CyberSecurity: Principle and Practice

BSc Degree in Computer Science
2025-2026

Lesson 10: Injection Attacks

Prof. Mauro Conti
Department of Mathematics
University of Padua
mauro.conti@unipd.it
http://www.math.unipd.it/~conti/

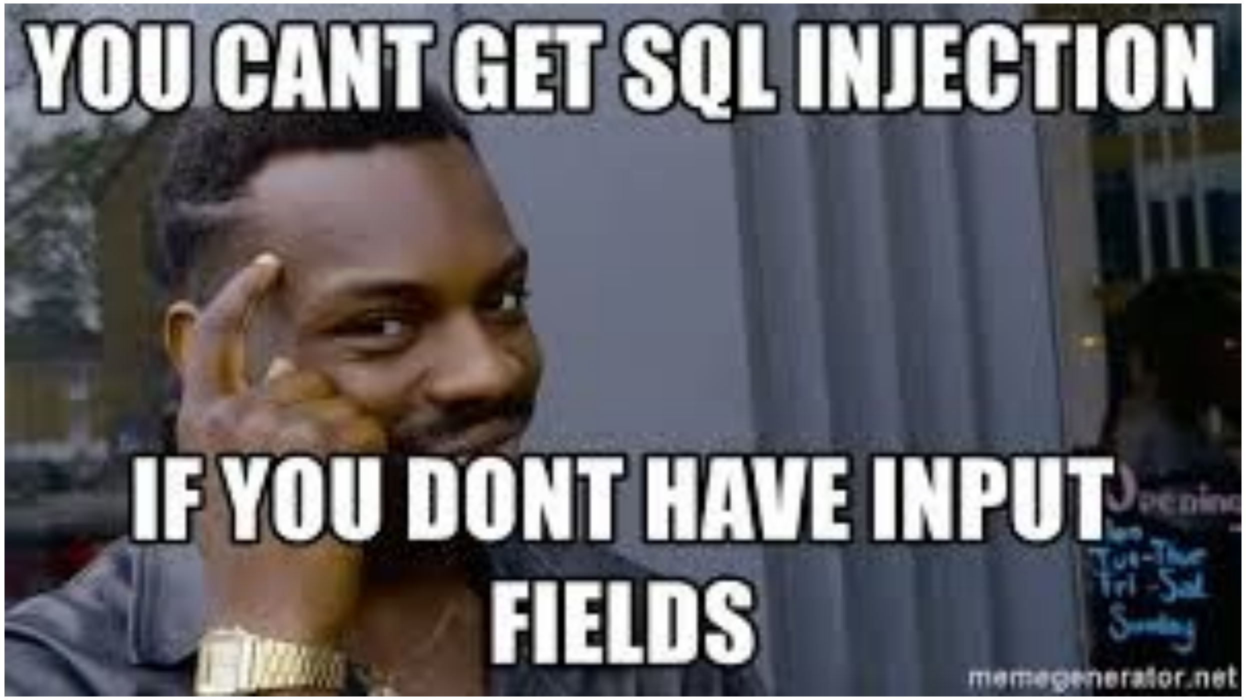
Teaching Assistants
Giulio Umbrella
giulio.umbrella@phd.unipd.it
Francesco De Giudici
francesco.degiudici@studenti.unipd.it

UNIVERSITÀ DEGLI STUDI DI PADOVA

SPRITZ SECURITY & PRIVACY RESEARCH GROUP

DIPARTIMENTO MATEMATICA

Preliminaries



Case 3: CRLF

- Intro to CRLF:
  - the web server uses the CRLF to understand when new HTTP header begins and another one ends
  - If an attacker can inject \r\n where it doesn't belong, they can break or tamper with HTTP headers or the response body, enabling response splitting, header injection, cache poisoning, XSS, session fixation, mail injection, or log forging.

Case 3: CRLF

- The attacker injects the CRLF characters into the input
- potential impact:
  - Injection of other attacks
  - information disclosure
- for example, a web server might collect logs
  - 123.123.123.123 - 08:15 - /index.php?page=home
- if an attacker can execute the CRLF attack, he can fake the log content

/index.php?page=home&**%0d%0a**127.0.0.1 - 08:15 - /index.php?page=home&restricredredactio=edit

- this attack will produce two entries in the log file
  1. 123.123.123.123 - 08:15 - /index.php?page=home
  2. 127.0.0.1 - 8:15 - /index.php?page,=home&restrictredaction=edit

Case 4: Cross-site Scripting (XSS)

- injection of custom scripts into a legitimate web server
  - usually JS codes
- this code is then executed in the victim's browser
  - occur when the victim visits the web application
- the web page is the vehicle of the attack
  - e.g., forums, message boards, and web pages that allow comments

Case 4: Cross-site Scripting (XSS)

- e.g., a browser might have a function that shows the last comment published, available in the database

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

- the programmer assumption is that this should only contain text
- an attacker might inject the following:

```
<script>doSomethingEvil();</script>
```

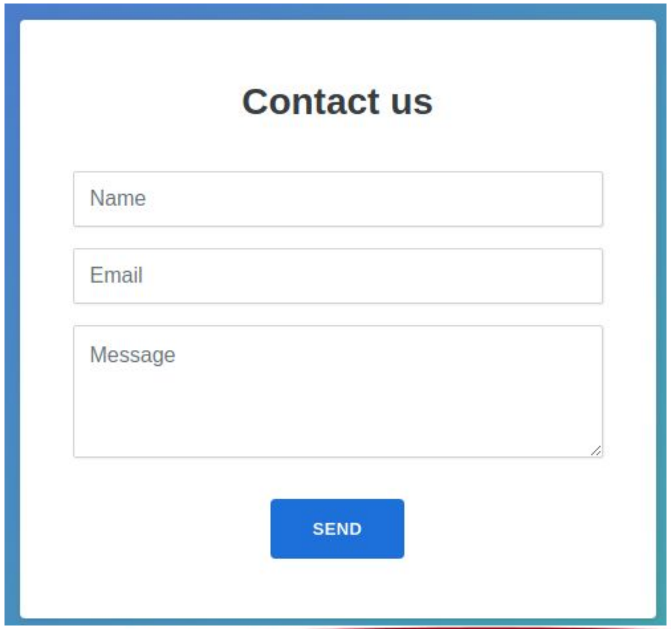
- The server will provide the following HTML:

```
<html>
<h1>Most recent comment</h1>
<script>doSomethingEvil();</script>
</html>
```

- when the victim loads the content, the script will be executed

Case 5: Email Header Injection

- Several web pages implement contact forms
- Most of the time, such contact forms set headers
- These headers are interpreted by the email library on the web server
    - turned into resulting SMTP commands
    - processed by the SMTP server
- A malicious user may be able to introduce additional headers into the message



Case 6: Host Header Injection

- Usually a web server (or IP address) hosts several web applications (trough virtual hosts or reverse proxy)
  - i.e., several web app in the same IP
- The **host header** specifies which website or web application should process an incoming HTTP request
  - the host header will dispatch the request to the proper application
- If the server implicitly trusts the Host header, and fails to validate or escape it properly, an attacker may be able to use this input to inject harmful payloads that manipulate server-side behavior.

Case 6: Host Header Injection

1. For example, say an attacker sends the following POST request to request a password reset for the “alice” user:

   ```
   POST /password-reset HTTP/1.1
   Host: www.attackers-domain.example.com
   ...
   user=alice
   ```

2. The application uses the value from the Host header to construct the reset link and sends an email to the alice user with the link:

   ```
   www.attackers-domain.example.com?reset_token=super
   -random-reset-token-value-12345
   ```

3. If the user clicks the link, the attacker captures the reset token and can reset the user’s password to their own value.

Case 7: SQL Injection

- injection of SQL instructions
- for example, given a database with credentials

```
# Define POST variables
uname = request.POST['username']
passwd = request.POST['password']

# SQL query vulnerable to SQLi
sql = “SELECT id FROM users WHERE username=’” + uname + “’ AND
password=’” + passwd + “’”

# Execute the SQL statement
database.execute(sql)
```

- The query returns always True if an attacker injects the following password:
  - password' OR 1=1

Questions? Feedback? Suggestions?