# Lesson 8: Server-side Code Injection

## Introduction

While client-side vulnerabilities like XSS primarily affect individual users, server-side code injection vulnerabilities can compromise entire applications and their data. These vulnerabilities occur when an application takes user input and combines it with code that will be executed by the server, without properly separating data from code.

Server-side code injection is particularly dangerous because attackers can:

* Read or modify any data in the application’s database
* Execute arbitrary commands on the server
* Access internal network resources
* Move laterally to attack other systems
* Steal or destroy sensitive data

This lesson covers two major types of server-side code injection: subprocess command injection and SQL injection.

## Subprocess Command Injection

Command injection occurs when an application passes unsafe user input to a system shell. Since the shell can run any command on the system, this essentially gives attackers complete control over the server.

### Example: A Vulnerable File Viewer

Consider this simple Node.js application that lets users view files stored on a server:

const app = express()  
  
app.get('/', (req, res) => {  
 res.send(`  
 <h1>File viewer</h1>  
 <form method='GET' action='/view'>  
 <input name='filename' />  
 <input type='submit' value='Submit' />  
 </form>  
 `)  
})  
  
app.get('/view', (req, res) => {  
 const { filename } = req.query  
 const stdout = childProcess.execSync(`cat ${filename}`)  
 res.send(stdout.toString())  
})

This code takes a filename from the user and uses the Unix cat command to display its contents. The vulnerability is in combining user input directly with shell commands. In Unix shells, the semicolon (;) separates different commands, so attackers can inject additional commands after the intended one. If an attacker submitted this input:

file.txt; rm -rf /

It would result in the following shell command being executed:

cat file.txt; rm -rf /

### Safer Alternatives

Instead of building shell commands using string concatenation, Node.js provides APIs that safely handle argument passing:

// Unsafe - string concatenation  
const stdout = childProcess.execSync(`cat ${filename}`)  
  
// Safe - argument array  
const child = childProcess.spawnSync('cat', [filename])  
if (child.status !== 0) {  
 res.send(child.stderr.toString())  
} else {  
 res.send(child.stdout.toString())  
}

The spawnSync function takes the command and its arguments separately, ensuring proper passing of arguments, and it invokes the subprocess directly without an intermediate shell, eliminating the risk that arguments may be parsed in a surprising way.

*Note:* spawn should be used instead of spawnSync to avoid event loop blocking, but spawnSync is easier to demonstrate concisely in this context.

## SQL Injection

Similarly, SQL injection occurs when user input is incorporated into database queries without proper escaping. This allows attackers to modify or extend the intended query, potentially accessing or modifying any data in the database.

### Basic SQL Injection

Consider this code for handling user authentication:

app.post('/login', (req, res) => {  
 const { username, password } = req.body  
 const query = `SELECT \* FROM users  
 WHERE username = '${username}'  
 AND password = '${password}'`  
  
 db.get(query, (err, row) => {  
 if (!row) {  
 res.send('Login failed')  
 return  
 }  
 // Login successful, set session etc.  
 res.send('Welcome ' + row.username)  
 })  
})

Note: This code is bad for multiple reasons – in particular, you should never store raw passwords in a database – but it demonstrates SQL injection clearly.

With this code, an attacker can bypass authentication by submitting the following input:

username: ' OR '1'='1  
password: ' OR '1'='1

This produces the following SQL query:

SELECT \* FROM users  
 WHERE username = '' OR '1'='1'  
 AND password = '' OR '1'='1'

As a result, the attacker will be able to log in successfully.

### Running Multiple Queries

Some database APIs allow multiple queries separated by semicolons. Consider this logging code:

db.exec(`INSERT INTO logs  
 VALUES ("Login attempt from ${username}")`)

An attacker could submit:

username: x"); DROP TABLE users; --

Producing:

INSERT INTO logs  
 VALUES ("Login attempt from x");  
 DROP TABLE users; --")

This executes two queries:

1. Insert a log entry
2. Delete the entire users table

## Defense: Use Parameterized Queries

Never build queries using string concatenation with untrusted input. Instead, use parameterized queries that handle escaping:

// Unsafe  
const query = `SELECT \* FROM users  
 WHERE username = "${username}"`  
  
// Safe  
const query = 'SELECT \* FROM users WHERE username = ?'  
const results = db.all(query, username)

The database driver escapes special characters and ensures user input can’t change the query structure.

## Summary

Server-side code injection vulnerabilities arise when applications fail to properly separate code from data. Command injection lets attackers execute system commands, while SQL injection enables database manipulation. The key defense is to never build commands or queries via string concatenation with untrusted input. Instead, use APIs that keep data separate from commands (e.g. spawn, SQL parameterized queries).