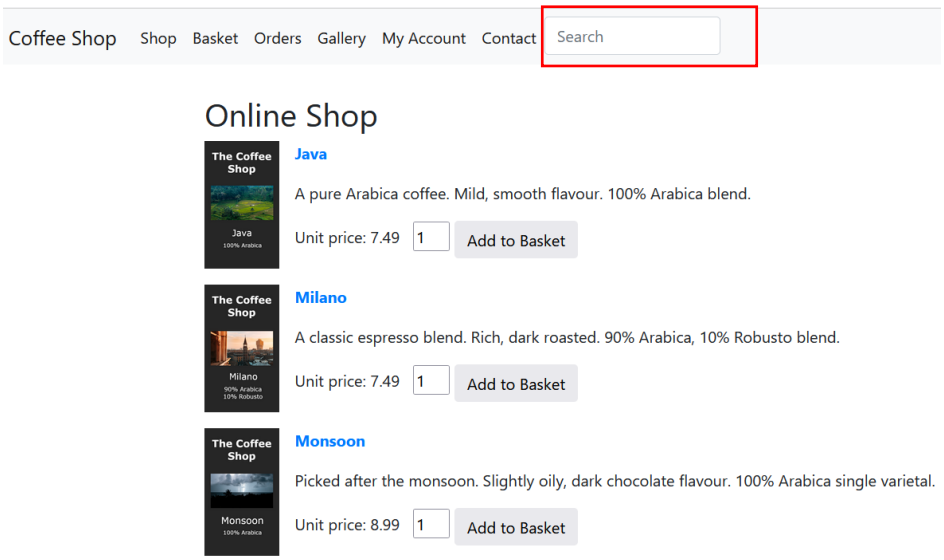
Secure Programming Lab
Injection Attacks

Today's lab will look at injection attacks. Injection attack vulnerabilities occur as a result of code that does not sanitise inputs from outside sources, e.g., user's login data from a HTML form.

Lab Deliverables

Please answer the 4 questions and the task (create your own SQL injection query) and upload the views.py fixed code.

The Coffeeshop application database is Postgres. Although you may not be familiar with this database system, the SQL commands will be quite similar to MySQL. Open up the Coffeeshop application on your web browser: http://127.0.0.1:8080/. We are going to demonstrate how the search function on the site is vulnerable to SQL injection attacks.



Retrieving users' credentials

We will now use an injection attack on the vulnerable search function to retrieve the users and credentials from the database. Let's first look at the backend code (You can access the code in the vagrant coffeeshop directory – see "Interacting with VMs.." document). The code can be found in the **views.py** code. The **search function** is the code we are interested in:

```
def search(request):
    log = logging.getLogger('django')

    cart_size = get_cart_size(request.user)
    search_term = None
    if ('search' in request.POST):
        search_text = request.POST['search']

    if (search_text is not None and search_text != ''):
        with connection.cursor() as cursor:
            template = "SELECT id, name, description, unit_price" + \
                       "    FROM coffeeshop_product" + \
                       "   WHERE (LOWER(name) like '%{}%'" + \
                       "          or LOWER(description) like '%{}%') "
            sql = template.format(search_text.lower(), search_text.lower())
            log.info("Search: " + sql)
            products = []
            try:
                cursor.execute(sql)
                for row in cursor.fetchall():
                    (pk, name, description, unit_price) = row
                    product = Product(id=pk, name=name,
                                      description=description,
                                      unit_price=unit_price)
                    products.append(product)
            except Exception as e:
                log.error("Error in search: " + sql)

    context = {"products": products, "cart_size": cart_size,
               "header": 'Search'}
    return render(request, 'coffeeshop/index.html', context)
```

Figure 1: vulnerable search function

```
            template = "SELECT id, name, description, unit_price" + \
                       "    FROM coffeeshop_product" + \
                       "   WHERE (LOWER(name) like '%{}%'" + \
                       "          or LOWER(description) like '%{}%') "
```

The **template** string constructs a template for a SQL SELECT query that retrieves product information (id, name, description, and unit_price) from the coffeeshop_product table. The query uses the LOWER() function to make the search case-insensitive, and the LIKE operator is used to perform a partial match on the name and description fields. For example, if I enter the word "java" into the search box, the equivalent SQL SELECT statement created would be:

```
SELECT id, name, description, unit price
FROM coffeeshop_product
WHERE (LOWER(name) LIKE '%java%'
OR LOWER(description) LIKE '%java%')
```

This Django code is vulnerable to SQL injection because it uses Python string formatting (format()) to directly insert user input **(search_text**) into the SQL query. By formatting the SQL query string with unvalidated user input, the application will allow malicious users to inject arbitrary SQL code, potentially gaining unauthorized access or modifying the database.

We will exploit the vulnerability by using the search term to display all usernames and hashed passwords from the **auth_user** table (columns username and password).

Note: The auth_user table is a **default table** in Django applications that use the Django authentication system. It's part of the django.contrib.auth module, which provides authentication features like user login, logout, password management, and more. This makes it easier for attackers to potentially exploit the database.

When you create a Django project, the auth_user table is automatically created during the migration process. This table is used to store user-related information such as usernames, hashed passwords, email addresses, and more.

Enter the following query into the search box:

```
xxx') union select id, username, password, 0.0 from auth_user --
```

Using the SQL injection string above, the following SQL query is generated:

```
SELECT id, name, description, unit_price
FROM coffeeshop_product
WHERE (LOWER(name) LIKE '%xxx%')
OR LOWER(description) LIKE '%xxx') UNION SELECT id, username, password, 0.0
FROM auth_user --%
```

If you look at the template string, the SQL SELECT QUERY contains an id, name, description and unit price. The corresponding union in the SQL injection string has to have matching data types for the selects on either side. As the code selects an integer, two strings, and a float, we have to select the same after the union. As the auth_user table has no float columns, we select a constant value as a placeholder to avoid errors (0.0).

Figure 2 shows the result of this attack. This data represents the users and their hashed passwords stored in Django's auth_user table.

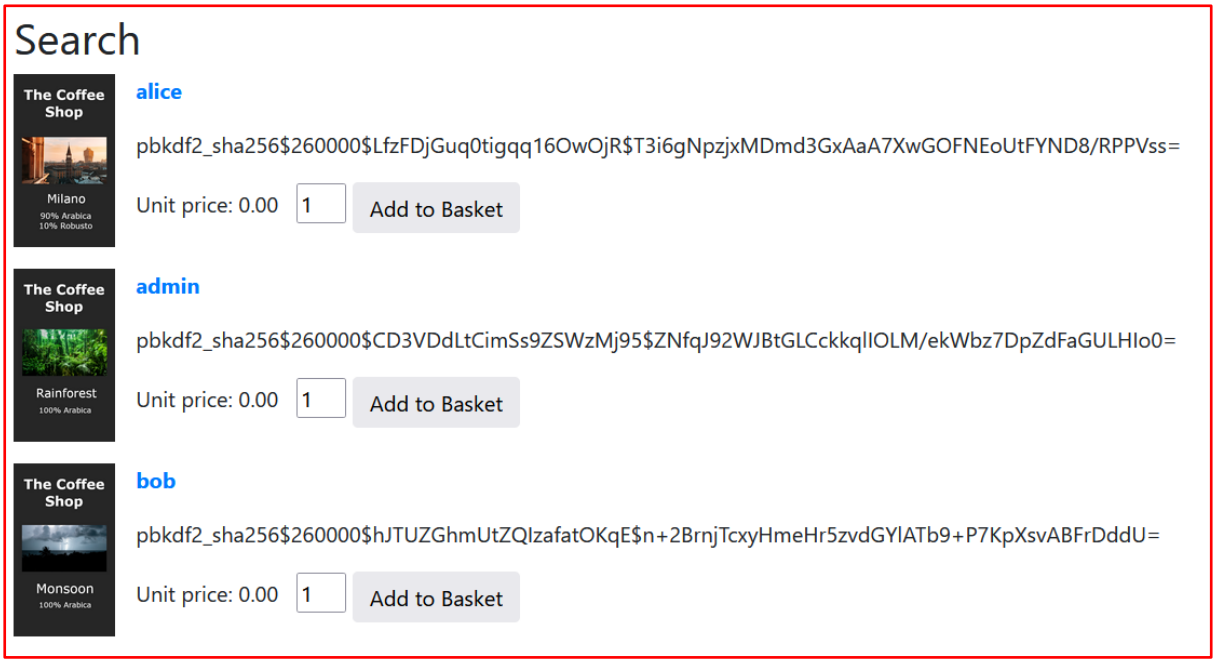


Figure 2: SQL injection results

The string returned is **not** the actual password, but rather a securely hashed version of the password that is stored in the database. Django typically stores password hashes using a format that includes the algorithm, the iteration count, the salt, and the hash. The format returned follows this structure:

algorithm$iterations$salt$hash

Taking Alice's details as an example:

pbkdf2_sha256:

- PBKDF2 (Password-Based Key Derivation Function 2) algorithm, combined with the SHA-256 hash function.

260000:

- This is the number of iterations used by the PBKDF2 algorithm when generating the hash.

LfzFDjGuq0tigqq16OwOjR:

- This is the salt used during the hashing process. A salt is a random string added to the password before hashing it to ensure that even if two users have the same password, their hashes will be different. This salt is unique for each password, and it prevents precomputed attacks like rainbow table attacks.

T3i6gNpzjxMDmd3GxAaA7XwGOFNEoUtFYND8/RPPVss=:

- This is the actual hashed password produced by the PBKDF2 algorithm using the provided password, salt, and iteration count.

Note: An attacker who obtains this hash cannot immediately recover the password, but they could attempt to brute-force it by trying many different passwords until they find one that, when hashed with the same salt and parameters, matches the stored hash. However, due to the use of a strong algorithm (PBKDF2 with many iterations) and a unique salt, this process would be computationally expensive and time-consuming and so not feasible.

There is lots of malicious injection attacks we can perform against the database. Much of the attacks require deeper knowledge of the database. But how can we get that information? Type the following into the search box:

```
xxx') UNION SELECT 1, table_name, column_name, 4 FROM
information_schema.columns WHERE table_name = 'auth_user' --
```

Q. 1 What is the result?

Q.2 You see terms like date_joined, is_active, is_staff etc. What do you think these terms represent?

Elevation of Privileges

Let's say you are bob and have an account on the site and you want to elevate privileges to admin. You know from some reconnaissance that the admin control panel is located at: http://127.0.0.1/admin/ (Figure 3). You have also gathered intelligence on the DB table in the previous section. Currently, you only have "regular" user permissions and cannot login to the admin panel. Try logging in with bob's credentials. You can find Bob and other user's credentials (along with all other users) in the **config.env** file located in the **coffeeshop/secrets** directory.

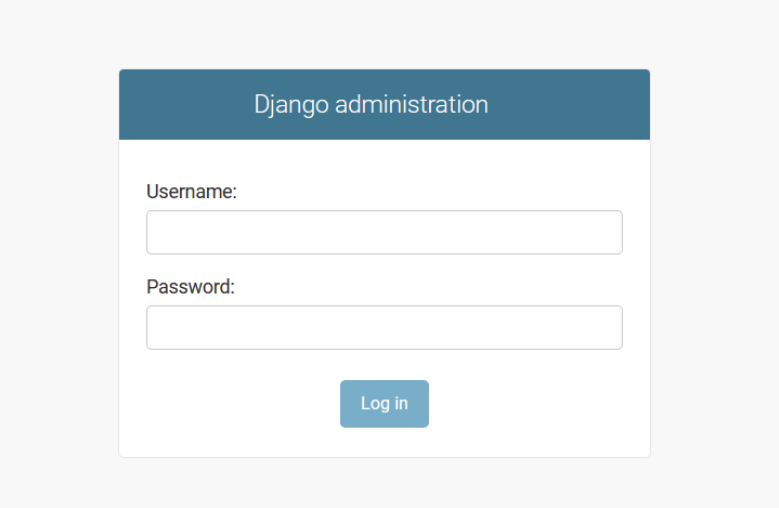


Figure 3: Admin panel login

Now, enter the following into the search box:

```
xxx'); update auth_user set is_staff=true, is_superuser=true where username = 'bob' --
```

You have guessed from your initial reconnaissance that is_staff and is_superuser should be set to true to make Bob both a staff member and a superuser (admin). Try to login again with Bob's credentials.

Q3. What do you see now?

Q4. What malicious actions could Bob take on this page?

Be sure to reset Bob's permissions, in case we made need them for future labs. You can do this by entering the following in the search box:

```
xxx'); update auth_user set is_staff=false, is_superuser=false where username = 'bob' --
```

Task

Your final task is to come up with another SQL injection string. You can use variations of the ones we have used in the lab, or you can search online for more elaborate examples.

Now it's time to fix the vulnerability!!.....

Fixing the Vulnerability

We will fix the code using a **prepared statement**. This only requires changing a couple of lines of code, but it is effective in preventing SQL injection. In your views.py code, update the search function code to:

```
def search(request):
    log = logging.getLogger('django')

    cart_size = get_cart_size(request.user)
    search_term = None
    if ('search' in request.POST):
        search_text = request.POST['search']

    if search_text is not None and search_text != '':
        with connection.cursor() as cursor:
            sql = '''SELECT id, name, description, unit_price
                     FROM coffeeshop_product
                     WHERE LOWER(name) LIKE %s OR LOWER(description) LIKE %s'''
            print(sql)

            products = []

            try:
                search_term = '%' + search_text.lower().replace('%', '%%') + '%'
                cursor.execute(sql, (search_term, search_term))

                for row in cursor.fetchall():
                    (pk, name, description, unit_price) = row
                    product = Product(id=pk, name=name,
                                      description=description,
                                      unit_price=unit_price)
                    products.append(product)

            except Exception as e:
                log.error("Error in search: " + str(e))


    context = {"products": products, "cart_size": cart_size,
               "header": 'Search'}
    return render(request, 'coffeeshop/index.html', context)
```

IMPORTANT: Indentation in Python is crucial because it defines the structure and flow of the code. Unlike many other programming languages that use symbols like {} or keywords to mark code blocks (e.g., for loops, if statements, functions), Python relies entirely on indentation to determine how code is grouped. If you indent the code incorrectly, you will get errors in your code.

Restart Apache

Once you have completed the code fix, ssh into your VM and restart Apache using the following command: **sudo service apache2 restart**

Now retry the SQL injection attack on the search box. The attack should not succeed.

Explanation of code fix

Prepared statements are a feature of SQL engine APIs. Placeholders are inserted into the SQL query where user-provided data is expected. The statement is compiled, and the user input is passed as a parameter to the execution function along with the compiled SQL statement. They also improve performance, especially if the query is reused, as it only needs to be compiled once. The placeholder is %s. The **cursor.execute()** function compiles the statement and then inserts the parameters that follow the comma, in this case the search term is entered twice as it is checked against both the name and description. Note that the %s is a placeholder for all data types, not just strings.

Escaping characters

This application uses psycopg2, a popular PostgreSQL adapter for Python that handles automatic type conversion, which simplifies the process of interacting with the database. However, there are still cases where certain characters need to be escaped manually, especially when constructing SQL queries with wildcard searches or special characters like the percentage sign %. In our case, psycopg2 makes it a bit trickier when we want to use a LIKE clause in our SQL. The following line of code from the search function looks messy, but it is necessary:

```
search_term = '%' + search_text.lower().replace('%', '%%') + '%'
```

The line .replace('%', '%%') replaces any literal % characters in the search_text with %%. This is important because in Python's psycopg2, %s is used as a placeholder for parameters, and a literal % can be confused with that placeholder syntax if not escaped properly. By replacing % with %%, you're telling psycopg2 to treat it as a literal percentage sign, not part of a placeholder.