A02:2025 – Security Misconfiguration

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

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

Introduction

What is Security Misconfiguration?

- **Definition (OWASP)**
  - “A system, application, or cloud service is set up incorrectly from a security perspective, creating vulnerabilities”
- **The code may be perfect – the deployment is not**
  - Default passwords, debug mode in production, open cloud storage, missing security headers, sample applications left on the server…
- **Every layer can be misconfigured**
  - Operating system, web server, application server, framework, database
  - Containers, orchestrators, cloud services, CDN, CI/CD pipelines
  - The browser itself, through the directives sent by the server
- **Consequences**
  - From information disclosure (versions, stack traces)
  - …to complete compromise of the server or of the cloud account

A02:2025 – Up from #5 to #2

- **Was #5 in 2021** (A05), #6 in 2017
- **100%** of tested applications had some form of misconfiguration
- **Why the rise?**
  - Software becomes more and more *configurable*
  - Cloud, containers, Infrastructure as Code: configuration *is* code
- **XML External Entities** (XXE, CWE-611) is part of this category

CWEs mapped	16
Max incidence rate	27.70%
Avg incidence rate	3.00%
Max coverage	100%
Avg coverage	52.35%
Avg weighted exploit	7.96
Avg weighted impact	3.97
Total occurrences	719 084
Total CVEs	1 375

When is an Application Vulnerable? (OWASP)

- **Missing hardening** across the stack, or improperly configured permissions on cloud services
- **Unnecessary features** enabled or installed: ports, services, pages, accounts, testing frameworks, privileges
- **Default accounts** and their passwords still enabled and unchanged
- **Error handling** reveals stack traces or other overly informative messages
- **Latest security features** disabled or not configured securely after an upgrade
- **Backward compatibility** favoured over secure configuration
- **Insecure default values** in servers, frameworks (Struts, Spring, ASP.NET), libraries, databases
- **Security headers** not sent, or directives not set to secure values

Mapped CWEs

- **CWE-16** Configuration
- **CWE-15** External Control of Configuration Setting
- **CWE-260** Password in Configuration File
- **CWE-547** Hard-coded, Security-relevant Constants
- **CWE-526** Sensitive Info in Environment Variables
- **CWE-489** Active Debug Code
- **CWE-11** ASP.NET: Creating Debug Binary
- **CWE-13** ASP.NET: Password in Config File
- **CWE-611** XML External Entity Reference
- **CWE-776** Recursive Entity References in DTDs
- **CWE-614** Sensitive Cookie without `Secure`
- **CWE-1004** Sensitive Cookie without `HttpOnly`
- **CWE-315** Cleartext Sensitive Info in a Cookie
- **CWE-942** Permissive Cross-domain Policy
- **CWE-5** J2EE: Transmission Without Encryption
- **CWE-1174** ASP.NET: Improper Model Validation

Source: `https://top10.owasp.org/2025/A02_2025-Security_Misconfiguration/`

The OWASP Example Scenarios

1. **Sample applications** left on the production server, with known vulnerabilities. The admin console still uses the **default password**: the attacker logs in and takes over the server.
2. **Directory listing** is enabled. The attacker downloads the compiled Java classes, decompiles them and finds a severe access control flaw.
3. **Detailed error messages** with stack traces are returned to the user: component versions and internal details are disclosed.
4. A **cloud service** provider's default sharing permissions make the storage accessible from the Internet: sensitive data is read by anyone.

- None of them requires a bug in the application code
- All of them are found by **automated scanners within hours** of exposure

Why is Misconfiguration so Widespread?

- **Defaults are made for developers, not for production**
  - “It works out of the box”: no password, debug on, everything enabled
- **Huge configuration surface**
  - Hundreds of settings per component, thousands per cloud account
  - Nobody knows all of them; documentation is spread over many products
- **Environments diverge (configuration drift)**
  - A quick fix in production, a test bucket made public “for one hour”
- **Shared responsibility is misunderstood**
  - The cloud provider secures the cloud, *you* secure what you put in it
- **The Internet is scanned continuously**
  - Shodan, Censys, ZoomEye: a new open service is found in minutes

Key Concepts

Six Concepts to Understand Security Misconfiguration

1. **Hardening and attack surface reduction**
   - Unnecessary features, sample apps, admin consoles, default accounts
2. **Secrets in configuration**
   - Passwords and keys in files, repositories, environment variables
3. **Error handling and debug mode**
   - What a production system should never reveal – or execute
4. **Security directives for the browser**
   - HTTP security headers and cookie attributes
5. **Insecure parser defaults: XML External Entities**
   - A feature of the standard that becomes a vulnerability
6. **Cloud and infrastructure misconfiguration**
   - Storage, databases, containers exposed to the Internet

Concept 1: Hardening and Attack Surface

- **Attack surface**
  - All the points where an attacker can interact with the system: ports, URLs, services, accounts, file uploads, APIs
  - Each unnecessary element is a potential entry point – and must be patched, monitored, configured
- **Hardening = removing and locking down**
  - Minimal installation: no samples, no documentation, no test pages
  - Disable unused modules, HTTP methods, protocols (TLS 1.0, SMBv1)
  - Change or disable **default accounts**
  - Run services with a dedicated, unprivileged account
- **Reference baselines**
  - CIS Benchmarks, NIST SP 800-123, DISA STIGs, vendor hardening guides

Example: Admin Console with Default Credentials

- **An application server ships with a management console** (Tomcat Manager, JBoss/WildFly console, Jenkins, database admin pages)

```
<!-- conf/tomcat-users.xml : weak account added "just for testing" -->
<role rolename="manager-gui"/>
<user username="tomcat" password="tomcat" roles="manager-gui"/>
```

- **Why it is dangerous**
  - The console was opened to the Internet (recent Tomcat versions allow only localhost by default; old versions and images shipped weak accounts)
  - Default credential lists (`admin/admin`, `tomcat/tomcat`…) are tried automatically by botnets within minutes
  - A management console can deploy new applications ⇒ an authenticated attacker runs **arbitrary code** on the server
- **Fixes**
  - Remove the `manager` / `examples` webapps in production
  - Bind the console to localhost or a VPN; enforce strong, unique passwords

Example: Spring Boot Actuator Over-Exposed

- **Actuator = monitoring and management endpoints**

```
# application.yml -- "expose everything, it is handy while debugging"
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

```
GET /actuator/env        -> configuration properties, data source URLs
GET /actuator/mappings   -> every route of the application
GET /actuator/heapdump   -> full memory dump of the JVM
```

- A heap dump holds **whatever is in memory**: credentials, API keys, session tokens – often recoverable with standard analysis tools
- **Fix**: expose only `health`/`info`, put Actuator on a separate management port, require authentication

Concept 2: Secrets in Configuration

- **Secrets end up where they should not**
  - Hard-coded in source (CWE-547), in config files (CWE-260/13)
  - In environment variables dumped by error pages (CWE-526)
  - Committed to a Git repository – and kept forever in its history
- **Why this is a misconfiguration, not just “a bad habit”**
  - Repos get cloned, forked, leaked; CI logs are readable; Docker image layers keep deleted files
  - A single leaked key can open the whole cloud account
- **Good practice**
  - Secret manager (Vault, AWS Secrets Manager, Azure Key Vault)
  - Short-lived, federated credentials (OIDC) instead of static keys
  - Secret scanning in CI (gitleaks, trufflehog), pre-commit hooks

Example: A Key Committed to Git

```
$ git log --oneline
a1b2c3d  Remove AWS key from config   <- "fixed" in the latest commit
9f8e7d6  Add deployment config

$ git show 9f8e7d6:config/prod.env
AWS_ACCESS_KEY_ID=AKIA................
AWS_SECRET_ACCESS_KEY=.............................
```

- Deleting the key in a later commit does **not** remove it: it is still in the history, in forks, and in clones already made
- Automated bots scan public pushes to GitHub/GitLab in **seconds**
- **Response**
  - **Rotate the secret immediately** – assume it is already abused
  - Rewrite history (`git filter-repo`), then prefer a secret manager

Concept 3: Error Handling and Debug Mode

- **Development settings are dangerous in production**
  - Verbose stack traces reveal framework versions, file paths, SQL queries, internal IPs – a free map of the application (CWE-489, CWE-209)
- **Debug consoles can be far worse than disclosure**
  - Some frameworks ship an interactive debugger on the error page
  - If reachable in production, it can let an attacker *execute code*
- **Principle**
  - Users get a **generic** error page and a reference ID
  - Full details go to the server-side logs only
  - `DEBUG = False`, debug binaries disabled, test endpoints removed

Example: Debug Mode Left On

```
# settings.py (Django) -- committed as-is to production
DEBUG = True
ALLOWED_HOSTS = ["*"]
```

- On any error, the response page discloses
  - The full stack trace and source code snippets
  - Settings and environment variables: names containing `KEY`, `SECRET`, `PASS`… are masked, but e.g. a `DATABASE_URL` with embedded credentials is not
  - Installed app versions and the URL routing table
- **The error page becomes a reconnaissance tool**: the attacker learns exactly what to target next
- **Fix**
  - `DEBUG = False`, explicit `ALLOWED_HOSTS`, custom 404/500 pages, secrets read from the environment / a secret manager
  - Verify it in CI: fail the build if `DEBUG` is true for a prod target

Concept 4: Security Headers and Cookie Flags

- **The server tells the browser how to behave – or stays silent**

Header	Protects against
`Strict-Transport-Security`	protocol downgrade, SSL strip
`Content-Security-Policy`	XSS, data injection, clickjacking
`X-Content-Type-Options: nosniff`	MIME sniffing
`X-Frame-Options` / CSP `frame-ancestors`	clickjacking
`Referrer-Policy`	leaking URLs to third parties

- **Cookie attributes**
  - `Secure` (CWE-614): only sent over HTTPS
  - `HttpOnly` (CWE-1004): not readable from JavaScript
  - `SameSite`: mitigates CSRF
- Missing directives do not “break” anything – which is exactly why they are so often forgotten

Example: Hardening the Response

Insecure (defaults) – no headers, a cookie usable by scripts and over HTTP

```
HTTP/1.1 200 OK
Set-Cookie: session=8f3a2b...
```

Hardened – directives sent on every response

```
add_header Strict-Transport-Security "max-age=63072000" always;
add_header Content-Security-Policy "default-src 'self'" always;
add_header X-Content-Type-Options "nosniff" always;
server_tokens off;   # do not advertise the exact nginx version
```

```
Set-Cookie: session=8f3a2b...; Secure; HttpOnly; SameSite=Lax
```

- Verify with `securityheaders.com`, Mozilla Observatory, or in tests
- A strict CSP is the single most effective mitigation against XSS

Concept 5: XML External Entities (XXE)

- **XML can define entities** – including *external* ones that the parser resolves by reading a URL or a local file
- **The vulnerability is the default configuration**
  - Many XML parsers resolve external entities **by default** (CWE-611, CWE-776)
  - Any endpoint accepting XML (SOAP, SAML, SVG, DOCX/XLSX, RSS, config upload) can be affected
- **What an attacker can achieve**
  - Read local files (secrets, `/etc/passwd`, source code)
  - Server-Side Request Forgery towards internal services / cloud metadata
  - Denial of service via recursive entity expansion (“billion laughs”)

Example: XXE File Disclosure and the Fix

```
<?xml version="1.0"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/hostname"> ]>
<order><note>&xxe;</note></order>
```

- A vulnerable parser replaces `&xxe;` with the file's content, which is then reflected back in the response or stored
- Variants point at `http://169.254.169.254/...` (cloud metadata, SSRF) instead of a local file

Fix: disable DTDs / external entities in the parser

```
DocumentBuilderFactory f = DocumentBuilderFactory.newInstance();
f.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
f.setExpandEntityReferences(false);
```

- Prefer less complex data formats (JSON) when XML is not required

Concept 6: Cloud and Infrastructure

- **The cloud moved configuration to the front line**
  - A single wrong setting exposes data to the whole Internet
  - “Make it public” is often one click or one line of IaC away
- **Recurring patterns**
  - Object storage (S3, Azure Blob, GCS) readable/writable by anyone
  - Databases (Elasticsearch, MongoDB, Redis) bound to `0.0.0.0` with no authentication
  - Over-broad IAM roles; instance metadata reachable (links to SSRF/XXE)
  - Kubernetes dashboards or kubelet APIs exposed without auth
- **Shared responsibility**
  - The provider secures the infrastructure; the **configuration of your resources is on you**

Example: Public Object Storage

```
{ "Effect": "Allow",
  "Principal": "*",
  "Action": "s3:GetObject",
  "Resource": "arn:aws:s3:::acme-customer-exports/*" }
```

- The bucket name is guessable or found by enumeration tools; the objects are then downloadable by anyone, with no credentials
- Countless breaches follow this exact shape: backups, ID scans, medical and voter data exposed because a bucket was left public
- **Fixes**
  - Enable “Block Public Access” at the account level
  - Deny-by-default bucket policies; access via signed URLs or IAM roles
  - Continuous checks (IAM Access Analyzer, Prowler, ScoutSuite) in CI

Complex Attack Scenarios

Real-World Attacks

- **Misconfiguration is rarely exploited in isolation**
  - One exposed component gives a foothold; the next weak setting extends it
- **Scenarios studied**
  1. Exposed cloud metadata behind an SSRF-able component (Capital One, 2019)
  2. Unauthenticated databases exposed to the Internet (Meow / wave of wipes)
  3. CI/CD and build-server misconfiguration (Codecov, 2021)
  4. Exposed orchestrator consoles leading to cryptomining (Tesla, 2018)
  5. A chained scenario: info leak $\rightarrow$ secret $\rightarrow$ cloud takeover

Capital One (2019): Metadata + Over-Broad Role

- **The misconfigurations**
  - A perimeter component could be coerced into making internal requests (SSRF)
  - Instance metadata was reachable **without a token** (IMDSv1)
  - The attached IAM role had **far more permissions than needed**
- **The chain**
  - SSRF → read temporary credentials from the metadata endpoint → list and read storage buckets
- **Impact**: data of about **106 million** customers; about $270 million in penalties and settlements
- **Lessons (all configuration)**
  - Require IMDSv2 (token + hop limit), scope IAM roles to least privilege, segment the network, alert on unusual API calls

Internet-Exposed Databases

- **The misconfiguration**
  - Elasticsearch / MongoDB / Redis instances bound to a public interface with **authentication disabled by default** (older versions)
- **How they are found and abused**
  - Indexed by Shodan/Censys within hours of exposure
  - Read in bulk; often the data is deleted and replaced by a **ransom note**
  - The automated “Meow” campaign (2020) *wiped* about 4 000 open databases, leaving only empty indexes named `<random>-meow`
- **Why it keeps happening**
  - “It works” on a developer laptop; the same config reaches the cloud
  - No firewall / security group rule to restrict the source
- **Fixes**: bind to localhost / private subnet, enable auth and TLS, restrict with security groups, scan your own address space

Codecov (2021): The Build Pipeline as a Target

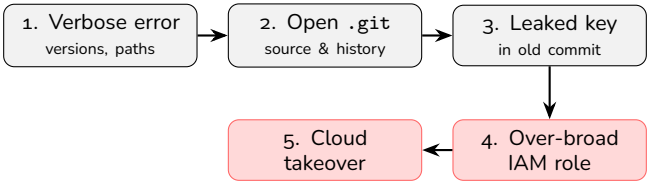
- **Context**: a widely used code-coverage tool, run inside customers' CI pipelines
- **Root cause**: a misconfiguration in how a Docker image was created let an attacker extract a credential and **modify the uploader script**
- **Effect**
  - The altered script ran in thousands of pipelines and could exfiltrate **environment variables** – i.e. the secrets of each build (tokens, cloud keys, signing keys)
  - A classic **supply-chain** amplification: one weak config, many downstream victims
- **Lessons**
  - CI is production: harden images, pin and verify dependencies and scripts (checksums), use short-lived CI secrets, least privilege in pipelines

Tesla (2018): Open Kubernetes Dashboard

- **The misconfiguration**
  - A Kubernetes administration dashboard was reachable **without authentication**
- **The chain**
  - Open dashboard → access to a pod holding **cloud access credentials** → reach cloud resources
  - Attackers deployed **cryptomining** and tried to stay stealthy (throttled CPU, hidden behind a CDN, no public mining pool)
- **Why it matters**
  - The same open-console pattern leaks data *and* gives compute; the attacker's goal here was resources, not information
- **Lessons**: authenticate every console, isolate credentials from workloads, network policies, monitor for anomalous workloads and egress

Kill Chain: From an Error Page to Cloud Takeover

A composite scenario built only from misconfigurations



- Each step is a **default or forgotten setting**, not a code bug
- Fixing *any one* of them breaks the chain – defense in depth

Kill Chain: The Steps

- **1. Verbose error** discloses the framework and an absolute path
- **2. Exposed** `.git` directory (directory listing / web root misconfiguration) – the whole repository is reconstructable:

```
GET /.git/config           -> remote URL, structure
# tooling reconstructs the full source tree and commit history
```

- **3. Secret in history**: an old commit still contains a cloud key that was “removed” later but never rotated
- **4. Over-broad role**: the key maps to an identity allowed to read storage, create users, or start compute
- **5. Takeover**: data exfiltration, persistence, or abuse of compute
- **Takeaway**: misconfigurations compound; severity is driven by the *weakest* setting plus the *broadest* privilege reachable from it

Prevention

How to Prevent Security Misconfiguration

- **Repeatable, automated hardening**
  - Same, scripted configuration across dev / QA / prod (only the secrets differ), so environments cannot drift
- **Minimal platform**: no unused features, components, samples or docs
- **Patch and review process**: configuration is reviewed like code; audit cloud permissions regularly
- **Segmentation**: containers, security groups, network policies, ACLs
- **Send security directives** to the client (headers, cookie flags)
- **Centralised error handling**: generic messages, details only in logs
- **Automated verification** of the configuration in every environment
- **No embedded static credentials**: identity federation, short-lived credentials, platform roles

Configuration as Code, Checked Automatically

- **Describe the infrastructure declaratively** (Terraform, CloudFormation, Kubernetes manifests) – then **scan the definition before deployment**

```
# A policy check (e.g. tfsec / Checkov / OPA) fails the build on:
#   - storage buckets without "block public access"
#   - security groups open to 0.0.0.0/0 on admin ports
#   - IAM policies using "Action: *" / "Resource: *"
#   - missing encryption at rest / in transit
```

- **Guardrails in the pipeline**
  - Secret scanning (gitleaks), container image scanning, SBOM
  - Fail on `DEBUG=true`, default passwords, missing security headers
- **Continuous posture checks** in the running environment
  - CSPM tools, CIS benchmark scans, external header/TLS scanners

Conclusion

Conclusion

- **Security misconfiguration is now #2** and affects **100%** of tested applications
- **The vulnerability is in the setup, not the code**
  - Defaults, forgotten settings, drift between environments
- **Every layer counts**
  - OS, server, framework, database, container, cloud, and the browser directives you send
- **Six recurring themes**
  - Reduce the attack surface (harden); keep secrets out of config; disable debug/verbose errors; send security headers; fix insecure parser defaults (XXE); lock down cloud and infrastructure

Conclusion (Cont.)

- **Exposure is discovered fast**
  - Internet-wide scanners and bots find open services and leaked keys in minutes – there is no “nobody will notice” window
- **Attacks chain misconfigurations**
  - Error leak → exposed source → leaked key → over-broad role → takeover; one correct setting breaks the chain
- **Make secure the default and the automatic**
  - Configuration as code, scanned in CI, identical across environments
  - Least privilege for users *and* for machines/roles
  - Verify continuously: posture scans, header checks, secret scanning
- **If a secret may have leaked, rotate it – assume compromise**

References

- OWASP Top 10:2025 – A02 Security Misconfiguration
  `https://top10.owasp.org/2025/A02_2025-Security_Misconfiguration/`
- OWASP Cheat Sheets – Configuration, XML External Entity Prevention, HTTP Security Response Headers
  `https://cheatsheetseries.owasp.org/`
- OWASP ASVS V14 – Configuration
- CIS Benchmarks; NIST SP 800-123 Guide to General Server Security
- PortSwigger Web Security Academy – XXE injection
  `https://portswigger.net/web-security/xxe`
- Capital One (2019): US DOJ indictment and OCC consent order
- Codecov (2021): bash uploader supply-chain compromise – vendor advisory
- RedLock / Tesla (2018): exposed Kubernetes dashboard, cryptojacking
- UpGuard / Shodan reports on exposed databases; “Meow” attacks (2020)