Application Level Security Practical Project

Professor: E. Coppa

A.Y. 2022/23

Table of Contents

List of Figures

List of Tables

1 Introduction

Web application security is a critical concern for organizations and individuals alike, as the potential risks and consequences of an attack can be significant. One aspect of web application security is application level security, which involves protecting the application itself from various types of vulnerabilities and attacks. As part of a Cyber Security course, this report focuses on the architecture of Django, a powerful web application, as well as the identification and analysis of two types of vulnerabilities in this framework: bypassing access control based on URL paths and SQL injection. In this report, we will provide an overview of the web application and framework's architecture, describe the two identified vulnerabilities, discuss how the vulnerabilities were patched by the community, and provide guidance on how to prevent or mitigate these types of vulnerabilities in the future. Additionally, we will provide a Docker environment that reproduces the vulnerabilities to demonstrate a strong understanding of the framework and the vulnerabilities.

2 Django Architecture

Django is a popular open-source web application framework written in Python. It follows the **Model-View-Controller** (MVC) architectural pattern, which separates the application into three interconnected components: the Model, which manages the application's data and logic; the View, which handles the presentation layer and user interface; and the Controller, which manages the flow of data between the Model and the View.

In Django, the Model is represented by the database schema, which is defined using Python classes. The View is represented by the Python functions or classes that handle HTTP requests and generate responses. The Controller is represented by the URL routing system, which maps incoming requests to the appropriate View.

Other key components of a Django application include templates, which define the presentation layer and are typically written in HTML with embedded Python code; and forms, which provide a convenient way to handle user input and validate data.

Django also includes a powerful administrative interface, which allows developers to manage the application's data and configuration without writing any code. Django's architecture is designed to be flexible, scalable, and easy to maintain, making it a popular choice for building complex web applications.

3 Vulnerabilities - Description

In Django, the resolvers.py module is responsible for converting requested URLs to callback view functions. The URLResolver class is the main class in this module, and its resolve() method takes a URL as a string and returns a ResolverMatch object that provides access to all attributes of the resolved URL match.

The URLResolver class uses regular expressions to match requested URLs to URL patterns defined in the Django application's urls.py module. Once a match is found, the ResolverMatch object is returned, which provides access to attributes such as the view function that should handle the request.

4 Vulnerabilities - Exploitation

First, create a new directory for the project:

```
mkdir  django-vulnerability-demo
cd  django-vulnerability-demo
```

Create a new virtual environment and activate it:

```
python3 -m venv venv
venv\Scripts\activate.bat
```

Install Django version 3.1 in the virtual environment:

```
pip install Django==3.1
```

Create a new Django project:

```
django-admin startproject demo_project
cd demo_project
```

Create a new app within the project:

```
python manage.py startapp demo_app
```

In demo_project/settings.py, add the app to the INSTALLED_APPS list:

```
INSTALLED_APPS = [ 'demo_app',
'django.contrib.admin',
'django.contrib.auth','django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
]
```

In demo_app/views.py, create a new view that requires authentication to access:

```
from django.contrib.auth.decorators import login_required
from django.http import HttpResponse

@login_required
def protected_view(request):
return HttpResponse("This is a protected view!")
```

Create a new Django project:

Create a new Django project:

Create a new Django project:

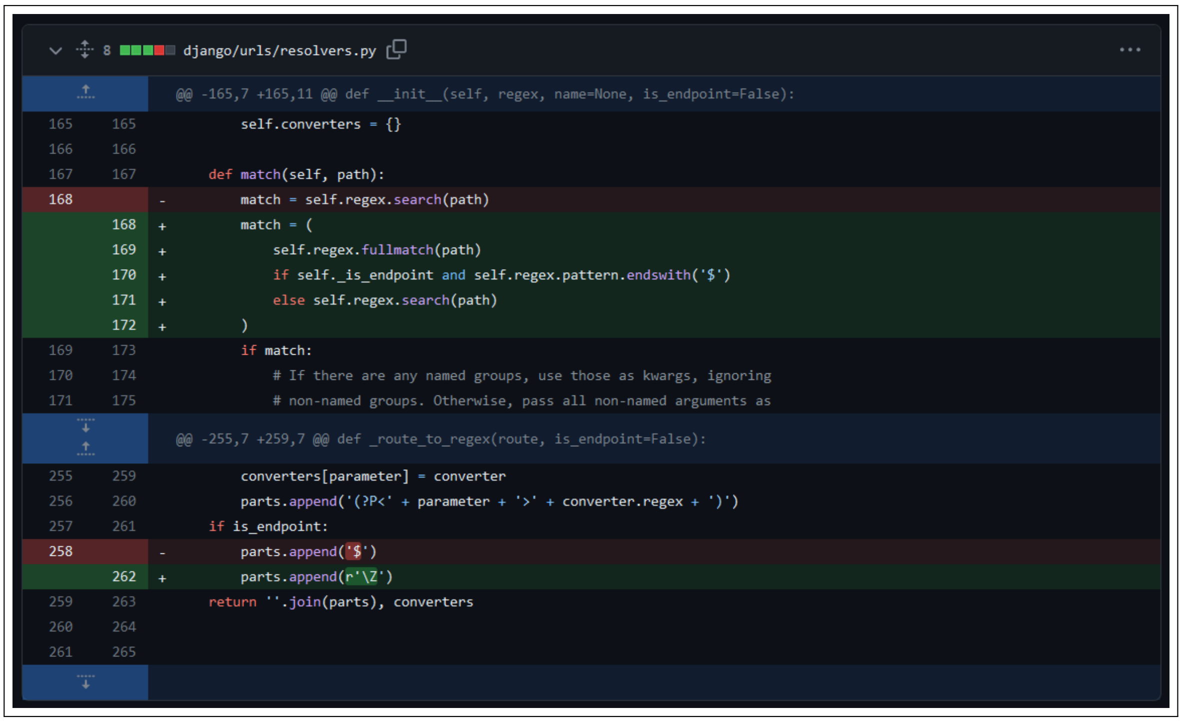
5 Vulnerabilities - Patched

The logic behind the issue is related to how Django handles URL path matching. In Django, URL patterns are defined using regular expressions, which are compiled into a regex object that is used to match incoming requests against the defined patterns.

The issue with the previous implementation of the match method is that it used the search method of the regex object to match the incoming request path. The search method looks for the pattern anywhere in the string, so it can match even if there is additional content after the end of the pattern. This means that a URL with a trailing newline character would match the pattern, even if it was not intended to.

The fix implemented in the new version of Django changes the way that URL pattern matching is performed. It uses the fullmatch method of the regex object instead of the search method, which requires the entire string to match the pattern. Additionally, it checks if the pattern ends with a $ character, which indicates an endpoint, and only uses the fullmatch method if the pattern is an endpoint. If the pattern is not an endpoint, it falls back to using the search method.

The modification to the match method adds a new condition that appends the \Z character to the regex pattern if the pattern is an endpoint. The \Z character matches the end of the string, so it ensures that there is no additional content after the end of the URL pattern. This ensures that URLs with trailing newline characters are not matched by the pattern, even if they are not intended to be.
This description can be seen on the main branch in the resolver class (Figure 1)



```
django/urls/resolvers.py

@@ -165,7 +165,11 @@ def __init__(self, regex, name=None, is_endpoint=False):
          self.converters = {}

      def match(self, path):
-         match = self.regex.search(path)
+         match = (
+             self.regex.fullmatch(path)
+             if self._is_endpoint and self.regex.pattern.endswith('$')
+             else self.regex.search(path)
+         )
          if match:
              # If there are any named groups, use those as kwargs, ignoring
              # non-named groups. Otherwise, pass all non-named arguments as
@@ -255,7 +259,7 @@ def _route_to_regex(route, is_endpoint=False):
          converters[parameter] = converter
          parts.append('(?P<' + parameter + '>' + converter.regex + ')')
      if is_endpoint:
-         parts.append('$')
+         parts.append(r'\Z')
      return ''.join(parts), converters
```

Figure 1: CVE 2021 45115 Issue On Main Branch Comparison