts each 16-byte block independently
cipher = AES.new(key, AES.MODE_ECB)
ct = cipher.encrypt(pad(image_bytes, 16))   # same block in -> same block out
```

- The classic “ECB penguin”: an encrypted bitmap still shows the picture, because repeated regions produce repeated ciphertext
- The same leak applies to any structured data (fixed headers, repeated fields, tokens)

Correct – authenticated encryption with a unique nonce

```
aes = AES.new(key, AES.MODE_GCM, nonce=os.urandom(12))
ct, tag = aes.encrypt_and_digest(data)   # integrity + no block patterns
```

- Store `nonce` and `tag` with the ciphertext; verify the tag on decryption

Complex Attack Scenarios

Real-World Attacks

- **Crypto failures turn a data *leak* into a data *breach***
  - Encryption done right means a stolen database is still useless
- **Scenarios studied**
  1. Session sidejacking over HTTP (Firesheep, 2010)
  2. Adobe (2013): ECB + reused key + password hints
  3. Unsalted/fast password hashes cracked at scale (LinkedIn, 2012)
  4. WEP: IV reuse breaks a whole protocol
  5. Heartbleed (2014): leaking the private key itself
  6. A chained scenario: from weak hash to full account takeover

Adobe (2013): How *Not* to Protect Passwords

- ∼**153 million** accounts leaked, including the password column
- **The mistakes compounded**
  - Passwords were **encrypted** (reversible) instead of hashed
  - A **symmetric cipher in ECB mode** with a **single key** for everyone
  - So **identical passwords produced identical ciphertext**
  - The **password hints** were stored in clear text
- **Result**
  - Without ever recovering the key, analysts grouped equal ciphertexts and used the plaintext hints to deduce enormous numbers of passwords
  - The most common passwords became a widely shared teaching example
- **Lessons**: hash (don't encrypt) passwords; never ECB; salt per user; never store reversible hints

LinkedIn (2012): Unsalted, Fast Hashes

- **The storage**: passwords hashed with **unsalted SHA-1**
- **Why that is weak**
  - SHA-1 is **fast** – GPUs test billions of guesses per second
  - **No salt** ⇒ precomputed rainbow tables apply directly, and identical passwords share the same hash
- **Outcome**
  - A very large fraction of the hashes was cracked quickly
  - The full scale (over 100 million credentials) surfaced years later and fuelled **credential-stuffing** against other sites
- **Lessons**
  - Use salted, slow, adaptive hashing (Argon2/bcrypt/scrypt/PBKDF2)
  - A breach is not a one-time event: cracked passwords are reused for years

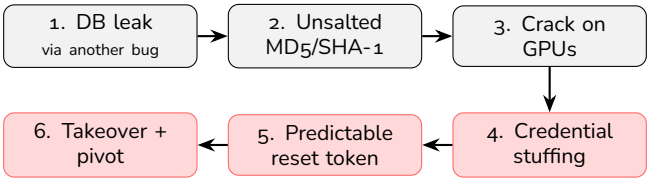
WEP: When IV Reuse Breaks a Protocol

- **Context**: WEP was the original Wi-Fi encryption standard
- **The flaw is in the *use* of the cipher**
  - A stream cipher ($RC4$) keyed by (shared key $\|$ IV)
  - The **IV was only 24 bits** and sent in clear – on a busy network it **repeats within hours**
  - Reused IV + same key $\Rightarrow$ same keystream $\Rightarrow$ XOR of two ciphertexts reveals plaintext relationships
- **Result**
  - Statistical attacks recover the key from captured traffic in **minutes**; WEP is completely broken
- **Lessons**
  - A strong cipher used with a tiny, reused IV gives *no* security
  - Nonce/IV management is not a detail – it is the attack

Heartbleed (CVE-2014-0160): Leaking the Key Itself

- **Context**: a bug in the OpenSSL TLS “heartbeat” extension
- **What happened**
  - A missing bounds check let a client ask the server to return *more memory* than it should – up to 64 KB per request, repeatedly
  - That memory could contain session cookies, credentials, and crucially the server’s **TLS private key**
- **Why it is a cryptographic failure**
  - TLS was configured correctly, yet the **secret that underpins it** could be exfiltrated – and retroactively decrypt captured traffic if no forward secrecy
- **Lessons**
  - After key exposure: **rotate keys and revoke/reissue certificates** – patching alone is not enough
  - Prefer forward-secrecy ciphers so past traffic stays safe

Kill Chain: From a Weak Hash to Account Takeover



- Each step is a cryptographic failure: no salt, fast hash, reused passwords, weak randomness
- **Any one fix** – salted adaptive hashing, CSPRNG tokens, MFA – breaks the chain

Kill Chain: Why Each Step Works

- **1. The leak**: an unrelated flaw (SQLi, misconfig, backup) exposes the user table – encryption at rest only helps against stolen disks and backups, not against SQLi through the application
- **2–3. Weak storage**: unsalted, fast hashes are cracked en masse on commodity GPUs; adaptive salted hashing would make this uneconomical
- **4. Reuse**: users reuse passwords, so cracked credentials open accounts on *other* services (credential stuffing)
- **5. Weak tokens**: a predictable password-reset token lets the attacker seize even accounts with unique passwords
- **6. Pivot**: a taken-over admin or a shared password reaches further systems
- **Defense in depth**: encrypt at rest, hash properly, use CSPRNG tokens, enforce MFA, detect stuffing (rate limits, anomaly detection)

Prevention

How to Prevent Cryptographic Failures

- **Classify data**; encrypt sensitive data **at rest**, and apply controls matching the classification; minimise retention (tokenize/truncate)
- **Encrypt all data in transit**: TLS $\geq$ 1.2, forward secrecy, drop CBC/legacy; enforce with HSTS; validate certificates
- **Use strong, current algorithms** and authenticated encryption; avoid MD5/SHA-1/DES/RC4/ECB/PKCS#1 v1.5
- **Hash passwords** with Argon2id / scrypt / bcrypt / PBKDF2-SHA-512 – salted, with a real work factor
- **Manage keys**: generate with a CSPRNG, store in HSM/KMS, separate from data, rotate; never hard-code keys
- **Randomness**: CSPRNG everywhere; unique IV/nonce per message
- **Review** crypto choices with specialists; **plan for post-quantum** (protect high-risk systems by $\sim$2030)

Making Good Crypto the Easy Path

- **Prefer high-level, misuse-resistant libraries**
  - libsodium, Google Tink, the platform's vetted APIs – they pick modes, IVs and parameters for you

```
# Tink / libsodium style: you ask for "encrypt", not for a mode and an IV
aead = keyset.primitive(Aead)
ct   = aead.encrypt(plaintext, associated_data)   # AEAD, nonce handled safely
```

- **Automate the checks**
  - TLS scanners (testssl.sh, SSL Labs), HSTS and cipher policy in CI
  - Linters/static analysis for ECB, MD5/SHA-1, `verify=False`, non-crypto RNG, hard-coded keys; secret scanning (see A03)
- **Have a key-compromise playbook**: rotate keys, reissue certs, force password resets – rehearsed in advance

Conclusion

Conclusion

- **Cryptographic failures are about *use*, not maths**
  - The algorithms are fine; the salting, modes, IVs, keys and randomness are where it breaks
- **Three domains to protect**
  - In transit (TLS, HSTS, cert validation); at rest (encryption + key management); secrets (keys, passwords)
- **Passwords are hashed, not encrypted**
  - Salted, slow, adaptive – Argon2/bcrypt/scrypt/PBKDF2
- **Randomness is a security primitive**
  - CSPRNG for tokens and keys; never reuse an IV/nonce with a key
- **The key is the secret**
  - Never hard-code it; store it in an HSM/KMS, separate from the data

Conclusion (Cont.)

- **Good crypto neutralises a breach**
  - A stolen database of properly encrypted data / adaptive hashes is of little use to the attacker
- **Failures compound (Adobe)**
  - Reversible storage + ECB + single key + clear-text hints = catastrophe
- **Breaches echo for years**
  - Cracked passwords feed credential-stuffing; “harvest now, decrypt later” threatens weak or non-PQ crypto
- **Make the secure path the easy path**
  - Misuse-resistant libraries, enforced TLS, automated crypto checks in CI, a rehearsed key-compromise response, and a post-quantum migration plan

References

- OWASP Top 10:2025 – A04 Cryptographic Failures
  `https://top10.owasp.org/2025/A04_2025-Cryptographic_Failures/`
- OWASP Cheat Sheets – Password Storage, Cryptographic Storage, Transport Layer Protection, Key Management
  `https://cheatsheetseries.owasp.org/`
- OWASP ASVS V6 (Cryptography) and V9 (Communications)
- Mozilla TLS configuration guidelines; `testssl.sh`; SSL Labs
- NIST SP 800-57 (Key Management), SP 800-63B (secrets), SP 800-131A
- libsodium `https://doc.libsodium.org/`; Google Tink
- Heartbleed: CVE-2014-0160. WEP weaknesses: Fluhrer–Mantin–Shamir (2001)
- Adobe (2013) and LinkedIn (2012) breach analyses; Firesheep (2010)