SQL injection
is a type of attack that occurs when an attacker is able to manipulate user input to inject malicious SQL statements into a web application's database.

This leads to data breaches, unauthorised access to sensitive information, and other security vulnerabilities.

The attack works by exploiting a web application's vulnerability to inject malicious SQL commands through the input fields that are then processed by the application's database. This can allow the attacker to access, modify, or delete data in the database, bypass authentication, and execute arbitrary code on the server.

Input Validation & Input Sanitisation

- Input validation involves ensuring that user input conforms to expected formats, such as checking that an email address has a valid format.
- Input sanitisation involves removing any characters or sequences that could be used to inject malicious SQL statements, such as removing quotes or semicolons.

Use parameterized queries.

- Parameterized queries allow user input to be passed as a parameter rather than being directly incorporated into the SQL statement.

- Parameterized queries help to prevent SQL injection attacks by ensuring that user input is treated as data rather than as part of the SQL code.

- By using parameterized queries, the application is able to automatically sanitise user input to prevent SQL injection attacks, helping to keep the application secure.

In Django, parameterized queries can be implemented using the ORM (Object-Relational Mapping) framework. Here's an example of how to use parameterized queries to protect a Django application against SQL injection attacks:

```
from django.db import connection

# Define a function that uses a parameterized query to retrieve user
data
def get_user_data(user_id):
   with connection.cursor() as cursor:
       cursor.execute("SELECT * FROM users WHERE id = %s", [user_id])
       row = cursor.fetchone()
   return row
```

By using parameterized queries like this, Django is able to automatically sanitise user input to prevent SQL injection attacks, helping to keep the application secure.

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a type of web application vulnerability that can allow attackers to inject malicious code into a website, potentially stealing sensitive data or taking over user sessions. In Django applications, XSS attacks can occur when user input is not properly validated or when untrusted data is included in dynamic content.

In Django applications, XSS attacks can occur when user input is not properly sanitised or when untrusted data is included in dynamic content.

Preventing XSS attacks in Django applications involves the use of security headers, such as Content Security Policy (CSP), can be effective in preventing XSS attacks. In Django, developers can use the built-in security features to prevent XSS attacks.

Examples Attacks:
An attacker could inject a malicious script into a comment.
When other users view the comment, the script could execute and steal their session cookies, allowing the attacker to take over their accounts.
Another example is an attacker embedding a script in an image or video file that is then uploaded to a website.

Prevention

- Always sanitise user input using Django's built-in security measures.
- Use output encoding to ensure that user-generated content is not interpreted as code.
- Use security headers, such as Content Security Policy (CSP), to restrict the types of content that a website can load, preventing attackers from injecting malicious scripts.

Django Implementation:

- Use the escape function from Django's utils.html module is used to ensure that any user-generated content is properly escaped before being displayed on the web page

```
from django.shortcuts import render
from django.utils.html import escape

def my_view(request):
return render(request, 'my_template.html', {'user_input': escaped_input,
'csrf_token': csrf(request)})
```

- Use Django's built-in CSRF protection middleware and XSS protection middleware

```
MIDDLEWARE = [    'django.middleware.security.SecurityMiddleware',
'django.middleware.csrf.CsrfViewMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',
'django.middleware.xss.XSSProtectionMiddleware',]
```