Java Injection Vulnerabilities

- Sources of untrusted data
  - From the network
    - Sockets
      - java.net.Socket.getInputStream()
      - java.net.Socket.getOutputStream()
    - WebSockets
    - Servlets
      - javax.servlet.Servlet.service(ServletRequest, ServletResponse) (interface)
      - javax.servlet.http.HttpServlet.doGet(HttpServletRequest, HttpServletResponse), doPost(..)
    - Spring REST API Framework
      - @RestController/@Controller: class containing methods that implement http operations
      - @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: methods to handle those http requests
      - @PathVariable, @RequestParam, @RequestHeader, @RequestBody: Maps fields from http request to parameters of methods
      - @ResponseBody, @ResponseStatus map elements of code to the response
  - From standard input
  - From files, environment variables
- Sinks
  - Command execution statements (Runtime.exec()) (command injection)
    - Runtime rt = Runtime.getRuntime(); Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
    - Solution: blacklisting : Runtime rt = Runtime.getRuntime(); if (Pattern.matches("[0-9A-Za-z]+", dir)) { Process p = rt.exec(new String[] {"sh", "-c", "ls " + dir});
    - Solution: whitelisting
  - SQL execution statements (JDBC java.sql.Statement) (SQL injection)
    - execute(String [,..])
    - addBatch(String)
    - executeQuery(String)
    - PrepareQuery(String); executeQuery()
    - executeUpdate(String [,..])
    - Solution: String sqlString = "SELECT * FROM users WHERE username = ? AND password= ?"; PreparedStatement stmt = connection.prepareStatement(sqlString); stmt.setString(1, username); stmt.setString(2, password); ResultSet rs = stmt.executeQuery();
  - Log statements, operations on local files (sensitive data leak, maybe caused by XML External Entities XXE javax.xml.parsers.DocumentBuilder javax.xml.parsers.SAXParser org.xml.sax.XMLReader – javax.xml.transform.Transformer)
    - parse(InputStream [,...])
    - parse(InputSource [,...])
    - parse(File[,...])
    - parse(String [,...]
    - transform(Source, Result)
    - Solution: SAXParserFactory factory = SAXParserFactory.newInstance(); SAXParser saxParser = factory.newSAXParser(); saxParser.parse(input, defaultHandler);
    - Custom entity resolver: SAXParserFactory factory = SAXParserFactory.newInstance(); SAXParser saxParser = factory.newSAXParser(); XMLReader reader = saxParser.getXMLReader(); reader.setEntityResolver(new CustomResolver()); reader.setErrorHandler(defaultHandler); reader.parse(new InputStream(input));
  - Statements that respond to http requests (xss) , send emails (phishing)
    - Reflected XSS if untrusted data written in HttpResponse -> void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException { String name = req.getParameter("name"); resp.getWriter().write("Hello, "+name); }
    - Stored XSS if untrusted data stored in a DB
    - Solution: resp.getWriter().write("Hello, "+Encode.forHtml(name)); or if (name.matches("[a-zA-Z]+")) resp.getWriter().write("Hello, "+Encode.forHtml(name));
  - Print function that takes unsanitized format strings -> Format strings don't let attacker read/write memory, but could crash the application.
  - Unsanitized regex under attacker's control -> Evil Regex DoS if (password.matches(username)) { log("Fatal error: password contains username"); } evil regex: ([a-zA-Z]+)* DOS triggered by input: aaaaaaaaaaaaaaaaaaaaaaaa!