Code Hardening: Java and Interpreted Languages

CEN429 Secure Programming — Week 5

Asst. Prof. Dr. Uğur CORUH · 16.10.2026

Today's plan (3 hours)

Hour	Section	Topic
1	0–3	Basic concepts · managed languages · CERT Java · the root of injection
2	4–9	SQL/command/path injection · deserialisation · XML/XXE · Python/JS
3	10–16	Bytecode · ProGuard/R8 · string obfuscation · Android · SBOM · project

Short history — managed languages and the gaps that remain

- **1995** — **Java/JVM**: most memory bugs end at the **language level**
- **1998** — SQL injection is documented · **2003 OWASP Top 10**
- **2002** — **ProGuard** (later **R8**): because bytecode is easy to reverse
- **2015** deserialisation · **2020–21** SolarWinds and **Log4Shell** → the **SBOM** era

> The language solves **memory bugs**; it does **not solve injection or dependency risk**.

Where does this week fit?

- **Week 4:** C/C++ hardening (memory bugs).
- **This week (5):** Java/interpreted languages — most memory bugs are **solved**, but a new class appears: **injection** and **dependency**.
- The binary-protection side: **ProGuard/R8** and **SBOM**.

Learning outcome

This week is about **LO.3 / LO.4**.

By the end you will be able to:

- Recognise and prevent the injection class (SQL, command, path, XML, deserialisation)
- Obfuscate with ProGuard/R8 and write `-keep` rules
- Produce an **SBOM** and manage dependency risk

Main idea

> A managed language largely **solves** memory bugs; but **data mixing with commands** (injection) exists in every language.

Today we look at this common root and the language-specific defences.

0. Basic concepts (from scratch)

Why this section exists

This section **assumes no prior knowledge**.

- We define, from scratch, the terms we will use for the rest of the week.
- If you do not know a term, read **here** first.
- The sections that follow build **on top of** these terms.

What is a managed language?

- **Managed language:** a language that manages memory **automatically** (Java, C#, Python, JS).
- The programmer does not call `malloc` / `free`; the **garbage collector** (GC) handles it.
- → most memory bugs (overflow, UAF) **disappear**.

The JVM and bytecode

- **JVM (Java Virtual Machine):** the virtual machine that runs Java.
- **Bytecode:** the intermediate form Java source is compiled into (`.class` files).
- Bytecode is machine-independent; the JVM executes it.

The garbage collector (GC)

- **GC (Garbage Collector):** automatically cleans up objects that are no longer used.
- The programmer never frees memory by hand.
- This is why UAF/double free is **almost nonexistent** in Java.

What is injection?

- **Injection:** user **data** being interpreted as a **command**.
- Example: text entered as a username leaking into a SQL query as a command.
- The main theme of this week.

SQL and databases

- **SQL:** a database query language (`SELECT ... WHERE ...`).
- The application puts user input into the query.
- If put in wrong → **SQL injection**.

Parameterised query

- **Parameterised query (prepared statement):** the query skeleton is fixed; the data is supplied separately, as **parameters**.
- Data is never interpreted as a command.
- The **real** fix against SQL injection.

Serialisation

- **Serialisation:** converting an object to a byte sequence (to save or send it).
- **Deserialisation:** converting the bytes back into an object.
- Deserialising untrusted bytes is **dangerous** (code execution).

XML and XXE

- **XML:** a structured data format (built from tags).
- **XXE (XML External Entity):** abuse of XML's "external entity" feature → file reading, SSRF.
- External entities are **disabled** in the XML parser.

Path traversal

- **Path traversal:** escaping **outside** the allowed folder with `../`.
- Like `files/../../etc/passwd`.
- Prevented with canonicalisation + a root check.

ProGuard and R8

- **ProGuard / R8:** tools that **shrink** Java/Android bytecode and **obfuscate names**.
- R8 is Android's default tool.
- Name obfuscation + dead-code elimination + shrinking.

The `-keep` rule

- **`-keep`**: telling ProGuard/R8 "do not **change** this class's/method's name."
- Code called by name through reflection has to be kept.
- A wrong `-keep` → either a crash or weak obfuscation.

What is reflection?

- **Reflection:** finding and calling a class/method **by its name, at run time**.
- `Class.forName("...")`, `getMethod("...")`.
- Obfuscation can break this → `-keep` is required.

What is an SBOM?

- **SBOM (Software Bill of Materials):** the software's **bill of materials** — which libraries, which versions.
- When a vulnerability is disclosed in a library, it answers "are we affected?" **quickly**.
- Formats: CycloneDX, SPDX.

Dependency

- **Dependency:** the external libraries your project uses.
- Even if your own code is secure, a flaw in a dependency affects you.
- Supply-chain security.

Now we are ready

Terms:

managed language · JVM/bytecode · GC · injection · SQL/parameterised query · serialisation · XML/XXE · path traversal · ProGuard/R8 · `-keep` · reflection · SBOM · dependency

Now: what do managed languages solve, and what do they not solve?

1. Managed languages and the root of injection

What does a managed language solve?

- Automatic memory management (GC) → overflow, UAF, double free are **largely gone**.
- Array access is bounds-checked → `ArrayIndexOutOfBounds` (a crash, not exploitation).
- Type safety is stricter.

What does a managed language NOT solve?

- **Injection** (data = command) — exists in every language.
- **Logic errors**, authorisation errors.
- **Dependency** vulnerabilities.
- **Deserialisation** attacks.

> Memory bugs went down; the new front is injection and the supply chain.

Managed language: solves / does not solve — diagram

Yönetilen dil neyi çözer, neyi çözmez?

Java/Kotlin/C# çöp toplayıcı + sınır denetimi

ÇÖZER (dil düzeyinde)

tampon taşması
use-after-free
çift serbest bırakma
ilklendirilmemiş değişken

→ bellek güvenliği

ÇÖZMEZ

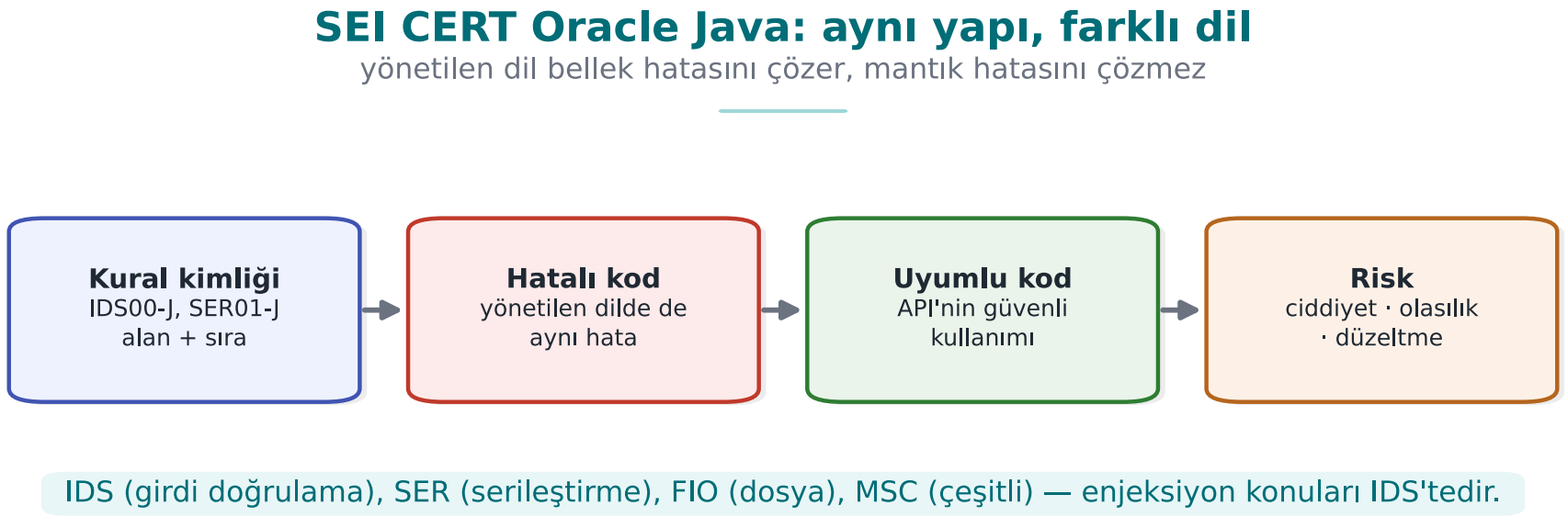
enjeksiyon (SQL, komut, XML)
güvensiz deserialization
koda gömülü sırlar
zafiyetli bağımlılıklar
mantık/yetki hataları

Dil bir hata SINIFINI kaldırır; kalan sınıflar için hâlâ siz sorumlusunuz.

SEI CERT Oracle Java

- A separate **rule book** for Java (CERT Oracle Coding Standard for Java).
- Same structure: noncompliant example + compliant solution + risk.
- Sample areas: injection, serialisation, concurrency, sensitive data.

CERT Oracle Java — diagram



IDS (girdi doğrulama), SER (serileştirme), FIO (dosya), MSC (çeşitli) — enjeksiyon konuları IDS'tedir.

SEI CERT · rules touching this week

Prefix	Category	Rule (example)
`IDS`	Input validation	IDS00-J SQL · IDS07-J `Runtime.exec` · IDS16/17-J XML
`FIO`	Input/output	FIO16-J: canonicalise path names
`SER`	Serialisation	SER12-J: do not deserialise untrusted data
`MSC`	Miscellaneous	MSC03-J: do not embed sensitive information in code
`ERR`	Error handling	ERR01-J: do not leak sensitive information in an exception

IDs end with the `-J` suffix.

Sensitive data in Java

- Java `String` is **immutable** → it cannot be erased from memory.
- For a password/key use `char[]`, and **fill** it (`Arrays.fill`) once you are done.
- The GC does **not guarantee** when it will collect.

The common root of injection

> Injection = **data** mixing with **command**.

- User data goes to an interpreter (SQL, shell, XML) as a **command**.
- The root is the same; the target (SQL/shell/XML) changes.

The root of injection — data or code? — diagram

Enjeksiyonun ortak kökü: veri ile komutun karışması

bütün enjeksiyon türleri aynı şemanın farklı yorumlayıcılardaki hâli

HATALI: tek kanal
"...WHERE x='" + girdi

Yorumlayıcı girdiyi
KOMUT olarak ayrıştırır

SQLi · komut enj. · XXE
şablon enjeksiyonu

DOĞRU: ayrı kanal
WHERE x = ? (parametre)

Komut SABİT kalır,
girdi yalnız VERİ

enjeksiyon ailesi
tümüyle kapanır

Kaçış (escaping) son çaredir; asıl çözüm komutu ve veriyi ayrı kanala koymaktır.

The common root of the fix

The same principle in every injection type:

- **Separate data from code** (a parameterised API).
- Never embed data into a command via **string concatenation**.
- Allow list + context-appropriate escaping (last resort).

One channel or two channels? — table

Interpreter	One channel (wrong)	Two channels (correct)
SQL engine	`"...WHERE name='" + name + "'"`	`PreparedStatement` + `?`
Shell	`exec("sh -c '...' " + host)`	`ProcessBuilder(...)` — no shell
File system	`new File(root, request)`	Canonicalise + root check
XML	Building XML by string concatenation	Building elements with a DOM/StAX API

Same principle, four different interpreters.

Injection families (today)

- SQL injection
- Command injection
- Path traversal
- Deserialisation
- XML/XXE, templates, XSS

All share the **same** root and the **same** fix logic.

Rule of this section

> A managed language solves the **memory** bug; it does not solve **injection**.

- The root of injection is always the same: data = command.
- The fix is always the same: **separate** data from code.
- We will now see this principle applied to each injection type in turn.

Section 1 — quick check

1. Which error class does a managed language largely solve?
2. What does it not solve?
3. What is the common root of injection?

Section 1 — answers

1. **Memory-safety** bugs (buffer overflow, use-after-free, double/dangling pointer) — through the GC + bounds checking.
2. **Logic/injection/authorisation** bugs, unsafe deserialisation, weak cryptography, **dependency** flaws — these are above the language, unsolved.
3. Untrusted **DATA** mixing with **CODE/command** in the same channel; the interpreter parses the data as a **command**.

2. SQL injection

Buggy code

```
String q = "SELECT * FROM kul WHERE ad = '" + ad + "'";
stmt.executeQuery(q);   // HATALI: ad komuta karışır
```

What if `ad` contains a quote and SQL?

Attack · example

The user enters this as `ad`:

```
' OR '1'='1
```

The query becomes:

```
SELECT * FROM kul WHERE ad = '' OR '1'='1'
```

`'1'='1'` is always true → **all rows** are returned.

Why did it happen?

- `ad` was concatenated into the query **text**.
- The quote closed the data and moved into the **command** part.
- Data and command got **mixed**.

SQL injection step by step — diagram

Correct · parameterised query

```
PreparedStatement ps =
    con.prepareStatement("SELECT * FROM kul WHERE ad = ?");
ps.setString(1, ad);    // ad yalnız VERİ
ps.executeQuery();
```

`?` is a **parameter**; `ad` is never interpreted as a command.

Why does it work?

- The query skeleton is sent (compiled) to the database **beforehand**.
- `ad` is bound afterwards as **data**.
- Even if it contains a quote, it stays data.

Other layers

- **Least privilege:** the application's DB user should do only what is needed.
- **Input validation:** an allow list, regardless.
- If you use an **ORM**, use its parameterised API too (not string concatenation).

⚠️ Escaping is the last resort

- Manually escaping quotes is **error-prone**; every DB is different.
- The **always**-first priority is a parameterised query.
- Escaping only where parameters are not possible, and carefully.

SQL injection · rule of this section

- **Never** build a query with string concatenation.
- **Always** use a parameterised query (`PreparedStatement`, `?`).
- Use an **allow list** for identifiers such as column/table names.
- Escaping only as a last resort.

3. Command injection

Shell or no shell — diagram

Buggy code

```
Runtime.getRuntime().exec("ping " + host);  // HATALI
```

What if `host` contains a shell metacharacter?

Attack · example

As `host`:

```
example.com; rm -rf veri
```

The shell runs this as **two commands**: `ping` **and** `rm`.

(The example is synthetic; never run it.)

Why did it happen?

- The command was handed to a **shell** as text.
- `;` , `|` , `&&` are **command separators** in the shell.
- Data got mixed into the command line.

Correct · argument array

```
new ProcessBuilder("ping", "-c", "1", host).start();
```

- **No shell**; `host` is a single **argument**.
- Metacharacters are not interpreted.

Additional rules

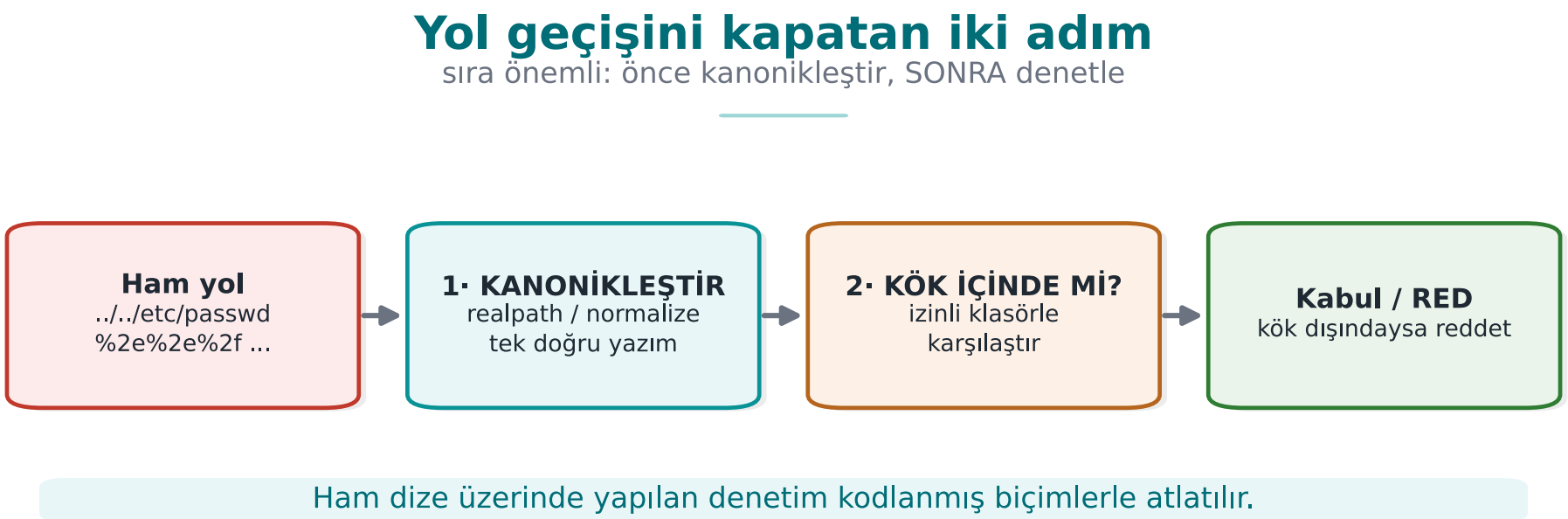
- Where possible, do **not** run an external command at all; use a library API.
- If you must: a fixed path, an argument array, an allow list.
- Similar to `ENV33-C`: calling `system()`/a shell.

Command injection · rule of this section

- Where possible, **never** run an external command.
- If you must: a shell-less **argument array** (`ProcessBuilder`).
- A fixed path + allow list; watch input starting with `-` (argument injection).

4. Path traversal

Two steps that close off path traversal — diagram



Buggy code

```
File f = new File("yuklenen/" + adi);  // HATALI
```

What if `adi` contains `../`?

Attack · example

As `adi`:

```
../../etc/passwd
```

Result: `yuklenen/../../etc/passwd` → **outside** the allowed folder.

Why did it happen?

- `adi` was appended **directly** into the file path.
- The `../` sequence was read by the interpreter (the file system) as "go up a folder."
- There was no check: whether the folder was left was never even asked.

Correct · canonicalise + root check

```
Path kok = Paths.get("yuklenen").toRealPath();
Path hedef = kok.resolve(adi).normalize().toRealPath();
if (!hedef.startsWith(kok)) throw new SecurityException();
```

Is the canonical path **inside the root**? If not, reject.

Why does it work?

- First the path is turned **canonical** (single, definite).
- Then we check whether the canonical path is **inside** the root.
- `../`, an encoded form (`..%2f`), or a symbolic link — all of them land on the same real path **after** canonicalisation; none of them can escape the check.

Path traversal · extra

- Restrict the file name to an **allow list** (`[a-zA-Z0-9_.-]`).
- Symbolic links (`symlink`) are resolved by canonicalisation.
- Do not make a user name **directly** into a file name.

Path traversal · rule of this section

- **Never** trust the raw string (`contains("..")` is not enough).
- First **canonicalise**, then check whether it is **inside the root**.
- Resolve symbolic links too (`toRealPath()`); repeat the check after the file is opened as well.

Three injections · common lesson

Type	Wrong	Correct
SQL	String concatenation	Parameterised query
Command	Text to the shell	Argument array
Path	Concatenating `../`	Canonicalise + root check

Same root: **separate data from code/command**.

Sections 2–4 — quick check

1. Why does `' OR '1'='1` return all rows?
2. Why is a `ProcessBuilder` argument array safe?
3. Which two steps close off path traversal?

Sections 2–4 — answers

1. Once joined into the string it makes the `WHERE` condition **always true** → all rows (data was parsed as code).
2. Arguments pass **to the program, not the shell**, one by one; there is no metacharacter interpretation (`;` `|`) → command injection is closed off.
3. (1) Convert to a **canonical path** (`realpath` /normalise), (2) verify it is inside the allowed **root directory** (an allow list). Never put user input directly into a path.

5. Unsafe deserialisation

Buggy code

```
Object o = new ObjectInputStream(giris).readObject();  // TEHLİKELİ
```

What if `giris` comes from an untrusted source?

Deserialisation gadget chain — diagram

The problem

- Turning untrusted bytes into an object is **dangerous**.
- In Java, `ObjectInputStream.readObject()` can run **code** while unpacking (gadget chains).
- Result: remote code execution.

Why is this so bad?

- Objects' **constructors/methods** run during unpacking.
- If suitable libraries (gadgets) are present, they can be chained to run commands.
- Input that looks like "just data" behaves like **code**.

Correct · best: do not do it at all

- Do not deserialise untrusted data with **native serialisation**.
- Use a **JSON/data** format + a strict schema instead.
- Read only the fields you expect.

If you must · filter

```
ObjectInputFilter f = ObjectInputFilter.Config
    .createFilter("com.uygulama.*;!*");  // yalnız izinli sınıflar
ois.setObjectInputFilter(f);
```

- **Allow list**: only the expected classes.
- Also set a depth/size limit (against DoS).

Deserialisation · rule of this section

- Deserialising an untrusted byte stream hands the attacker the power to **choose a class**.
- Best: **do not do it at all** (JSON + a strict schema).
- If you must: an **allow-list filter**, a depth/size limit.

6. XML, templates, XSS

XXE · the problem

- XML lets you define an **external entity**; the parser **resolves** it.
- An attacker can use this to read files or make the server issue a request (SSRF).

XXE — diagram

Kural: XML/YAML/şablon ayrıştırıcılarında gereksiz yetenekleri KAPATIN.

XXE · correct

```
dbf.setFeature(
  "http://apache.org/xml/features/disallow-doctype-decl", true);
// ya da dış varlıkları kapat
```

- **Disable external entities**, reject DOCTYPE.
- A one-line setting in most libraries.

Template injection (SSTI)

- Handing user input to a template engine **as the template itself**.
- The engine can run it like code.
- Rule: user input is **data**, the template is **fixed**.

XSS (briefly)

- **XSS:** user data leaking into a web page as **HTML/JS**.
- Fix: **context-appropriate output encoding** (HTML-encode), CSP.
- Same root again: data ≠ code.

XML/template/XSS · rule of this section

- Know **which feature of the interpreter is turned on**, and disable what is not needed.
- XML: external entities + DOCTYPE disabled.
- Templates: user data is only a **variable**, never the template itself.
- HTML: encode output **context-appropriately**.

7. The same bugs in Python and JavaScript

The same bugs in interpreted languages — diagram

Same root, different language

- Injection exists in every language: Python `os.system`, JS `child_process`.
- The fix is the same: an argument array, a parameterised query.

Python · examples

- `os.system("cmd " + x)` → command injection; use `subprocess.run([...])`.
- `eval(input)` / `pickle.loads(untrusted)` → **never** with untrusted data.
- SQL: parameterised (`cursor.execute("... ?", (x,))`).

Python · `eval` example (code)

```
deger = eval(girdi)                 # HATALI: herhangi bir Python ifadesi

import ast
deger = ast.literal_eval(girdi)     # Daha iyi: yalnız sabitler
deger = int(girdi)                  # En iyisi: beklenen türü ayrıştır
```

Rule: user data must **never** reach `eval`.

ReDoS (regular expression DoS)

- A bad **regex** can spend **exponential** time on certain input.
- An attacker can use this to lock up the CPU.
- Fix: a safe regex, an input-length limit, a timeout.

ReDoS · a concrete example

```
Desen:  ^(a+)+$
Girdi:  "aaaaaaaaaaaaaaaaaaaaaaaaaaaa!"
```

- The engine tries **millions** of paths to prove there is no match.
- A single request locks the CPU for seconds/minutes (CWE-1333).

JavaScript · prototype pollution

- **Prototype pollution:** an attacker corrupts a property **shared** by all objects, through `__proto__`.
- Unexpected behaviour, authorisation bypass.
- Fix: safe merging, `Object.create(null)`, up-to-date libraries.

Interpreted languages · common rule

- `eval`, `exec`, `pickle`, dynamic `require` → do **not** use with untrusted data.
- Input validation + a parameterised API + up-to-date dependencies.

Sections 5–9 — quick check

1. Why can unsafe deserialisation run code?
2. Which setting disables XXE?
3. What is ReDoS, and how is it mitigated?

Sections 5–9 — answers

1. While an object is being constructed, classes' side-effecting methods (`readObject` /**gadget chain**) fire → an attacker builds a flow using the object graph.
2. The parser setting that disables **external entities and DOCTYPE** (`disallow-doctype-decl` , external entities off).
3. **ReDoS:** a catastrophic-backtracking regex; bad input → **exponential** time → the CPU locks up. Mitigation: a linear-time engine (RE2), avoid nested quantifiers, a length/time limit.

8. Bytecode and decompilation

Bytecode vs machine code — diagram

Bayt kod neden makine kodundan kolay okunur?

bu yüzden Java'da gizleme (ProGuard/R8) daha kritik

Java bayt kodu
sınıf/metot/alan ADLARI durur
tipler durur
yapı korunur

decompiler ≈ kaynağı verir

C/C++ makine kodu
adlar atılır
tipler kaybolur
optimizasyon yapıyı bozar

disassembler ≈ ham komut

Gizleme hatayı düzeltmez; yalnız 'kolayca okunma' avantajını geri alır.

Java bytecode is easy to read

- `.class` / `.jar` bytecode decompiles back **very close** to the source.
- Tools: `javap` (disassembly), JD-GUI/CFR/Procyon (decompilation).
- Names, strings, structure are largely **visible**.

`javap` example

```
javap -c -p Uygulama.class    # bayt kodunu sök
```

- Method names, calls, constants are visible.
- This is exactly why obfuscation is **even more** necessary in Java.

`javap` output — what is visible?

```
private static final java.lang.String GECERLI_PIN;
public boolean pinDogru(java.lang.String);
  Code:
     0: aload_1
     1: ldc           #7    // String 4729
     3: invokevirtual #9    // String.equals
     6: ireturn
```

There is no source code, but the **constant** (`4729`), the **names** (`GECERLI_PIN` , `pinDogru`) and the **logic** ("compare the input against this constant") are plainly visible.

Why is it easier than C?

- Bytecode is **high-level** and carries type information.
- Variable/method names are mostly preserved.
- → reverse engineering is cheap; obfuscation is essential.

⚠️ There are no secrets on the client

- Obfuscation **makes this harder**, but it does not really **hide** the secret.
- A key/password that does not have to live on the client → keep it **on the server**.
- A secret that **has to** live on the client needs whitebox cryptography (Week 11).

9. ProGuard and R8

R8 and the measurement steps — diagram

mapping.txt kaybolursa üretimdeki çökme raporlarını bir daha çözemezsiniz.

What do ProGuard/R8 do?

Three jobs:

- **Shrinking:** drops unused code.
- **Optimisation:** simplifies the code.
- **Obfuscation:** makes names meaningless (`a` , `b` , `c`).

ProGuard/R8 · four stages — table

Stage	What it does	Security contribution
Shrinking	Drops unused code	Attack surface shrinks
Optimisation	Inlining, constant folding	Structure moves away from the source
Obfuscation	Turns names into `a`, `b`, `c`	Meaningful names disappear
Preverification	Verification info for older JVMs	—

Name obfuscation example

```
LisansDenetleyici.dogrula()  →  a.b()
```

- Meaningful names disappear.
- The map gets harder for a reverse engineer.
- Removing dead code shrinks the binary.

Why is `-keep` needed?

- Code called **by name** through reflection (an API, entry points, JNI) must be kept.
- If obfuscation changes the name, the call **breaks**.

```
-keep class com.uygulama.Api { public *; }
```

The `-keep` balance — diagram

The `-keep` balance

- **Too little `-keep`**: the application crashes (reflection breaks).
- **Too much `-keep`**: obfuscation is weakened (everything stays exposed).
- Keep **only the entry points that are genuinely required**.

The mapping file

- R8 produces a `mapping.txt`: obfuscated name → real name.
- It is **kept secret** (not shipped), but it is retained to decode crash traces.
- Archived together with the version identifier.

Mapping file · example (`mapping.txt`)

```
com.ornek.odeme.KartYoneticisi -> com.ornek.a:
    android.content.Context baglam -> a
    boolean varsayilanKartiAyarla(java.lang.String) -> b
```

Old name → new name. This file **completely** undoes the obfuscation.

ProGuard/R8 · rule of this section

- Shrink + optimise + obfuscate; it does **not obfuscate strings**.
- `-keep` only for what is **genuinely** called through reflection/JNI.
- `mapping.txt` is a **secret**: keep it, do not distribute it.

10. String obfuscation and reflection

String obfuscation and dynamic invocation — diagram

Bu bir şifreleme değil, bir GECİKTİRMEdir: anahtar da ikilidedir, bulunabilir.

Strings are exposed in Java

- Constant strings inside `.class` appear **in the clear** (`strings`, a decompiler).
- Sensitive URLs, key fragments, messages leak.

String obfuscation

- Strings are **encoded** at build time, decoded at use.
- Same rule as Week 4: it stops `strings`, it does **not keep a secret**.
- The decoding code is also in the bytecode → limited protection.

⚠️ Reflection conflicts with obfuscation

- A call by name (`getMethod("dogrula")`) breaks **after** obfuscation.
- Fix: either `-keep`, or drop reflection and call directly.
- Reducing reflection is good for both security and obfuscation.

11. Android R8 and measurement

R8 on Android

- R8 is the **default** shrinker/obfuscator of the Android build.
- `minifyEnabled true` + `proguard-rules.pro`.
- Produces DEX (Android bytecode).

Turning R8 on for the release build

```
android { buildTypes { release {
    minifyEnabled true
    shrinkResources true
    proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                  'proguard-rules.pro'
}}}
```

The debug build is **not obfuscated**; verify that the shipped build is the **release** one.

Measuring the effect of obfuscation

- Before/after obfuscation:
  - the ratio of meaningful names
  - the count of plain-text sensitive strings
  - APK size
- Measure it and write it into S9 (the measurement logic from Week 9).

Measurement table — extended

Metric	Expected change
Number of classes/methods	Decreases with shrinking
Meaningful-name ratio	Drops sharply with obfuscation
Plain-text sensitive strings	Goes to zero with string obfuscation
Package size	Decreases with shrinking
Number of log calls	Zero in the release
Time to locate a check	Increases (human trial)

Static analysis for Java

- SpotBugs, Error Prone, SonarQube.
- Checks close to the CERT Java rules.
- On every merge, in CI.

Sections 8–13 — quick check

1. Why is Java bytecode easier to read than C?
2. What is the `-keep` balance?
3. Why does reflection conflict with obfuscation?

Sections 8–13 — answers

1. Bytecode is **high-level**; type and **name metadata** (class/field/method) are preserved → a decompiler produces near-source output. C compiles to machine code, discarding names.
2. **The `-keep` balance:** too much keep → obfuscation is weak; too little keep → reflection/serialisation breaks. Keep only the **external entry points**.
3. Reflection calls **by string** name; once the obfuscator renames it, the string no longer matches → a run-time error. Those names need to be kept (a gap in the obfuscation).

12. Dependency security and SBOM

The problem · supply chain

- Even if your own code is flawless, a flaw in a **library** you use affects you.
- A modern application contains hundreds of dependencies.
- "Which version do I have?" is a critical question.

Real examples

- Log4Shell (Log4j): a single library flaw affected millions of systems.
- A malicious package (typosquatting): a fake package with a similar name.
- An unmaintained/abandoned dependency.

Log4Shell · a concrete example (December 2021)

- A specific expression logged by Log4j 2 loaded and ran a class from a **remote** address.
- CVE-2021-44228, CVSS **10.0** (the highest score).
- The real crisis was not applying the patch: it was the question of **which system had which version**.

What is an SBOM? (recap)

- The software's **bill of materials**: which component, which version, which licence.
- When a flaw is announced, "am I affected?" is answered **quickly**.

SBOM formats

- **CycloneDX** (OWASP) — security-focused, widely used.
- **SPDX** (Linux Foundation) — strong on licensing too.
- Generated automatically by tools (during the build step).

A CycloneDX example (JSON, synthetic)

```
{
  "bomFormat": "CycloneDX",
  "components": [{
    "type": "library", "name": "ornek-json", "version": "1.2.0",
    "purl": "pkg:maven/com.ornek/ornek-json@1.2.0"
  }]
}
```

Every component: name, version, **purl** (an ecosystem-independent, uniform identifier), a hash value.

What is VEX?

- **VEX (Vulnerability Exploitability eXchange):** shares "I have this flaw, but it is **not exploitable**" information.
- Reduces noise: not every CVE is a panic.
- Accompanies the SBOM.

Dependency management rules

- **Pin** versions; do not pull "latest" at random.
- Scan for known vulnerabilities (`dependency-check`, `npm audit`, `pip-audit`).
- A safe update process; apply critical patches fast.

SBOM generation · example flow

Each release's SBOM is stored with its release identifier.

Link to the project

- **S13:** a secure development process + SBOM.
- An up-to-date SBOM and dependency scan for every release.
- Handed-off requirements: "safe updates" belong to the upstream application (Week 13).

SBOM · rule of this section

- Even if your own code is flawless, **your dependency's flaw affects you**.
- SBOM = the bill of materials; VEX = the answer to "am I genuinely affected?"
- **Pin** the version, **scan** continuously, apply the critical patch fast.

Section 14 — quick check

1. What is an SBOM good for?
2. Why does VEX reduce noise?
3. Why do we pin dependency versions?

Section 14 — answers

1. **SBOM** = a component/dependency inventory; when a new flaw appears it answers "do I have it?" **quickly**.
2. **VEX** states whether the flaw is genuinely **exploitable** in the product; unaffected ones are marked "not affected" → false alarms drop.
3. A pinned version = a **reproducible** build + supply-chain security; it blocks an unexpected/malicious update.

Project and closing

Project · S9/S13 (the Java side)

- [ ] Turn injection-prone spots into parameterised APIs (SQL, command, path).
- [ ] A filter/schema if there is unsafe deserialisation.
- [ ] Obfuscate with R8/ProGuard; `-keep` only the entry points that are needed.
- [ ] String obfuscation; no sensitive string in `strings` /a decompiler.
- [ ] Produce an SBOM + a dependency scan (S13).

Solved self-check

Question 1

Which error class does a managed language solve, and which does it not?

Answer: It largely solves memory bugs (overflow, UAF). It **does not** solve injection, logic/authorisation errors, dependency, and deserialisation attacks.

Question 2

Why does `' OR '1'='1` return all rows?

Answer: The input enters the query through string concatenation; the quote closes the data, and `OR '1'='1'` is always true. A parameterised query prevents this.

Question 3

How does `ProcessBuilder` prevent command injection?

Answer: It supplies arguments as an **array** without using a shell; metacharacters (`;`, `|`) are not interpreted, and the input stays a single argument.

Question 4

Which two steps close off path traversal?

Answer: Canonicalisation (`normalize` / `toRealPath`) + a root check (`startsWith(kok)`). Plus a file-name allow list.

Question 5

Why is unsafe deserialisation dangerous? The fix?

Answer: Object methods run while unpacking; code can be run through a gadget chain. Fix: a data format + schema instead of native serialisation; if you must, an `ObjectInputFilter` allow list.

Question 6

What is XXE, and which setting disables it?

Answer: Abuse of XML external entities (file reading/SSRF). The parser setting that disables external entities and DOCTYPE.

Question 7

What are ReDoS and prototype pollution?

Answer: ReDoS: a bad regex spending exponential time (DoS). Prototype pollution: corrupting all objects through `__proto__` in JS. Both need input validation + a safe API.

Question 8

Why does Java bytecode need obfuscation more than most?

Answer: Bytecode is high-level and carries type and name information; it decompiles very close to the source. Names/strings appear in the clear.

Question 9

What is the `-keep` balance?

Answer: Too little `-keep` → reflection breaks, the application crashes. Too much → obfuscation is weakened. Only the entry points genuinely called through reflection are kept.

Question 10

Does string obfuscation keep a key secret in Java?

Answer: No. It stops `strings` /static scanning, but the decoded string is in the clear in memory, and the decoding code is in the bytecode. A key needs different protection.

Question 11

What are SBOM and VEX for?

Answer: An SBOM is the component list; when a flaw is announced it tells you quickly whether you are affected. VEX states whether the flaw is exploitable, reducing noise.

Question 12

What is the common root and fix of every injection type?

Answer: Root: data mixing with command. Fix: **separating** data from code (a parameterised/argument-array API), not building a command with string concatenation.

Summary: this week in one sentence

> A managed language solves memory bugs, but **injection exists in every language**; the fix is always separating data from the command,
> and on the protection side it is obfuscation (R8) and **dependency/SBOM** management.

Next week

Week 6 — RASP and run-time protection

Tamper/debugger detection, integrity checking, response policy and layered defence.

Appendix A · An injection from start to finish

Context · a login form

An application that logs a user in with a username and password.

```
String q = "SELECT * FROM kul WHERE ad='" + ad +
           "' AND parola='" + p + "'";
```

Where is the mistake?

Step 1 · the attacker's input

Into the `ad` field:

```
yonetici' --
```

`--` starts a **comment** in SQL; everything after it is ignored.

Step 2 · what does the query become?

```
SELECT * FROM kul WHERE ad='yonetici' -- ' AND parola='...'
```

The password check stayed **inside the comment** → an `yonetici` login with no password.

Step 3 · impact

- Authentication was **bypassed**.
- Further: reading other tables with `UNION SELECT`.
- `; DROP TABLE` (if stacked queries are supported) — data loss.

Step 4 · the fix

```
PreparedStatement ps = con.prepareStatement(
  "SELECT * FROM kul WHERE ad=? AND parola_ozet=?");
ps.setString(1, ad);
ps.setString(2, ozetle(p));
```

`ad` and the password are now **data**; `--` has no effect.

Step 5 · layered defence

- **Never store a password in plain text:** a hash + salt (Week 3).
- **Least privilege:** the DB user only has `SELECT`.
- **Input validation:** an allow list for the username.
- **Logging/monitoring:** failed login attempts.

Lesson

- A single root: data got mixed with the command.
- The real fix: a parameterised query.
- But defence in depth at every layer.

Appendix B · A deserialisation mini case study

Scenario

An application deserialises a "session object" sent by the user using native serialisation:

```
Object o = new ObjectInputStream(giris).readObject();  // TEHLİKELİ
```

What could happen?

- The attacker prepares a **gadget chain** from libraries found on the classpath.
- The chain runs while unpacking → command execution.
- The input was thought to be "data," but it behaved like code.

Fix 1 · change the format

- **JSON** + a strict schema instead of native serialisation.
- Only the expected fields are read.
- The object graph never "comes alive."

Fix 2 · if you must, filter

```
ois.setObjectInputFilter(ObjectInputFilter.Config
  .createFilter("com.uygulama.Oturum;!*"));  // yalnız bu sınıf
```

- An allow list + a depth/size limit.

Case study · lesson

- Turning untrusted data into a "live object" is risky.
- Safest: never do it (use a data format).
- If you must: a strict allow list.

Appendix C · Injection, language by language

The same bug, three languages

Danger	Java	Python	JS
Command	`Runtime.exec(str)`	`os.system(str)`	`exec(str)`
Safe	`ProcessBuilder([...])`	`subprocess.run([...])`	`execFile([...])`

The same bug, extracting data

Danger	Java	Python	JS
Code execution	`readObject`	`pickle.loads` , `eval`	`eval` , `Function`
Safe	JSON + schema	`json.loads`	`JSON.parse`

The root is always the same: data ≠ code.

Three languages · one rule

- A dynamic "run" (`eval`, `exec`, `pickle`, `readObject`) → **never** with untrusted data.
- Command → an argument array.
- SQL → parameterised.
- Input → an allow list + rejection.

Appendices A–C · summary

- Injection is not tied to a language; it is tied to a **pattern**.
- Recognise the pattern: "is user data going to an interpreter as a command?"
- Once recognised, the fix is ready: separate.

Appendix D · R8 obfuscation, step by step

Step 1 · turn it on

`build.gradle`:

```
buildTypes { release {
    minifyEnabled true
    proguardFiles(..., 'proguard-rules.pro')
}}
```

Shrinking + obfuscation are turned on for the release build.

Step 2 · protect the entry points

```
-keep class com.uygulama.PublicApi { public *; }
-keepclassmembers class * {
    @com.uygulama.Reflected *;
}
```

Keep only what is called through reflection/JNI.

Step 3 · build and verify

- Decompile the APK; see that names are `a`, `b`.
- **Keep** `mapping.txt` (do not distribute it).
- **Test** flows that use reflection (no crashes?).

Step 4 · measure

- Meaningful-name ratio: before 90% → after 10%.
- Plain sensitive-string count: before N → after 0.
- APK size: did it shrink?

Write the measurement into S9.

Step 5 · add string obfuscation

- Encode sensitive strings, decode at use, then wipe immediately.
- Must not appear in `strings` /a decompiler.
- Not a key; only a static-scanning deterrent.

Appendix E · SBOM, step by step

Step 1 · generate

```
# örnek: CycloneDX eklentisi
mvn org.cyclonedx:cyclonedx-maven-plugin:makeBom
```

A `bom.json` / `bom.xml` is produced during the build.

Step 2 · scan

```
dependency-check --scan . --format HTML
```

Maps known vulnerabilities (CVEs) to dependencies.

Step 3 · assess (VEX)

- For every flaw: do I **have** it? Is it **exploitable**?
- If not exploitable, mark it with VEX (noise drops).
- If exploitable: update or mitigate.

Step 4 · archive

- The SBOM + scan report are stored **with the release identifier**.
- When a new flaw appears: "which release had this dependency?" is answered fast.

Step 5 · process

- Pin versions; no automatic "latest" pulling.
- Apply critical patches fast.
- If you cannot handle safe updates yourself → **hand it off** (Week 13).

Appendix F · Quick reference

Common mistakes

- Building SQL/commands with string concatenation.
- `eval` / `pickle` / `readObject` with untrusted data.
- Leaving external entities open in XML.
- Protecting everything with `-keep *` (obfuscation wasted).
- Not scanning dependencies.

Glossary

Term	Meaning
Injection	Data = command mixing
Parameterised query	Data is bound separately
Deserialisation	Bytes → object (risky)
XXE	Abuse of an XML external entity
R8/ProGuard	Shrinking + name obfuscation
SBOM/VEX	Component list / exploitability

Checklist

- [ ] SQL parameterised, command as an argument array
- [ ] Path canonical + root check
- [ ] Deserialisation filtered/schema'd
- [ ] XXE disabled
- [ ] R8 on, `-keep` minimal, measured
- [ ] SBOM produced, dependency scan clean

Final word (Week 5)

> Even though the language solves memory bugs, **injection and the supply chain remain your responsibility**.

Separate data from code; obfuscate the binary with R8; manage dependencies with an SBOM.