Injection vulnerabilities

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

Injection

- Vulnerability class — CWE-74
  - *Improper Neutralization of Special Elements in Output Used by a Downstream Component*
  - Short name: ’**Injection**’
- Description
  - ***“The software constructs […] a command, data structure, or record using externally-influenced input** […] **but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted** […]”*
  - *“[…] **the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism.** While buffer overflows, and many other flaws, involve the use of some further issue to gain execution, **injection problems need only for the data to be parsed.** “*

Injection examples

OS command injection CWE-78

```
// PHP fragment
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

SQL injection CWE-89

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

- What is wrong with both fragments?
- SQL injection and OS command injection are the top two entries in the CWE/SANS Top 25 Most Dangerous Software Errors
- We will look at these two types of injection vulnerabilities in more detail today.

CWE TOP 25 - http://cwe.mitre.org/top25/

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function

- #1 and #2 are SQL and Command injection
- #3 and #4 - Buffer overflows and cross-site scripting (discussed later in the course)
  - require complementary mechanisms beyond plain data parsing
  - but also relate to bad input handling and rely on a blurred distinction between code and data

Injection — general attack pattern

- **General attack pattern**
  - Malicious input is supplied to a software system.
  - The input is used to build part of a command or executable code, that alters the expected flow of execution.
- **Basic problem**
  - Distinction between code and data is blurred !
  - Data disguised as code/command !
- **What must be do ?**
  - Ensure that (malicious) input cannot be construed as code or command.
  - “**All Input is Evil” / “Trust no input”:** security mantras we should keep in mind (for injection and various other types of vulnerabilities).
- **Terminology — Prevention vs mitigation vs detection**
  - **Detection**: plain detection of a vulnerability
  - **Mitigation**: limit possible damage
  - **Prevention**: take measures to eliminate the vulnerability

Running example — DVWA

- DVWA: Damn Vulnerable Web Application
  - Homepage
  - GitHub repository
  - Docker image

OS Command injection

OS commands

OS command injection CWE-78

```
// PHP fragment
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

- OS commands are often handy as “glue”, for example to invoke third-party software / OS utilities.
- Example ways to execute OS commands:
  - Traditional `system` call in C, Python, Perl, or equivalent support in other languages.
  - Within SQL in some database engines (!!!): `xp_cmdshell` in SQL Server
  - `exec` directive in "server-side includes"

OS command injection attacks

OS command injection [CWE-78]

```
// PHP fragment
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);
```

Example injection:
userName = ‘john; rm -fr /’

- Special meta-characters are interpreted by the system shell.
- Above you have something like:
  - `system(“someBeautifulCommand $input”)`
- Now consider
  - `$input = “someArg; rm -fr /“`
- The executed commands are in fact
  - `someBeautifulCommand someArg`
  - `rm -fr /`

Handling OS command injection

- #1 — Do not use OS command execution !
  - Not always practical !
- #2 — Prevention by input validation
  - Blacklisting strategy — **disallow** commands with special shell characters
  - Sanitisation — Meta-character escaping for sanitisation
  - Whitelist — **allow** only (validated) commands that adhere to a strict format
  - Blacklisting / sanitisation are prone to loopholes: system shells can be very heterogeneous and feature-rich; escaped characters may still be dangerous due to sub-command invocation.
- #3 — Run with least privilege (mitigation)
  - Make sure commands are not invoked by processes with more privileges than necessary (e.g. process is not executed as root user!)
- #4 — Environment-based attacks (prevention)
  - Only use commands with full path specified and explicitly control PATH setting / other dangerous variables
- #5 — Detect statically or in live execution in “tainted mode” (detection)

Input validation examples — DVWA

- Have a look at the source code for command injection:
  - `low.php` : input is used directly to execute command
    - Exploit easy : `127.0.0.1 ; cat /etc/passwd` or `127.0.0.1 && cat /etc/passwd`
  - `medium.php`: blacklisting of “;” and “&&”, that are erased from input
    - Still easy: 127.0.0.1 | cat /etc/passwd
  - `high.php`: blacklisting of more characters, but a small typo breaks the validation logic!
    - Check the source code to understand why the following exploit works: 127.0.0.1 || cat /etc/passwd
  - `impossible.php`: whitelist strategy, validates that input is strictly an IP address, reject all other input

OS command injection — other techniques

- Environment may determine ***what*** commands are executed.
- Suppose you have
  - `system("someBeautifulCommand $input")`
- This assumes some`BeautifulCommand` is in the PATH of the running program …
- If attacker can "infect" the PATH, a malicious `myBeautifulCommand` may be executed instead.
- Variations
  - The path for dynamically-linked code can also be controlled by environment variables, e.g. LD_PRELOAD and LD_LIBRARY_PATH in Unix
  - IFS separator variable changes the interpretation of file names!

Shellshock

- **ShellShock:** A “family” of security bugs starting with CVE-2014-6271
- Base vulnerability:
  - The UNIX bash shell unintentionally executed commands that were concatenated to the end of function definitions stored in environment variables (!).
  - Environment variables should also be considered as program inputs!
  - Example
    - `export X='() { }; maliciousCommand'`
    - `runAnything # does not need to access X!!`
    - … and `maliciousCommand` would run!
  - Example exploits — HTTP request headers converted to environment variables to CGI programs in many platforms! So …
    - Check more info here and here.

Static analysis example — using WAP

```
 File: /Users/edrdo/qses19/tools/dvwa/DVWA/vulnerabilities/exec/
source/low.php
  = = = =  Vulnerability n.: 1  = = = =
  Type: OS Command Injection

  Vulnerable code:
  5:   $target = $_REQUEST[ 'ip' ];
  10:    $cmd = shell_exec( 'ping  ' . $target );

  Corrected code:
  5:   $target = $_REQUEST[ 'ip' ];
  10: if (san_osci($_REQUEST['ip']) == 0)
  11:      $cmd = shell_exec( 'ping  ' . $target );
```

- WAP (Web Application Protection) tool
  - Static analysis tool for PHP web applications
  - Developed at University of Lisbon

Perl's "taint mode"

```
#! /opt/local/bin/perl
$username = $ARGV[0];
system("finger $username");
```

Input-based command injection through command-line argument.

Environment-based command injection also possible. PATH may be arbitrarily set.

`-T` switch

```
#! /opt/local/bin/perl -T
$username = $ARGV[0];
system("finger $username");
```

This will force us to transform the program. Let's run it and see how!

- Small Perl program (similar to the Java example)
- Malicious use: `unsafe.pl "someUserName; arbitraryCommand"`
- `-T` switch activates "taint mode" which tries to keep track of "tainted" (unsecure) data and their use in security-sensitive spots
- This is for Perl, but "tainted modes" are available for other languages like PHP and Python

Perl's "taint mode" (2)

```
#! /opt/local/bin/perl -T
$username = $ARGV[0];
system("finger $username");
```

Insecure $ENV{PATH}

```
#! /opt/local/bin/perl -T
$ENV{PATH} = '/usr/bin';
$username = $ARGV[0];
system("finger $username");
```

Set PATH to a safe value.

Insecure dependency in system

```
#! /opt/local/bin/perl -T
$ENV{PATH}='/usr/bin';
$username = $ARGV[0];
if ($username =~ /^([\w\_]+)$/ ) {
  system("finger $1");
} else {
  print "Invalid argument: '" . $username . "
}
```

Regular expression match assumed to yield untainted data. The reasoning is to force program to perform input validation. Regular expression can still potentially be inadequate.
Note: **$1** in the call to **system** denotes the regular expression match result, assumed to be secure by the "taint mode" module.

SQL injection

SQL injection — Context

SQL injection [CWE-89]

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

- SQL (Structured Query Language): the standard query language for interfacing with relational databases.
- SQL code is defined by arbitrary programs in interface to a database engine, in conjunction with some language-specific API (e.g. JDBC for Java as above).
- SQL injection: malicious inputs affect SQL code to execute unintended functionality.

SQLi — example attacks

SQL injection is possible through the ‘id’ input!

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

- The id parameter, used for building the query, is a “front door” for arbitrary command execution possible leading to
- Command sequences (also called piggy-backed queries)
  - `id = “1234 ; DELETE FROM USERS”`
- UNION queries
  - `id = “1234 UNION SELECT PASSWORD FROM USERS WHERE ID=1234”`
- Tautologies
  - `id = “1234 OR 1=1”`

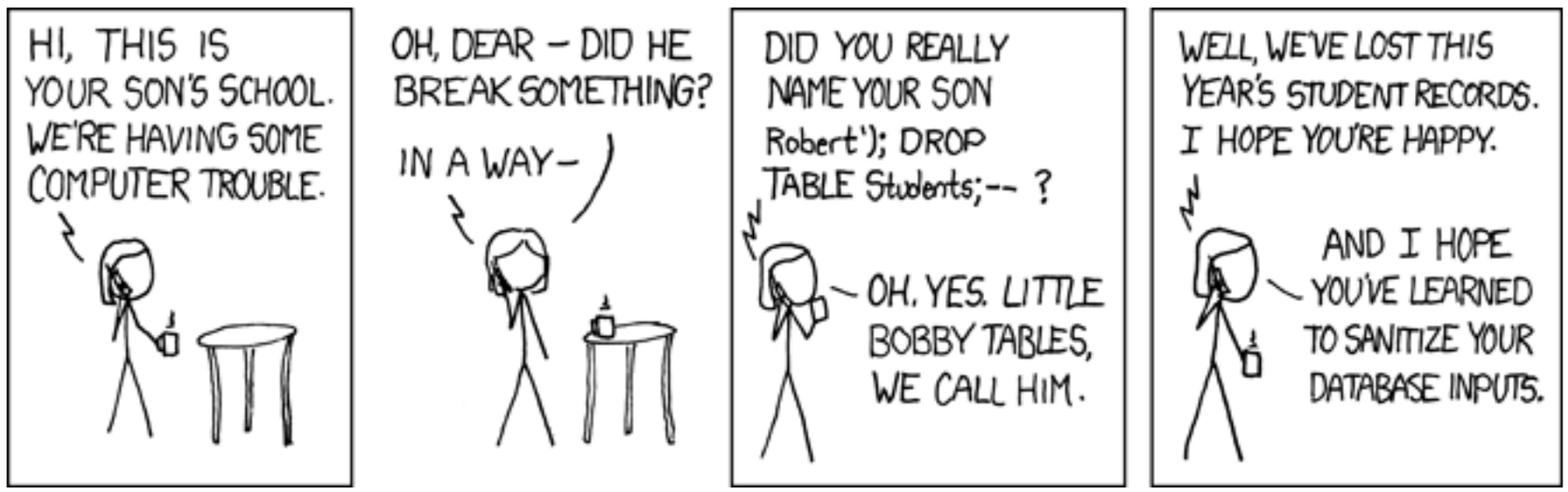
- Further reading: "***A Classification of SQL Injection Attacks and Countermeasures***", Halfond et al., ISSSE’06

SQLi — threats

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

- The id parameter, used for building the query, is a front door for arbitrary SQL injection possible leading to
  - Data tampering - modifying the database
  - Information disclosure - disclosing unauthorised data, e.g. the even the entire database schema
  - Denial of Service - by issuing a time-consuming query
  - …

Tales of ‘Bobby tables’



Exploits of a mom https://xkcd.com/327/

- SQLi attacks are common and cause serious damage
  - SQL injection hall of shame
  - The History of SQL Injection, the Hack That Will Never Go Away
- Some people just try to be funny (?) about it:
  - Did Little Bobby Tables migrate to Sweden?
  - ; DROP TABLE "COMPANIES";-- LTD

SQLi probes — malformed SQL

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

Searching for **O'Neil**

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'Neil' GROUP BY username ORDER BY username ASC' at line 1

This is the Query you executed (injection string is underlined):

SELECT username FROM users WHERE username LIKE 'O'Neil' GROUP BY username ORDER BY username ASC

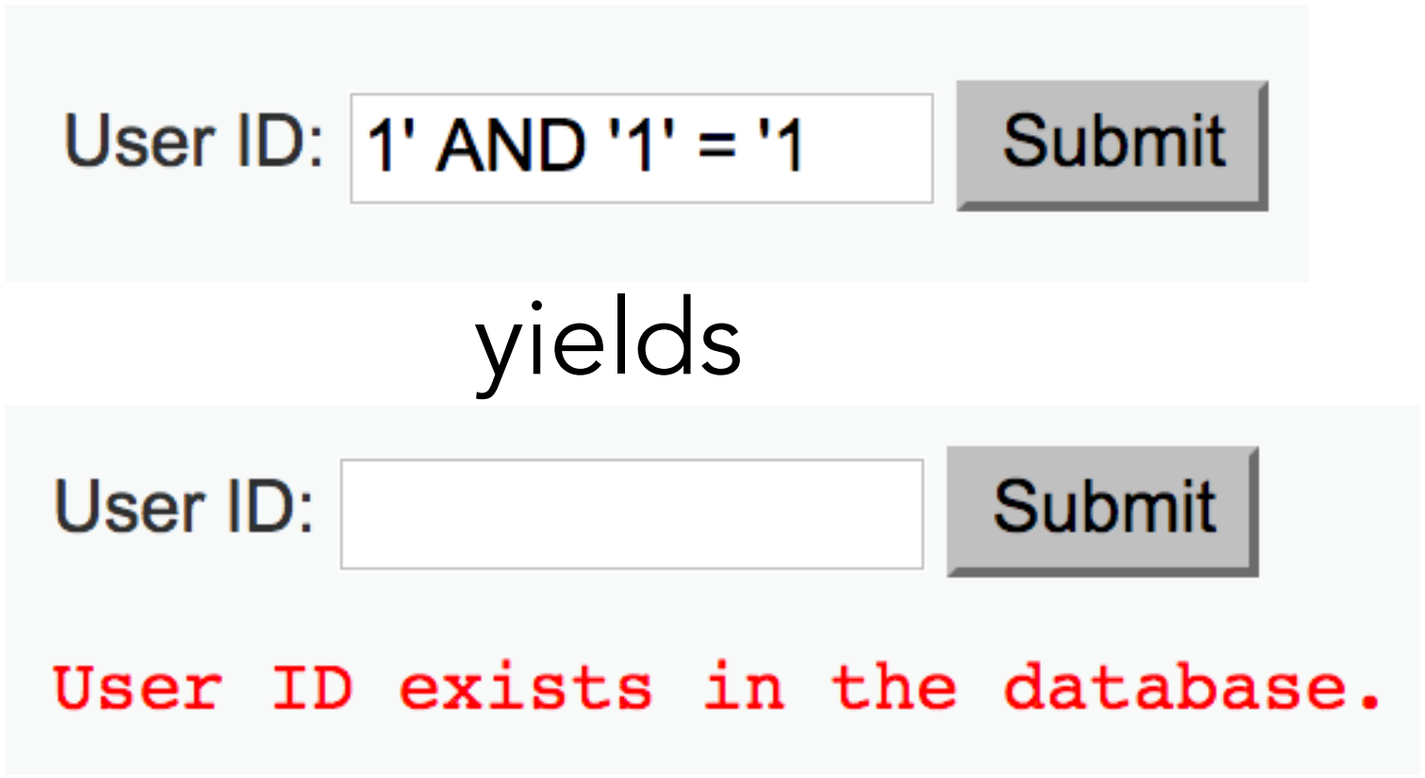
- Use malformed SQL
  - `id = “1234 FOLLOWED_BY_INVALID_SQL”`
  - Server may yield back detailed error messages providing positive indication that SQLi is possible

SQLi probes — Blind SQL injection

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

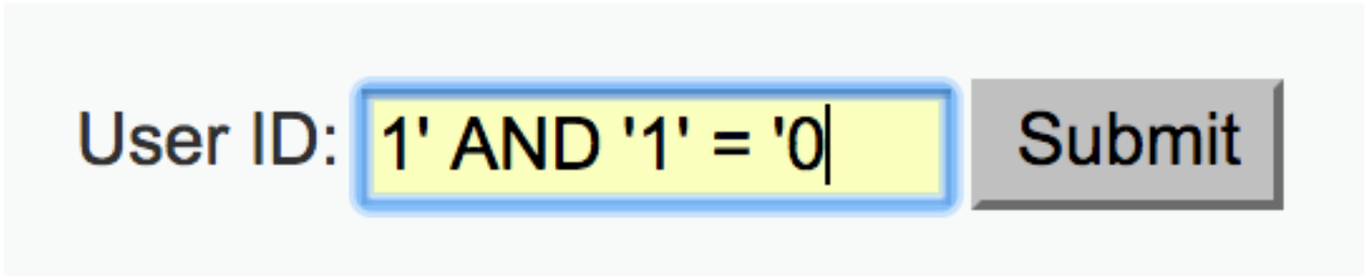
- Blind SQL injection works by issuing false / true statements and comparing the responses
  - id = “1234 AND 1=2”: server may report no data, but we may not be aware if the query was rejected / malformed
  - id = “1234 AND 1=1”: if output is different then SQLi should be possible

Blind SQLi - DVWA example

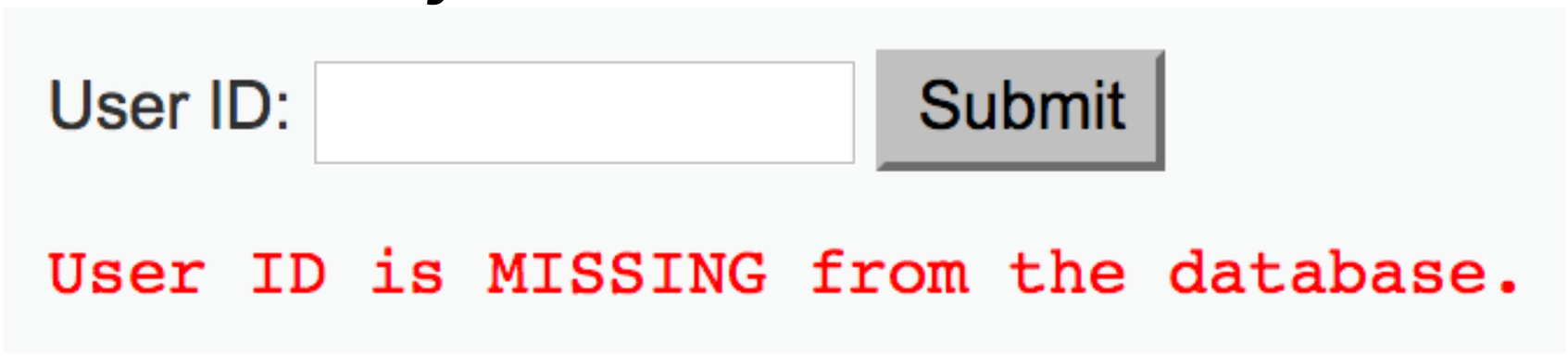


- Looks like a vulnerable answer …
- SQLi may be possible, how can we be sure ? Perhaps input was sanitised instead ?

Blind SQLi - DVWA example (2)



yields



- Different answer!
- SQLi point identified!

SQLi probes — time-based blind SQL injection

```
int id = input();
String query = "SELECT NAME FROM USERS WHERE ID=" + id
Statement conn = db.createStatement();
ResultSet rs = conn.executeQuery(query);
// ...
```

- Time-based SQL injection works by issuing functions that may cause a delay in the query
  - if a delay is noticeable, SQLi should be possible
  - E.g. SLEEP function for MySQL - info here at sqlinjection.net for different databases
  - DVWA demo: you may use `1' AND SLEEP(5)-- x`

SQLi and stored procedures

```
CREATE OR REPLACE PROCEDURE
prodDescr(vname IN VARCHAR2, vresult OUT VARCHAR2) AS
   vsql    VARCHAR2(4000);
BEGIN
   vsql := 'SELECT description FROM products
            WHERE name=''' || vname || '''';
   EXECUTE IMMEDIATE vsql INTO vresult;
END;
```

- Example taken from sqlinjection.net : we can attain SQL injection through the vname parameter
- Stored procedures are not necessarily more secure than embedded SQL.

Handling SQLi

- Applying general principles (mitigation)
  - Fail safely: do not leak internal database schema details in error messages.
  - Run with least privilege: standard programs should not connect as database administrator.
- Secure programming (prevention)
  - Input sanitisation by escaping input arguments (fragile) — example next.
  - Parameterised queries (secure) — example next.
- Detection
  - static analysis tools
  - “tainted” execution
  - pen-testing

Input sanitisation — DVWA example

`vulnerabilities/sqli_blind/source/medium.php`

```
$id = … mysql_real_escape_string($id, …)
$getid  = “SELECT first_name, last_name FROM users WHERE user_id = $id;"
```

Medium Level

The medium level uses a form of SQL injection protection, with the function of "mysql_real_escape_string()". However due to the SQL query not having quotes around the parameter, this will not fully protect the query from being altered.

The text box has been replaced with a pre-defined dropdown list and uses POST to submit the form.

Spoiler: `?id=a UNION SELECT 1,2;-- -&Submit=Submit`.

- Helpful, but prone to loopholes …

Parameterised queries

PHP (example from DVWA)

vulnerabilities/sqli/source/impossible.php

```
$data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE user_id =
(:id)');
$data->bindParam( ':id', $id, PDO::PARAM_INT );
$data->execute();
```

Java

```
String sql = "SELECT ID, NAME, PASSWORD, ROLE, CREATED FROM USERS WHERE LOGIN = ?";
PreparedStatement stmt = db.prepareStatement(sql);
stmt.setString(1, login);
```

- The most secure prevention for SQL injection:
  - **SQL statement is constant and unaffected by input.** Syntactic placeholders define that the statement may have inputs parameters.
  - **Input values are bound *after* preparing (compiling) the statement.**

Static analysis example — wap

```
> > > >  File: /Users/edrdo/qses/tools/dvwa/DVWA/vulnerabilities/sqli_blind/
source/high.php < < < <
     > Information:
        - Number of Lines of Code: 33
        - It is a include file: no
        - Included files: none
        - Defined user function: none
        - Number of Vulnerabilities detected: 1
           - Real Vulnerabilities: 1
           - False positives: 0

  = = = =  Vulnerability n.: 1  = = = =
  Vulnerable code:
  5:   $id = $_COOKIE[ 'id' ];
  8:   $getid  = "SELECT first_name, last_name FROM users WHERE user_id =
'$id' LIMIT 1;";
  9:   $result = mysqli_query($GLOBALS["___mysqli_ston"],  $getid ); //
Removed 'or die' to suppress mysql errors
```

Static analysis example — FindSecBugs

```
211
212  public User getUserUnsafe(String login) throws SQLException {
213     String sql = "SELECT ID, NAME, PASSWORD, ROLE, CREATED FROM USERS WHERE LOGIN = '" + login + "'";
214     try (ResultSet rs = db.prepareStatement(sql).executeQuery()) {
```

Problems | Javadoc | Declaration | Search | Console | Diagrams | Bug Explorer | Bug Info

UserDAO.java: 214

Navigation

Bug: This use of java/sql/Connection.prepareStatement(Ljava/lang/String;)Ljava/sql/PreparedStatement; can be vulnerable to SQL injection

The input values included in SQL queries need to be passed in safely. Bind variables in prepared statements can be used to easily mitigate the risk of SQL injection.

Vulnerable Code:

```
Connection conn = [...];
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("update COFFEES set SALES = "+nbSales+" where COF_NAME = '"+coffeeName+"'");
```

Solution:

```
Connection conn = [...];
conn.prepareStatement("update COFFEES set SALES = ? where COF_NAME = ?");
updateSales.setInt(1, nbSales);
updateSales.setString(2, coffeeName);
```

Pen-testing DVWA with sqlmap

```
python ./sqlmap.py \
  -u "http://127.0.0.1:8080/vulnerabilities/sqli/id=2&Submit=Submit#" \
  --cookie="PHPSESSID=89qopqq69ja0dggmbqa4mmsis4; security=low” \
  -b --current-db --current-user
```

```
GET parameter 'id' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 211 HTTP(s) requests:
---
Parameter: id (GET)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (MySQL comment) (NOT)
    Payload: id=2' OR NOT 1576=1576#&Submit=Submit

    Type: error-based
    Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
    Payload: id=2' AND (SELECT 2030 FROM(SELECT COUNT(*),CONCAT(0x716b707671,(SELECT
(ELT(2030=2030,1))),0x71767a7071,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY
x)a)-- avQj&Submit=Submit

…
```

Other types of injection

Code injection

```
my $code = "config_file_$action_key(\$fname, \$key, \$val);";
eval($code);
```

- Code injection — CWE-94 (click link to see full example)
  - Malicious input leads to unintended code execution
- Typically present in programs written using scripting languages (Python, PHP, Javascript, …) that facilitate dynamic definition of code, in particular through ***eval***-like constructs.
- A general advice on eval:
  - "Do not ever use eval" !!
- Is it widely used anyway? Yes. Are there ways around it? Yes.
  - "The Eval that Men Do — A Large-scale Study of the Use of Eval in JavaScript Applications", Richards et al., ECOOP'11
  - "Remedying the eval that men do", Jensen et al., ISSTA'12

A few other types of injection

- CWE-117 — Log Injection
- CWE-91 — XML injection
- CWE-643 — XPATH injection
- CWE-90 — LDAP injection