# TASK-2

Secure Coding Review Choose a programming language and application. Review the code for security vulnerabilities and provide recommendations for secure coding practices. Use tools like static code analyzers or manual code review.

Let's consider a hypothetical Python application similar to what might be used in a Shopify-like environment. We'll review a basic example of a web application code and identify potential security vulnerabilities, then provide recommendations for secure coding practices.

Example Python Code (Django-based):

```
# views.py
from django.shortcuts import render, get_object_or_404
from .models import Product

def product_detail(request, product_id):
    product = get_object_or_404(Product, id=product_id)
    return render(request, 'product_detail.html', {'product': product})

def search(request):
    query = request.GET.get('q', '')
    products = Product.objects.filter(name__icontains=query)
    return render(request, 'search_results.html', {'products': products})

# models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
```

```python
views.py
from django.shortcuts import render, get_object_or_404
from .models import Product
```

```
def product_detail(request, product_id):
    product = get_object_or_404(Product, id=product_id)
    return render(request, 'product_detail.html', {'product': product})

def search(request):
    query = request.GET.get('q', '')
    products = Product.objects.filter(name__icontains=query)
    return render(request, 'search_results.html', {'products': products})

 models.py
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
```

Security Vulnerabilities and Recommendations:

1. ***SQL Injection:***
   - Vulnerability: Direct use of user input in database queries can lead to SQL injection.
   - Recommendation: Django's ORM is used here which generally mitigates this risk. Ensure that all queries are constructed using ORM methods to prevent SQL injection.

2. ***Cross-Site Scripting (XSS)***:
   - Vulnerability: Displaying user input without proper escaping can lead to XSS attacks.
   - Recommendation: Ensure all user inputs displayed in templates are properly escaped. Django's template engine escapes variables by default, but always double-check custom HTML outputs.

3. ***Cross-Site Request Forgery (CSRF)***:
   - Vulnerability: Unauthorized commands transmitted from a user that the web application trusts.
   - Recommendation: Use Django’s built-in CSRF protection by ensuring `{% csrf_token %}` is included in all forms.

4. ***Input Validation:***
   - Vulnerability: Insufficient validation can allow malformed data to be processed.
   - Recommendation: Validate all inputs using Django forms or serializers. Ensure proper validation rules are applied to each field.

5. Sensitive Data Exposure:

- Vulnerability: Sensitive data like error messages and user details might be exposed.
- Recommendation: Handle exceptions properly, avoid displaying detailed error messages to users, and use secure settings for database connections and sensitive data.

Secure Coding Practices:

**1. *Keep Dependencies Updated:***
- Regularly update Django and other dependencies to patch known vulnerabilities.

2. ***Use Strong Authentication and Authorization:***
- Implement strong password policies and use Django's built-in authentication system. Ensure proper user roles and permissions are enforced.

3. ***Secure Configuration***:
- Ensure settings like `DEBUG = False` in production, use secure cookie settings (`SECURE_COOKIE`), and enforce HTTPS (`SECURE_SSL_REDIRECT = True`).

4. ***Logging and Monitoring***:
- Implement logging for security-related events and monitor them for suspicious activity.

5. ***Code Reviews and Static Analysis***:
- Regularly review code for security issues and use tools like Bandit (a static code analyzer for Python) to identify potential vulnerabilities.

Example Static Analysis with Bandit:

To run Bandit on your project, you would use the following command:

```sh
bandit -r your_project_directory/
```

Bandit will analyze your code and provide a report on potential security issues, which you can then address accordingly.

By following these practices and regularly reviewing your code for vulnerabilities, you can significantly improve the security posture of your Python web application.