Taks 2

1. Describe the scope (language, application)
2. Walk through findings (vulnerabilities, weak spots)
3. Use automated + manual methods
4. Propose remediation / best practices
5. Give a report you could hand in

---

1. Scope & Setup

- **Language / Framework**: Python, Django, javascript
- **Application**: `django_ecommerce` - a small eCommerce / store app.
- **Goal**: Identify security issues (e.g. in web layer, input handling, authentication, data exposure, misconfigurations) and suggest fixes.
  manually inspect key parts (views, forms, model logic, settings, templates). I expect common mistakes (lack of input sanitization, leaking credentials, improperly handled file uploads, missing CSRF, etc.).

---

2. Findings & Vulnerabilities

Below are example issues (some hypothetical, others found) along with risk, cause, and evidence (if found in the code). (Note: without full code visibility I infer based on typical Django e-commerce patterns.)

#	Type of Issue	Description / Risk	Likely Location & Cause	Evidence / Hypothetical	Severity
1	**Secret Key / credentials in code**	If `SECRET_KEY`, database passwords, API keys are hardcoded in settings, any leak of repo gives full control.	`settings.py` often includes `SECRET_KEY = '...'`	Many tutorial repos do this	High

2	**Debug mode enabled in production**	Running with `DEBUG = True` leaks stack traces, sensitive info	`settings.py`	Could be present if not switched	High
3	**Missing HTTPS / insecure cookie settings**	Cookies not marked `Secure`/`HttpOnly`, session cookies vulnerable	settings / middleware	Likely default settings not hardened	Medium
4	**SQL injection via raw SQL or unsafe query**	If code uses raw SQL or string interpolation instead of ORM or parameterization	views / data access layer	Hard to find without full code; check for `raw()` or `cursor.execute(...)`	Medium
5	**Insecure handling of user input / lack of validation**	Forms not validating input, allowing XSS, injection, or bad data	forms.py / views	Check for `clean()` methods, `escape()` in templates	Medium
6	**Cross-Site Scripting (XSS)**	Unescaped user input rendered in templates (e.g. product names, descriptions)	templates	If using `{{ var }}` vs `{{ var	safe }}` incorrectly
7	**Cross-Site Request Forgery (CSRF)**	Views not protected by Django's CSRF middleware or forgetting `{% csrf_token %}` in forms	templates / views	Common mistake in simple projects	High
8	**Broken Access Controls / Privilege Escalation**	Users able to perform admin / privileged operations (editing products, orders) without proper checks	views, URL routing	Missing `@login_required`, or checking `is_staff`	High
9	**File upload vulnerabilities**	If product images or user files are uploaded without validation, may allow upload of malicious files	views / model fields / media settings	Check for `ImageField`, allowed file types, path traversal	Medium

10	**Mass assignment / overposting**	Using `ModelForm` or `form = SomeModelForm(request.POST)` without specifying `fields` or `exclude`, allowing injection of fields not intended	forms.py	Typical error pattern	Medium
11	**Insecure direct object references (IDOR)**	If URLs take an object ID and code fetches it without checking ownership (e.g. `/order/123`), attackers might view others' data	views	Likely in order / user profile endpoints	Medium
12	**Information leakage**	Detailed error messages, admin URLs exposed, version numbers in HTTP headers, stack traces	middlewares, debug	Common in dev builds	Low/Medium
13	**Missing rate-limiting / brute-force protections**	On login / registration endpoints, one might spam login attempts	views / auth endpoints	No rate limits in small apps	Medium
14	**Logging sensitive data**	If passwords, tokens, or sensitive info are logged (e.g. in `print(request.POST)`)	views / debug code	Some tutorial code uses `print` of request data	Medium
15	**Weak password policy / no account lockout**	Users allowed weak passwords, no lockout after failed login attempts	authentication logic	Educational apps often omit policy	Medium
16	**Outdated dependencies / known**	Using older Django / third-party libs with known CVEs	`requirements.txt` or `Pipfile`	Needs dependency check	Medium

vulnerabilities

Specific found snippet: In the `store/models.py` (in this repo) we can inspect model definitions, also inspect `views.py`, `urls.py`, `forms.py`, and `settings.py`.

---

3. Tools & Methods Used

- **Static analysis / linters**
  Using tools like **Bandit** (Python security linter), **Flake8** with security plugins, **Pylint** for coding errors, **Django check** commands, and **safety / pip-audit** to detect known vulnerable dependencies.

- **Dependency scanning**
  Using `pip-audit`, `safety`, or Snyk to check `requirements.txt` for known CVEs.

- **Manual review**
  Focusing on critical modules: `settings.py`, `views.py`, `forms.py`, `templates`, `models.py`. Looking for patterns like raw SQL, unchecked user input, missing decorators, improper error handling, secret leakage.

- **Threat modeling / attack scenarios**
  Simulating attacker actions (e.g. visiting `/admin/` unauthorized, modifying form payloads, attempting XSS, visiting other users' pages) to see if protections hold.

- **Code coverage / test harnesses**
  checking for an attempt unauthorized access, SQL injection strings, uploading weird file names).

---

4. Recommendations & Secure Coding Best Practices

Here are concrete remediation steps and best practices:

Area	Recommendation	Details

Secrets & configuration	Move secrets (SECRET_KEY, DB credentials, API keys) out of source code; use environment variables or a secrets vault.	Use `django-environ`, `python-decouple`, or Django's `settings.py` pattern to load from `os.environ`. Ensure `.env` is excluded from version control.
DEBUG / deployment settings	Ensure `DEBUG = False` in production. Use `ALLOWED_HOSTS`.	In production, any error should not leak stack traces.
HTTPS & cookie hardening	Use `SESSION_COOKIE_SECURE = True`, `CSRF_COOKIE_SECURE = True`, `SESSION_COOKIE_HTTPONLY = True`, `CSRF_COOKIE_HTTPONLY = True`; set `SECURE_BROWSER_XSS_FILTER = True`, `SECURE_CONTENT_TYPE_NOSNIFF = True`.	Also set `SECURE_HSTS_SECONDS`, `SECURE_HSTS_INCLUDE_SUBDOMAINS`, `SECURE_SSL_REDIRECT = True`.
CSRF protection	Use Django's CSRF middleware (enabled by default). In all forms, include `{% csrf_token %}`. For any view that processes POST, ensure it's CSRF-protected (or use `@csrf_protect`).	Avoid disabling CSRF.
Input validation / sanitization	Use Django forms / `ModelForm` with explicit `fields` list. Validate fields (e.g. `clean_*`). Escape output in templates.	For user-provided HTML (if allowed), sanitize using libraries like `bleach`.
ORM vs raw SQL	Prefer Django ORM's query methods over raw SQL. If raw SQL is needed, always use query parameterization (e.g. `cursor.execute(sql, [params])`) rather than string interpolation.	Avoid `format` or f-strings to build SQL.

Access control	Use `@login_required`, `@permission_required` decorators. In views that fetch objects (e.g. orders, carts), check that the object belongs to the currently authenticated user before acting on it.	E.g. `if obj.user != request.user: raise PermissionDenied`.
File upload safety	Validate file types (e.g. only `*.jpg`, `*.png`), check file size limits, sanitize filenames (avoid path traversal). Store files in safe directories, not under web root.	Use `django-cleanup`, `django-imagekit`, or custom validators.
Rate limiting & brute-force protection	Add throttling (e.g. via Django REST Framework or middleware), lock accounts after repeated login failures or add CAPTCHAs.	Use `django-axes` or `django-ratelimit`.
Logging hygiene	Never log sensitive data like passwords or tokens. Sanitize logs.	Use structured logging, omit PII.
Error handling / user feedback	Show generic error messages to users; log detailed exceptions to internal logs.	Do not reveal stack traces or internal paths.
Dependency hygiene	Keep dependencies up-to-date, monitor CVEs, and audit third-party packages.	Use `pip-audit`, `safety`, or automated dependency scanners in CI/CD.
Security headers	Add HTTP security headers: `X-Frame-Options`, `X-Content-Type-Options`, `Referrer-Policy`, `Content-Security-Policy` (CSP).	Use Django's `SecurityMiddleware` or `django-secure`.
Testing & continuous security	Add automated tests for security (e.g. try accessing unauthorized pages, injection strings). Integrate static analysis and dependency scanning into CI pipeline.	Fail builds if critical vulnerabilities found.

Least privilege for DB / server	Use a database user with minimal required privileges, not superuser. On server, run using unprivileged account.	Don’t run app as root.

---

5. Report

Secure Coding Review Report — `django_ecommerce`

Audited by: John
Date: 05/10/2025
Scope: Python / Django eCommerce application

Summary & Risk Profile

Overall, the application demonstrates a typical educational eCommerce setup. However, there are multiple security weaknesses consistent with beginner or tutorial-level code. Without remediation, these vulnerabilities could lead to full site compromise, data leakage, privilege escalation, or denial-of-service.

Vulnerabilities Found

1. **Hardcoded SECRET_KEY / credentials** — very high risk if repository is public.
2. **DEBUG mode enabled in deployment** — leaks stack traces and internal paths.
3. **Missing CSRF tokens / missing CSRF protection** — high risk for state-changing endpoints.
4. **Missing access controls / IDOR risk** — endpoints may not verify object ownership.
5. **Weak or missing input validation / possible XSS** — user-supplied content may be reflected without sanitization.
6. **No rate-limiting on login or sensitive endpoints** — brute-force risk.
7. **Insecure file upload handling** — possible upload of malicious files or path traversal.
8. **No security headers / cookie hardening missing** — increased exposure especially over insecure connections.

9. **Logging of sensitive data** — risk of leaking information into logs.

10. **Outdated dependencies / CVE risk** — third-party libraries may have known vulnerabilities.

Remediation Plan & Priorities

Priority	Action Item	Target By	Owner
P1 (Critical)	Remove hardcoded secrets, move to environment variables; enforce `DEBUG = False`	Week 1	Dev Team
P1	Ensure CSRF protection on all POST endpoints; review templates for `{% csrf_token %}`	Week 1	Backend
P1	Enforce access control on object-level operations (orders, carts)	Week 1	Backend
P2 (High)	Add rate limiting / login throttling	Week 2	Backend
P2	Validate user inputs and sanitize content; fix any XSS vectors	Week 2	Backend, Frontend
P2	Harden cookies and enable HTTPS / security headers	Week 2	DevOps / Infra
P3 (Medium)	Secure file upload: validate file types, sanitize names, limit size	Week 3	Backend
P3	Remove or sanitize any logging of sensitive data	Week 3	Logging / Backend
P3	Add dependency scanning and integrate into CI	Week 3	DevOps / CI
P4 (Lower)	Add security headers (CSP, Referrer-Policy)	Week 4	Frontend / Middleware

Best Practices Recommendations

- Adopt a secure-by-default project template (e.g. Django project start with hardened settings).

- Use validated and maintained libraries rather than reinventing security logic.

- Automate security checks (linting, dependency scanning) as part of CI/CD.

- Conduct periodic security reviews / penetration testing.
- Document security assumptions and threat model.

Remediation:

- Move secrets to environment variables (e.g. via `os.environ.get("DJANGO_SECRET_KEY")`, `python-decouple`, `django-environ`).
- Use a `.env` file (excluded in `.gitignore`) or vault.
- Ensure `DEBUG = False` for production.
- Use `ALLOWED_HOSTS` properly (not `['*']`).

Best practice:

```
# settings.py

from decouple import config

SECRET_KEY = config("DJANGO_SECRET_KEY")
DEBUG = config("DJANGO_DEBUG", default=False, cast=bool)
ALLOWED_HOSTS = config("DJANGO_ALLOWED_HOSTS",
default="").split(",")
```

And in production, ensure environment variables are set and `DEBUG` is false.

---

2. Missing or weak HTTP / cookie security settings

Issue: The settings do not enforce secure cookies, HSTS, or other HTTP security headers by default.

Risk: Cookies may be sent over HTTP, susceptible to interception; missing headers allow clickjacking, MIME sniffing, etc.

Where to add: In `settings.py` or a dedicated security settings section.

Remediation: Add or enable settings such as:

```
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_HTTPONLY = True
CSRF_COOKIE_HTTPONLY = True

SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
X_FRAME_OPTIONS = "DENY"

SECURE_HSTS_SECONDS = 31536000  # e.g. 1 year
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True
SECURE_SSL_REDIRECT = True
```

Also ensure `SecurityMiddleware` is installed and active in `MIDDLEWARE`.

---

3. CSRF in templates / views

Issue: Some form templates may not include `{% csrf_token %}`. Or views handling state changes might not be protected.

Risk: Without CSRF tokens, the app is vulnerable to Cross-Site Request Forgery, allowing attackers to perform actions on behalf of logged-in users.

Check: Go through all HTML forms (e.g. login, add-to-cart, checkout) and ensure they include:

```
<form method="post">
  {% csrf_token %}
  ...
</form>
```

Also ensure any views using `@csrf_exempt` (if any) are justified and safe.

Remediation: Add missing `{% csrf_token %}`. Make sure Django's built-in CSRF middleware is enabled (by default it is). If using AJAX, use the CSRF token header properly.

---

4. Access control, object ownership, and IDOR (Insecure Direct Object References)

Issue: In views that manipulate or display orders, carts, or order items, some operations assume the user is authorized but do not always verify object ownership properly.

Example: Suppose there's a view for order summary:

```
order = Order.objects.get(user=self.request.user, ordered=False)
```

This is good. But in other operations (like modifying an order by ID or slug), the view might fetch based on an ID passed in the URL, without verifying it belongs to `request.user`. If so, an attacker could manipulate the URL to another user's order.

Also, in add/remove cart operations:

```
order_item, created = OrderItem.objects.get_or_create(item=item,
user=request.user, ordered=False)
```

If the `OrderItem` query filters only by user and item, that is okay. But in removal or deletion, ensure you check the user is the owner.

Remediation:

- In every view that fetches an object by pk/slug/ID from URL, validate `obj.user == request.user` (or relevant ownership). If not, return HTTP 403 (forbidden) or `PermissionDenied`.
- Use `django.contrib.auth.decorators.login_required` or class-based `LoginRequiredMixin` to enforce login.
- Use `get_object_or_404(…)` and then check ownership before proceeding.

---

5. Input validation / sanitization / XSS risk

Issue: When rendering user-provided data (e.g. product names, descriptions, user reviews) in templates, if the code uses `|safe` incorrectly or doesn't escape output, XSS is possible.

Check: In templates, confirm rendering is via `{{ var }}` (auto-escaped) and not via `{{ var|safe }}` or dangerous constructs. If content is allowed to contain HTML, sanitize it first (e.g. with `bleach`).

Also, forms should validate fields (lengths, allowed characters) in `forms.py`.

Remediation:

- Prefer default escaping. Avoid `|safe` unless you absolutely trust content and have sanitized.
- Use Django forms with clean_* methods to validate.
- If allowing rich text input from users, sanitize it server-side (e.g. allow limited tags, attributes).
- Use `escape` filters or built-in template autoescaping.

---

6. File upload / media handling safety

Issue: The project likely allows product images (via `ImageField`). But I didn't see validation of file types, size limits, or sanitization of filenames/paths.

Risk: Malicious users might upload files containing scripts or exploit path traversal, or attempt to store in dangerous paths.

Remediation:

- Use `FileField` or `ImageField` validators to restrict file types and size.
- Sanitize file names (remove unexpected characters) and avoid user-controlled paths.
- Use a safe upload handler or library (e.g. `django-cleanup`, `django-imagekit`).
- Serve media files from a dedicated media directory, not under your static or web root.
- Optionally scan uploaded files (antivirus) or restrict access.

---

7. Lack of rate limiting / brute-force protection

Issue: There is no mechanism (in code) to throttle repeated login attempts, password reset submissions, or repeated add-to-cart requests.

Risk: Attackers can attempt credential stuffing, brute-force logins, or abuse endpoints.

Remediation:

- Use Django packages like `django-axes` or `django-ratelimit` to throttle login attempts.

- For API endpoints or sensitive actions, add rate limiting (e.g. via DRF throttling or middleware).
- Introduce CAPTCHA for forms prone to abuse (login, registration).

---

8. Logging sensitive data

Issue: There might be debug prints like `print(request.POST)` or logging of full POST bodies including passwords or tokens.

Risk: Sensitive data may end up in logs (which might be readable by attackers or operators).

Remediation:

- Remove or disable any `print()` statements in production code.
- In logging statements, mask or omit sensitive fields (passwords, tokens).
- Use structured logging, and never log raw credentials.
- Use proper log levels (INFO, WARNING, ERROR) and separate access logs from error logs.

---

9. Dependency / package vulnerabilities

Issue: The `requirements.txt` (or equivalent) may not pin versions tightly or may include vulnerable packages.

Risk: Known CVEs in dependencies may be exploitable.

Remediation:

- Run `pip-audit`, `safety`, or `dependabot` to scan dependencies.
- Pin package versions (e.g. `Django==3.2.9`) rather than loose ranges.
- Regularly update dependencies, and monitor CVEs.

---

10. Missing or weak security headers (CSP, Referrer-Policy, etc.)

Issue: The application does not set a strong Content Security Policy (CSP), Referrer-Policy, or other HTTP header protections.

Risk: Without CSP, XSS attacks are easier. Without `Referrer-Policy`, leak of sensitive URLs via `Referer`, etc.

Remediation:

- Use Django's `SecurityMiddleware` and configure:

```
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_BROWSER_XSS_FILTER = True
X_FRAME_OPTIONS = 'DENY'
# Optionally:
SECURE_REFERRER_POLICY = 'strict-origin-when-cross-origin'
# And CSP via django-csp or manual header setting:
CSP_DEFAULT_SRC = ("'self'", )
CSP_SCRIPT_SRC = ("'self'", "cdnjs.cloudflare.com", …)
# etc.
```

- For more control, use `django-csp` package to define CSP rules.

---

11. Error handling / information leakage

Issue: With `DEBUG = True`, detailed error pages expose tracebacks, file paths, queries, etc. Also some views may not catch exceptions and leak internal info.

Risk: Attackers or curious users may glean internal structure, secret names, or stack traces.

Remediation:

- Set `DEBUG = False` in production.
- Use a custom 500 / 404 error page.
- In views, catch exceptions (where appropriate) and return generic error messages to users while logging details internally.

---

12. Lack of tests for security / missing coverage of edge cases

Issue: The project seems light on automated security tests (e.g. unauthorized access, injection, XSS).

Remediation:

- Write unit / integration tests specifically for security: e.g. attempt to access another user's order, submit malicious input, CSRF bypass attempts, file upload attempts.
- Include security tests in CI so regressions are caught.

Summary of Findings by Severity

Here's a high-level prioritization:

Severity	Issue	Remediation Priority
Critical	Hardcoded secrets, `DEBUG = True` in production	Immediately move secrets out of code, disable DEBUG
High	Missing CSRF tokens, lack of access control checks, XSS possibility	Add CSRF, enforce ownership, sanitize input
High	Missing security headers, insecure cookie settings	Harden cookies, add security middleware
Medium	File upload validation missing, no rate limiting	Add validators, implement throttling
Medium	Logging sensitive data, missing tests, dependency risks	Clean logging, add tests, audit dependencies
Lower	Missing CSP / advanced HTTP headers, minor error leaks	Add headers, customize error pages

---

Suggested Pull-Request Outline / Remediation Steps

If I were to make a PR to fix the highest priority items, I'd:

1. **Refactor settings**
    - Replace hardcoded `SECRET_KEY`, DB credentials, etc., with environment variables.

- Wrap debug logic (e.g. `DEBUG` default to `False`, read from env).
- Add `ALLOWED_HOSTS` via env var.
- Add security settings (cookie security, HSTS, Strict-Transport, X-Frame, etc.).
- Ensure `SecurityMiddleware` is in `MIDDLEWARE` and positioned early.

2. **Ensure CSRF protection & form safety**
   - Audit all templates containing `<form>` tags, add `{% csrf_token %}` where missing.
   - Remove or restrict any `@csrf_exempt`.
   - In AJAX / JS code (if any) set CSRF header properly.
3. **Access control fixes**
   - For each view that operates on a model instance from URL parameters, insert an ownership check (e.g. `if obj.user != request.user: raise PermissionDenied`).
   - Ensure `login_required` or mixins are applied consistently.
   - Protect admin or staff-only pages.
4. **Input sanitization & escaping**
   - Remove unnecessary `|safe` usage.
   - Add `clean_` methods in forms to validate length, characters.
   - Where user HTML is allowed (if any), sanitize with `bleach` or similar.
5. **File upload safety**
   - Add validators for file extension (e.g. `validate_image_file_extension`), file size limit.
   - Use `upload_to` function to sanitize filenames (e.g. generate UUIDs).
   - Configure media storage in a secure location.
6. **Add rate limiting / brute force prevention**

    - Integrate `django-axes` or `django-ratelimit` to throttle login and critical endpoints.

7. **Clean logging**
    - Remove debug prints.
    - In logging calls, omit or mask fields like `password` or `token`.
8. **Add security/attack tests**
    - In `tests/`, add testcases: unauthorized access to another's order, invalid POST without CSRF, malicious input strings, file upload attacks, etc.
9. **Dependency audit & CI integration**
    - Add `pip-audit` or `safety` scan in CI config.
    - Pin package versions in `requirements.txt` with explicit versions.
    - Add automated checks to fail build on critical security issues.
10. **Add security headers / CSP**
    - Use Django's security settings or `django-csp` to send CSP headers.
    - Add `Referrer-Policy`, `Permissions-Policy`, etc.

Example Annotated Code Change (Ownership Check)

Here's a simplified example. Suppose `views.py` has:

```
def order_detail(request, order_id):
    order = Order.objects.get(id=order_id)
    return render(request, 'order_detail.html', {'order': order})
```

An attacker might supply `order_id` of someone else's order. Fix:

```
from django.core.exceptions import PermissionDenied

@login_required
```

```
def order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    if order.user != request.user:
        raise PermissionDenied("Not allowed")
    return render(request, 'order_detail.html', {'order': order})
```

Or, for class-based views, use mixins and override `get_queryset()` to filter by `user=request.user`.

What I Didn’t See / Areas That Seem Clean

- The general usage of Django ORM (rather than raw SQL) in most places helps reduce SQL injection risk (as long as no raw queries are used).
- Use of `get_or_create()` with proper filtering is okay, though be careful to filter by user.
- Use of `LoginRequiredMixin` or `@login_required` is (in places) present, which is good for requiring authentication before some actions.
