SECURE DOWNLOAD FILES

Kocliko - 2024

In Django, to handle confidential file downloads securely, you can leverage Django's file handling combined with permissions and custom views. Here's how to ensure confidentiality when downloading files:

Steps to Download a File with Confidentiality in Django:

1. **Store Files Securely**:
   - **Use Private Storage**: Store confidential files outside of the web-accessible directories (`MEDIA_ROOT`), so users can't directly access files via a URL.
   - You can store files in a non-public directory like `/srv/private_files/` on your server instead of in `media/`.
2. **Create a Secure View for File Download**:
   - Use Django's `HttpResponse` with the `FileResponse` class to serve the file.
   - Check user permissions or authorization before serving the file.

Example Code

models.py

python

Copier le code

```python
from django.db import models
from django.contrib.auth.models import User

class ConfidentialFile(models.Model):
    file = models.FileField(upload_to='confidential_files/')
    owner = models.ForeignKey(User, on_delete=models.CASCADE)  # Owner of the file
    description = models.CharField(max_length=255)

    def __str__(self):
        return self.description
```

views.py

python

Copier le code

```python
from django.http import FileResponse, Http404
from django.conf import settings
import os
from .models import ConfidentialFile

def download_file(request, file_id):
    try:
        file_record = ConfidentialFile.objects.get(id=file_id)

        # Verify the user is authorized to download the file
        if file_record.owner != request.user:
            raise Http404("You do not have permission to access this file.")

        # Securely serve the file
        file_path = os.path.join(settings.MEDIA_ROOT, file_record.file.name)

        if os.path.exists(file_path):
            response = FileResponse(open(file_path, 'rb'))
            response['Content-Disposition'] = f'attachment; filename="{file_record.file
            return response
        else:
            raise Http404("File not found.")
    except ConfidentialFile.DoesNotExist:
        raise Http404("File does not exist.")
```

urls.py

```python
from django.urls import path
from . import views

urlpatterns = [
    path('download/<int:file_id>/', views.download_file, name='download_file'),
]
```

Key Points:

1. **Authorization Check**: Before serving the file, we check whether the user is allowed to access it (`file_record.owner == request.user`).
2. **FileResponse**: Django's `FileResponse` is used to stream the file securely.
3. **Non-Web Accessible Storage**: Files should be stored in directories not directly accessible via URLs (e.g., outside `MEDIA_ROOT`).

Enhancements for Security:

1. **Serve Files with Expiration Links**: You can create one-time or time-limited URLs for file downloads.
2. **Encrypted Files**: Optionally, store files in an encrypted form and decrypt them before serving.
3. **Logging**: Keep logs of file access attempts and successful downloads.

BACKEND INTEGRATION

1. File Storage Considerations

Avoid Public File Storage:

Django's default `MEDIA_ROOT` is typically served directly via the web server, which can expose sensitive files. Instead:

- **Store Files Outside Web Root**: Place confidential files in a directory that is not web-accessible. For example, outside your project directory or web server document root, such as `/srv/private_files/`.
- **Use File Storage Backends**: You can configure Django to store files in cloud storage like AWS S3, but configure it to require authentication before accessing the files.

Configuration Example:

python Copier le code

```python
# settings.py

# Directory where confidential files are stored (outside MEDIA_ROOT)
CONFIDENTIAL_FILE_ROOT = '/srv/private_files/'
```

2. Serve Files Securely via Views

In Django, you should always serve confidential files through a controlled view instead of directly exposing the files. Django's `FileResponse` is useful for streaming large files without loading them entirely into memory.

python

Copier le code

```python
import os
from django.conf import settings
from django.http import FileResponse, Http404
from django.contrib.auth.decorators import login_required
from .models import ConfidentialFile

@login_required  # Ensures the user is authenticated
def download_file(request, file_id):
    try:
        file_record = ConfidentialFile.objects.get(id=file_id)

        # Authorization check: Ensure the user has permission to access this file
        if file_record.owner != request.user:
            raise Http404("You do not have permission to access this file.")

        # Securely serve the file from the private storage
        file_path = os.path.join(settings.CONFIDENTIAL_FILE_ROOT, file_record.file.name

        if os.path.exists(file_path):
            # Serve the file with correct headers
            response = FileResponse(open(file_path, 'rb'), as_attachment=True)
            response['Content-Disposition'] = f'attachment; filename="{file_record.file
            return response
        else:
            raise Http404("File not found.")
    except ConfidentialFile.DoesNotExist:
        raise Http404("File does not exist.")
```

Breakdown:

- `@login_required`: Ensures only authenticated users can access the view.
- **Permission Check**: `file_record.owner != request.user` ensures only the file owner can download the file.
- **File Handling**: The `FileResponse` object streams the file, minimizing memory usage.
- **Headers**: The `Content-Disposition` header is used to instruct the browser to download the file as an attachment with a specific filename.

3. Authentication and Authorization

Authentication:

Make sure the user is logged in before accessing any confidential files. Use Django's built-in authentication mechanism (`@login_required` or `request.user.is_authenticated`).

Authorization:

- **User Ownership**: In most cases, you will want to ensure that the file being downloaded belongs to the user making the request.
- **Custom Permissions**: You can add more complex permission checks based on user roles, groups, or even custom rules using Django's permission system or third-party packages like `django-guardian`.

python

Copier le code

```python
from django.core.exceptions import PermissionDenied

# Example of a custom permission check
if not request.user.has_perm('app.view_confidentialfile'):
    raise PermissionDenied
```

4. Logging File Access

Logging file downloads is a good practice, especially for sensitive data. You can log the user's identity, timestamp, and file details.

Example:

python Copier le code

```python
import logging

logger = logging.getLogger(__name__)

def download_file(request, file_id):
    try:
        file_record = ConfidentialFile.objects.get(id=file_id)

        # Authorization check
        if file_record.owner != request.user:
            raise Http404("You do not have permission to access this file.")

        # Log the download attempt
        logger.info(f"User {request.user} downloaded file {file_record.file.name} at {t

        # Serve the file
        file_path = os.path.join(settings.CONFIDENTIAL_FILE_ROOT, file_record.file.name
        if os.path.exists(file_path):
            return FileResponse(open(file_path, 'rb'), as_attachment=True)
        else:
            raise Http404("File not found.")
    except ConfidentialFile.DoesNotExi
        raise Http404("File does not exist.")
```

5.1. **Time-Limited Download Links**

You can generate time-limited URLs for file downloads to add another layer of security. For example, using Django's `signed` or `unsign` methods:

python Copier le code

```python
from django.core.signing import TimestampSigner
from django.utils.http import urlsafe_base64_encode

signer = TimestampSigner()
signed_value = signer.sign(file_id)
```

The file can only be downloaded for a specific period, and after that time, the link will expire.

5.2. **Encryption of Files**

For highly sensitive files, you might want to encrypt them before storing them. You could use Python libraries such as `cryptography` to encrypt the files and decrypt them on-the-fly before sending them for download.

python Copier le code

```python
from cryptography.fernet import Fernet

key = Fernet.generate_key()
cipher_suite = Fernet(key)

# Encrypt file contents before saving
with open(file_path, 'rb') as file:
    encrypted_data = cipher_suite.encrypt(file.read())

# Decrypt file contents before serving for download
with open(file_path, 'rb') as encrypted_file:
    decrypted_data = cipher_suite.decrypt(encrypted_file.read())
```

6. Cloud-Based Storage with Access Control

If using a cloud-based file storage solution such as AWS S3, ensure you configure access policies so that only authenticated users can access confidential files. For instance:

- **Private Buckets**: Store files in private buckets and generate pre-signed URLs for download.
- **Access Control Lists (ACLs)**: Define strict ACLs that limit who can download the files.

FRONTEND INTEGRATION

1. AJAX-Based File Download Handling

When a file is downloaded, it typically triggers a browser action, and there's limited control from JavaScript. However, we can use AJAX to make the request, verify permissions, and then trigger the file download.

Using AJAX to Request Secure Download:

Here's how you can structure your frontend to send a request for file download:

html

Copier le code

```
<button id="download-button" data-file-id="{{ file.id }}">Download File</button>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script
<script>
    $(document).on('click', '#download-button', function () {
        var fileId = $(this).data('file-id');
        $.ajax({
            url: '/download/' + fileId + '/',
            type: 'GET',
            xhrFields: {
                responseType: 'blob'  // To handle binary data (files)
            },
            success: function (data, status, xhr) {
                var filename = xhr.getResponseHeader('Content-Disposition').split('file
                var blob = new Blob([data], { type: xhr.getResponseHeader('Content-Type
                var link = document.createElement('a');
                link.href = window.URL.createObjectURL(blob);
                link.download = filename;
                document.body.appendChild(link);
                link.click();
                document.body.removeChild(link);
            },
            error: function (xhr, status, error) {
                alert("Failed to download file: " + error);
            }
        });
    });
</script>
```

- **AJAX Request**: The `$.ajax` call sends a request to your Django backend to retrieve the file.
- **ResponseType: blob**: Since we expect binary data (files), we set `responseType` to `blob`.
- **Dynamic Filename Handling**: The filename is extracted from the response header (`Content-Disposition`), and we create an anchor (`<a>`) to trigger the download.
- **Error Handling**: Display an alert if the file fails to download (could be enhanced with user-friendly messages).

2. Handling Large Files: Progress Bars

For large files, it's helpful to display a progress bar so that users can monitor the download process. We can achieve this by tracking the progress of the download using the `xhr` (XMLHttpRequest) object.

Example of Progress Bar Integration:

HTML:

html Copier le code

```html
<div>
    <button id="download-large-file" data-file-id="{{ file.id }}">Download Large File</
    <div id="progress-container">
        <progress id="download-progress" value="0" max="100"></progress>
        <span id="progress-text">0%</span>
    </div>
</div>
```

javascript Copier le code

```javascript
$(document).on('click', '#download-large-file', function () {
    var fileId = $(this).data('file-id');
    $.ajax({
        url: '/download/' + fileId + '/',
        type: 'GET',
        xhr: function () {
            var xhr = new window.XMLHttpRequest();
            // Handle progress
            xhr.addEventListener('progress', function (evt) {
                if (evt.lengthComputable) {
                    var percentComplete = Math.round((evt.loaded / evt.total) * 100);
                    $('#download-progress').val(percentComplete);
                    $('#progress-text').text(percentComplete + '%');
                }
            }, false);
            return xhr;
        },
        xhrFields: {
            responseType: 'blob'
        },
        success: function (data, status, xhr) {
            var filename = xhr.getResponseHeader('Content-Disposition').split('filename
            var blob = new Blob([data], { type: xhr.getResponseHeader('Content-Type') }
            var link = document.createElement('a');
            link.href = window.URL.createObjectURL(blob);
            link.download = filename;
            document.body.appendChild(
            link.click();
            document.body.removeChild(link);

            // Reset progress bar
            $('#download-progress').val(0);
            $('#progress-text').text('0%');
        },
        error: function (xhr, status, error) {
            alert("Failed to download file: " + error);
        }
    });
});
```

3. **Handling Download via Direct HTML Links with CSRF Protection**

In some cases, you might want to handle downloads without an AJAX call, using a direct link that includes token-based authentication or CSRF protection.

HTML:

html Copier le code

```
<form method="POST" action="{% url 'download_file' file.id %}">
    {% csrf_token %}
    <button type="submit">Download</button>
</form>
```

This approach ensures that CSRF tokens protect the download request, preventing unauthorized access.

4. Handling Multiple File Downloads (Batch Download)

If users need to download multiple files at once, you can bundle files into a ZIP file dynamically on the server and allow users to download it.

Django View for Zipping Files:

python Copier le code

```python
import zipfile
from io import BytesIO
from django.http import HttpResponse

def download_multiple_files(request, file_ids):
    # Check if the user has permission to access the files
    files = ConfidentialFile.objects.filter(id__in=file_ids, owner=request.user)

    # Create a Zip file in memory
    zip_buffer = BytesIO()
    with zipfile.ZipFile(zip_buffer, 'w') as zip_file:
        for file in files:
            file_path = os.path.join(settings.CONFIDENTIAL_FILE_ROOT, file.file.name)
            zip_file.write(file_path, file.file.name)

    # Serve the zip file
    response = HttpResponse(zip_buffer.getvalue(), content_type='application/zip')
    response['Content-Disposition'] = 'attachment; filename="files.zip"'
    return response
```

5. Handling Authentication on the Frontend

You might need to handle authentication for AJAX downloads. If your Django app uses session authentication or token-based authentication (like Django REST Framework's Token Authentication), ensure that proper authentication is handled in your requests.

Token-Based Authentication Example:

javascript Copier le code

```javascript
$.ajax({
    url: '/download/' + fileId + '/',
    type: 'GET',
    headers: {
        'Authorization': 'Token YOUR_API_TOKEN_HERE'
    },
    xhrFields: {
        responseType: 'blob'
    },
    success: function (data, status, xhr) {
        // Handle file download
    }
});
```

For CSRF protection in AJAX requests with Django, ensure that the CSRF token is sent with each request. You can retrieve the CSRF token from a meta tag in your HTML and include it in the AJAX header.

6. Improving User Feedback with Notifications

You can further improve the user experience by providing feedback using toast notifications (e.g., using libraries like `toastr` or `SweetAlert`) to notify users when the download starts or finishes successfully.

Example with Toastr:

javascript Copier le code

```javascript
$(document).on('click', '#download-button', function () {
    toastr.info('Your download is starting...');
    // Rest of your download logic
    $.ajax({
        // onSuccess:
        success: function() {
            toastr.success('Download completed successfully!');
        },
        error: function() {
            toastr.error('Download failed.');
        }
    });
});
```