SECURITY AND TESTING IN DJANGO

Prepared by: Rupak Koirala and Raj Prasad Shrestha

REVIEW WEEK 6

- Introduction to ORM
- Advantages of using ORM
- Django ORM
- Django Models
- Django Migrations commands
- Django QuerySet APIs(SQL queries)
- Django Model Relationships

AGENDA

- Introduction to Web Security
- Theme of Web Security
- Common Web Security vulnerabilities
- Security in Django
- Introduction to Code testing
- Writing test cases in Django
- Testing test cases in Django

INTRODUCTION TO WEB SECURITY

- Internet can be a scary place.

PROBLEMS IN APPLYING WEB SECURITY

- As **Web developers**, we have a duty to do what we can to combat these web security attacks.
- Every Web developer needs to treat security as a fundamental aspect of Web programming.
- Unfortunately, it turns out that implementing security is *hard* – attackers need to find only a single vulnerability, but defenders have to protect every single one.
- **Rescue**:
- **Django** attempts to mitigate this difficulty.
- It's designed to automatically protect you from many of the common security mistakes that new (and even experienced) Web developers make.

THEME OF WEB SECURITY

- Never – under any circumstances – trust data from the browser.
- You should make it a general practice to continuously ask, "Where does the data come from?"

SOME COMMON WEB SECURITY VULNERABILITIES

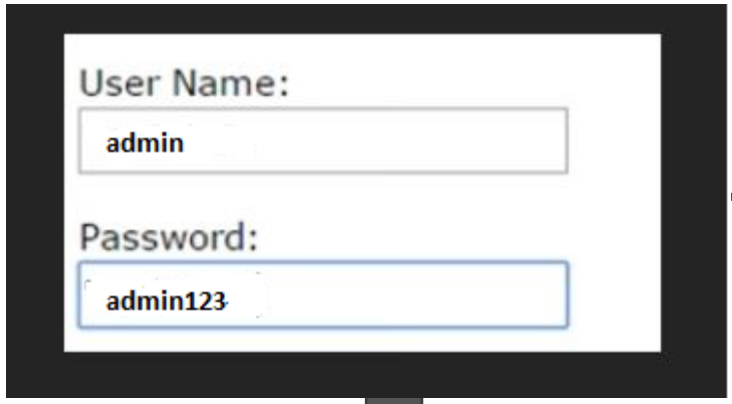
HERALD COLLEGE KATHMANDU

1. SQL Injection:

- SQL injection is a common exploit in which an attacker alters Web page parameters (such as GET/POST data or URLs) to insert arbitrary SQL statements that a naive Web application executes in its database directly.
- It's probably the most dangerous – and, unfortunately, one of the most common – vulnerabilities out there.

- When SQL injection is exploited it can be used to:
  - Extract arbitrary data
  - Insert data in the DB
  - Bypass authentication, Authorization controls
  - Control the server by executing OS commands

SQL INJECTION EXAMPLE



Login Form

get_username= request.POST['username']
get_password = request.POST['password']

sql = "SELECT * FROM User WHERE username="get_username" AND password ="get_password"

sql = "SELECT * FROM UserS WHERE username="admin" AND password ="admin123"

Login will be successful!!

SQL INJECTION EXAMPLE

Attacker

User Name:

" or ""="

Password:

" or ""="

Login Form

get_username= request.POST['username']
get_password = request.POST['password']

sql = "SELECT * FROM User WHERE username="get_username" AND password ="get_password"

sql = "SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""="" "

It will return all the rows from the "Users" tables

SQL INJECTION SOLUTION

- Although this problem is insidious and sometimes hard to spot, the solution is simple: *never* trust user-submitted data, and *always* escape it when passing it into SQL.
- The Django database API does this for you. It automatically escapes all special SQL parameters, according to the quoting conventions of the database server you're using (e.g., PostgreSQL or MySQL).

For example, in this API call:

```
foo.get_list(bar__exact="' OR 1=1")
```

Django will escape the input accordingly, resulting in a statement like this:

```
SELECT * FROM foos WHERE bar = '\' OR 1=1'
```

SOME COMMON WEB SECURITY VULNERABILITIES

HERALD COLLEGE KATHMANDU

2. Cross-Site Scripting (XSS):

- *Cross-site scripting* (XSS), is found in Web applications **that fail to escape user-submitted content properly before rendering it into HTML.**
- This allows an attacker to insert arbitrary HTML into your Web page, usually in the form of <script> tags,
- Attackers often use XSS attacks to steal cookie and session information, or to trick users into giving private information to the wrong person (aka *phishing*).

XSS EXAMPLE

VIEW FUNCTION

```
from django.http import HttpResponse

def say_hello(request):
    name = request.GET.get('name', 'world')
    return HttpResponse('<h1>Hello, %s!</h1>' % name)
```

This view simply reads a name from a `GET` parameter and passes that name into the generated HTML. So, if we accessed `http://example.com/hello/?name=Jacob`, the page would contain this:

Hello World, Jacob!

Output

XSS ATTACK EXAMPLE

VIEW FUNCTION

```
from django.http import HttpResponse

def say_hello(request):
    name = request.GET.get('name', 'world')
    return HttpResponse('<h1>Hello, %s!</h1>' % name)
```

This view simply reads a name from a `GET` parameter and passes that name into the generated HTML. So, if we accessed `http://example.com/hello/?name=Jacob`, the page would contain this:

http://example.come/hello/name=<h1>Hacked!!!</h1>

Hello, Hacked!

Output

XSS SOLUTION

- To guard against this, Django's template system automatically escapes all variable values.

View function

```
1   from django.shortcuts import render
2   def say_hello(request):
3       name = request.GET.get('name', 'world')
4       return render(request,'hello.html', {'name': name})
```

```
# hello.html

<h1>Hello, {{ name }}!</h1>
```

http://example.come/hello/name=<h1>Hacked</h1>

Hello,<i>Hacked</i>!

Django auto escaping back

SOME COMMON WEB SECURITY VULNERABILITIES

HERALD COLLEGE KATHMANDU

3. Cross-Site Request Forgery(CSRF):

- Cross-site request forgery (CSRF) happens when a malicious Web site tricks users into unknowingly loading a URL from a site at which they're already authenticated – hence taking advantage of their authenticated status.
- Django has built-in tools to protect from this kind of attack.

Cross Site Request Forgery (CSRF)

- CSRF is an attack where an attacker can force users of a website to perform actions without their permission.
- If a user is logged into website A, an attacker can let a user visit website B, which will perform actions on website A on behalf of the user.
- This happens because the forms in website A are not protected against CSRF.
- Basically CSRF means evil websites can let users of other websites perform actions without user permission.

CSRF EXAMPLE

```
<form method="POST" action="http://example.com/transfer/Bob">
    <input type ="number" name="amount" value="1000">
    <input type="submit">
</form>
```

Browser

Click submit button

Cookies
example.com
Sessionid:32

example.com django server

POST /transfer/Bob
Cookie:sessionId:32
amount =1000

sessionId: 32

Updates the table

CSRF ATTACK EXAMPLE

```
<form method="POST" action="http://example.com/transfer/Bob">
    <input type ="number" name="amount" value="1000">
    <input type="submit">
</form>
```

Browser

Click submit button

Cookies
example.com
Sessionid:32

hacker.com django server

POST /transfer/Bob
Cookie:sessionId:32
amount =1000

sessionId: 32

Updates the table

CSRF ATTACK SOLUTION

```
<form method="POST" action="http://example.com/transfer/Bob">
    <input type ="number" name="amount" value="1000">
    <input type ="hidden" name="csrf" value="12123asdfasdfasdfsdaf">
    <input type="submit">
</form>
```

Browser

Click submit button

Cookies
example.com
Sessionid:32
csrf:12123asdfasdfasdfsdaf

hacker.com django server

POST /transfer/Bob
Cookie:sessionId:32
amount =1000
csrf:12123asdfasdfasd
fsdaf

sessionId: 32
csrf:12123asdfasdfasdfsdaf

Updates the table

CODE TESTING

- As web applications become increasingly sophisticated and complex, it becomes increasingly important to thoroughly test them.
- Testing ensures that changes to a function used in many places don't cause completely different parts of the application to break.

AUTOMATED TESTING

HERALD COLLEGE KATHMANDU

Testing

Automated tests are a beneficial addition to any program. They not only help us to discover errors, but also make it easier for us to modify code – we can run the tests after making a change to make sure that we haven't broken anything. This is vital in any large project, especially if there are many people working on the same code. Without tests, it can be very difficult for anyone to find out what other parts of the system a change could affect, and introducing any modification is thus a potential risk. This makes development on the project move very slowly, and changes often introduce bugs.

Adding automated tests can seem like a waste of time in a small project, but they can prove invaluable if the project becomes larger or if we have to return to it to make a small change after a long absence. They can also serve as a form of documentation – by reading through test cases we can get an idea of how our program is supposed to behave. Some people even advocate writing tests *first*, thereby creating a specification for what the program is supposed to do, and filling in the actual program code afterwards.

TESTING IN PYTHON EXAMPLE

HERALD COLLEGE KATHMANDU

calculate.py

```python
def add(a,b):
    return a + b

# Testing add function
print("Sum = ",add(1,2))
print("Sum = ",add(-1,2))
```

Testing the add function

Output

PROBLEMS TERMINAL ··· 1: Code

```
PS D:\HeraldBIT\python\core\ADC6> python -u "d:\HeraldBIT\python\core\ADC6\calcualte.py"
Sum =  3
Sum =  1
```

Testing this way has some disadvantages:

1. Difficult to automate the test
2. Hard to maintain
3. No glance of what failed and what succeeded

UNIT TESTING MODULE IN PYTHON EXAMPLE

1. Creating test class

test_calculate.py

```
1   import unittest
2   import calculate
3
4   class TestCalculate(unittest.TestCase):
5       #Writing test cases here
```

2. Inheriting from TestCase class

UNIT TESTING MODULE IN PYTHON EXAMPLE

The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

UNIT TESTING MODULE IN PYTHON EXAMPLE

test_calculate.py

```python
import unittest
import calculate

class TestCalculate(unittest.TestCase):
    #Writing test cases here
    def test_add(self):
        result = calculate.add(4,5)
        self.assertEqual(result,9)
        self.assertEqual(calculate.add(-1,2),1)
```

3. Test case for add function

Output

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Test Passed

4. Run the test

UNIT TESTING MODULE IN PYTHON EXAMPLE

test_calculate.py

```python
import unittest
import calculate

class TestCalculate(unittest.TestCase):
    #Writing test cases here
    def test_add(self):
        result = calculate.add(4,5)
        self.assertEqual(result,9)
        self.assertEqual(calculate.add(-1,2),0)
```

test prefix

UNIT TESTING MODULE IN PYTHON EXAMPLE

HERALD COLLEGE KATHMANDU

Output

Run the test

Test failed!!!

```
F
======================================================================

FAIL: test_add (__main__.TestCalculate)
----------------------------------------------------------------------

Traceback (most recent call last):
  File "d:\HeraldBIT\python\core\ADC6\test_calculate.py", line 9, in t
est_add
    self.assertEqual(calculate.add(-1,2),0)
AssertionError: 1 != 0

----------------------------------------------------------------------

Ran 1 test in 0.001s

FAILED (failures=1)
```

UNIT TESTING MODULE IN PYTHON EXAMPLE

person.py

```python
class Person:
    TITLES = ['Dr', 'Mr', 'Mrs', 'Ms','Er']

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def fullname(self, title):
        return "%s %s %s" % (title, self.name, self.surname)
```

One test case for testing init method

One test case for testing fullname method

UNIT TESTING MODULE IN PYTHON EXAMPLE

test_person.py

```python
import unittest
from person import Person

class TestPerson(unittest.TestCase):

    def test_init(self):
        person1= Person("Jane", "Smith")
        self.assertEqual(person1.name, "Jane")
        self.assertEqual(person1.surname, "Smith")

        person1.name ="John"
        self.assertEqual(person1.name,"John")


    def test_fullname(self):
        person1= Person("Ram", "Adhikari")
        self.assertEqual(person1.fullname("Er"), "Er Ram Adhikari")
        self.assertEqual(person1.fullname("Dr"), "Dr Ram Adhikari")
```

UNIT TESTING MODULE IN PYTHON EXAMPLE

Run the test

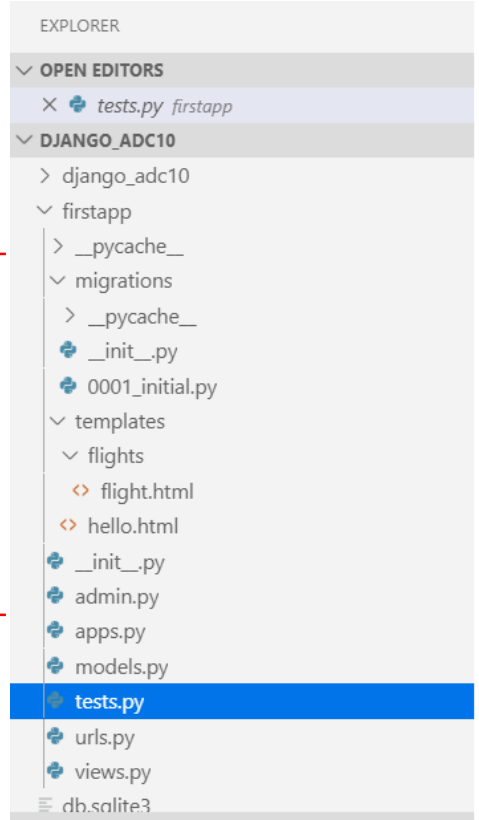
Output

```
..
----------------------------------------------------------------------

Ran 2 tests in 0.001s

OK
```

TESTING IN DJANGO

- Django has its own testing framework to make it easy to test web applications.
- Test code is found in the application directory in tests.py

Django app




```
from django.test import TestCase

# Create your tests here.
```

TESTING IN DJANGO

models.py

```python
from django.db import models

# Create your models here.
class Flight(models.Model):
    origin = models.CharField(max_length =50)
    destination = models.CharField(max_length= 50)
    duration = models.IntegerField()

    def __str__(self):
        return str(self.id) + " " + self.origin + " " + self.destination

    def is_valid_flight(self):
      return (self.origin != self.destination) and (self.duration >= 0)
```

Writing test case for this

TESTING IN DJANGO

tests.py

```python
from django.test import TestCase
from .models import Flight

# Create your tests here.
class FlightModelTestCase(TestCase):
    def test_valid_flight(self):
        flight1= Flight.objects.create(origin="Ktm", destination="Pkh", duration=100)
        self.assertTrue(flight1.is_valid_flight())

```

Running the test: python manage.py test

Output

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

References

1. https://docs.djangoproject.com/en/3.0/topics/security/
2. https://www.slideshare.net/levigross/django-web-application-security
3. https://www.youtube.com/watch?v=j-vdshWK9Lc
4. https://www.youtube.com/watch?v=6tNS--WetLI
5. https://python-textbok.readthedocs.io/en/1.0/Packaging_and_Testing.html#testing