Secure E-commerce Website
Vulnerability Assessment & DevSecOps Automation

Stanley Shaw

5th July 2025

github.com/stanly363/Secure-ecomerse-website

Abstract

This report presents a comprehensive security and DevSecOps evaluation of the "SecureCart" application. In Part A, I document my static and dynamic security assessments, detailing the tools used, the vulnerabilities discovered, and the remediation strategies applied. In Part B, I describe the design and implementation of a CI/CD pipeline, explaining how security testing was integrated into the automated build and deployment process. Throughout the report, I reflect on the challenges faced, solutions implemented, and key lessons learned in building a secure, automated development workflow.

Contents

1 Introduction

This report highlights a full DevSecOps workflow for the Secure Cart application. I first automated vulnerability discovery with OWASP ZAP for dynamic testing, and used Snyk for dependency and code analysis. I then verified each issue, applied code or configuration fixes, and reran the scanners to confirm closure. To ensure those controls remain effective, I embedded these same tools into a GitHub Actions pipeline that installs dependencies, runs unit tests, executes snyk tests with strict severity thresholds, launches a headless ZAP baseline scan, and deploys to a staging server only if every security gate passes—thereby preventing vulnerable code or packages from reaching production.

2 Part A — Vulnerability Assessment

This section documents my systematic security review of the running application. OWASP ZAP was ran on the site to uncover runtime weaknesses, I then validated and prioritised each finding before implementing code-or configuration-level fixes and re-scanning to confirm that the vulnerabilities were fully mitigated [OWASP, 2025].

2.1 OWASP ZAP Findings

OWASP ZAP was configured in "spider → passive-scan → active-scan" mode against the localhost instance running over HTTPS [OWASP, 2022]. Two findings stood out because they affect every page of the site and significantly widen the attack surface: an absent `Content-Security-Policy` header [Django CSP, 2024] and an outdated Bootstrap runtime library [Bootstrap Team, 2024]. The sections below summarise the risk, evidence and remediation for each.

2.1.1 Missing Content-Security-Policy Header

During ZAP's passive crawl of the application, every HTML response—beginning with the login page—was returned without a `Content-Security-Policy` header. This omission leaves modern browsers with no instructions on which external scripts, styles, or frames should be trusted.

Risk

Without a `Content-Security-Policy` header the browser imposes no restrictions on script, style, image, frame or connect sources [OWASP CSP, 2025]. An attacker who can inject a single `<script>` tag—via reflected or stored XSS, compromised CDN, or a malicious third-party ad—can execute arbitrary JavaScript in every shopper's session, enabling credential theft, card-skimming, or full account takeover.

Evidence

Figure 1 shows ZAP's response for `/accounts/login/`. The missing `Content-Security-Policy` triggers ZAP's medium-severity warning (CWE-693), indicating the page is exposed to content-injection risks.

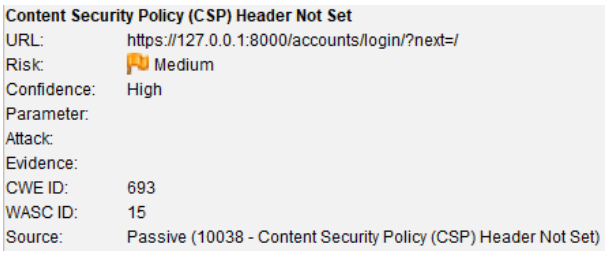

Figure 1: ZAP evidence – no `Content-Security-Policy` in the response

Remediation

In order to fix this, I installed `django-csp`, inserted `csp.middleware.CSPMiddleware` near the top of the `MIDDLEWARE` list in `settings.py`, and declared a restrictive policy, as shown in Figure 2 [Django CSP, 2024].

```
CONTENT_SECURITY_POLICY_REPORT_ONLY = {
    "DIRECTIVES": {
        "default-src": ("'self'",),
        "script-src": ("'self'", "https://cdn.jsdelivr.net"),
        "style-src":  ("'self'", "https://cdn.jsdelivr.net"),
        "img-src":    ("'self'", "data:"),
        "object-src": ("'none'",),
        "frame-ancestors": ("'none'",),
        "report-uri": "/csp-report/",
    }
}
```

Figure 2: `Content-Security-Policy` django implementation code

All pages now post with a `Content-Security-Policy` header, closing the gap ZAP flagged and protecting the application against XSS, data-injection, and framing attacks.

```
HTTP/1.1 301 Moved Permanently
Date: Tue, 13 May 2025 17:39:38 GMT
Server: WSGIServer/0.2 CPython/3.12.10
Content-Type: text/html; charset=utf-8
Location: /product/7/
Vary: Cookie
Content-Security-Policy-Report-Only: style-src 'self' https
https://cdn.jsdelivr.net; default-src 'self'; report-uri /c
Strict-Transport-Security: max-age=31536000; includeSubDoma
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin
Set-Cookie: sessionid=""; expires=Thu, 01 Jan 1970 00:00:00
Connection: close
```

Figure 3: ZAP evidence – `Content-Security-Policy` present in the response

Impact Reduction.

With the policy enforced, any injected script from an external domain is blocked; inline scripts are disallowed unless whitelisted via nonce/hashes, closing the most common exploit path for card-skimming malware in e-commerce sites.

2.1.2 Out-of-date Bootstrap 4.3.1 Library

During ZAP's passive analysis, Retire.js [Retire.js Community, 2020] reported that the application `bootstrap.bundle.min.js` version 4.3.1 was outdated, and known to be vulnerable [Bootstrap Team, 2024].

Risk

Bootstrap 4.3.1 is affected by two medium-severity XSS vulnerability's:

- **CVE-2019-8331 – Tooltips / Popovers.** Whatever you put in the *data-template* field is copied straight onto the page [National Vulnerability Database, 2025b]. If an attacker slips in something like <img onerror=alert(1)>, it fires when anyone hovers the tooltip.
- **CVE-2018-20677 – Affix.** The old Affix plugin does the same with its *data-target* field. A harmful value there runs as soon as the page loads or scrolls [National Vulnerability Database, 2025a].

If untrusted data is echoed into those attributes an attacker can inject arbitrary HTML/JavaScript, leading to session theft or full account compromise.

Evidence

Figure 4 shows ZAP Retire.js flagging the vulnerability together with providing direct links to both CVEs.

Figure 4: Retire.js flags Bootstrap 4.3.1 with two XSS CVEs

Remediation

To fix this, I upgraded to Bootstrap 5.3.6 and added sub-resource integrity (SRI) to the CDN link ensuring that the correct CDN is loaded:

1. Updated bootstrap to version 5.3.6.
2. Added `integrity="sha384-..."` and `crossorigin="anonymous"` to the CDN script tag.

The updated Bootstrap integration with SRI is shown in Figure 5, demonstrating the inclusion of the integrity and crossorigin attributes to protect against CDN tampering [Bootstrap Team, 2024].

```
<!-- Bootstrap JS -->
<script src="{% static 'js/bootstrap.bundle.min.js' %}"
    integrity="sha384-ONB+SnjmmjdZc5tpXzfdRtB4ElGyA1Tw2BdvWVEu6xE7W8uXxAbBxgLg0bEU9jvc"
    crossorigin="anonymous"></script>
```

Figure 5: SRI integrity integration

The updated file hash now passes Retire.js, and the correct updated version of `bootstrap` is loaded as shown in Figure 6.

```
/*!
  * Bootstrap v5.3.6 (https://getbootstrap.com/)
  * Copyright 2011-2025 The Bootstrap Authors (https://github.com/twbs/boots
  * Licensed under MIT (https://github.com/twbs/bootstrap/blob/main/LICENSE)
  */
```

Figure 6: Updated Bootstrap version 5.3.6 now used

Impact reduction

With the library patched, inline scripts nonce-protected, and server-side sanitisation maintained via `django`, the attack surface for tooltip-based XSS is effectively closed.

2.2 Snyk Dependency Findings

2.2.1 Out-of-date Django 4.2.16 Version

During Snyk's dependency scan of the `requirements.txt` file, Django version 4.2.16 was flagged as vulnerable to multiple high and critical severity issues [Snyk Ltd., 2025].

Risk

Django 4.2.16 has the following known vulnerabilities:

- **Command Injection (High Severity).** Vulnerable template filters or management commands could allow attackers to execute system commands if user input is not properly handled. This could result in complete system compromise [National Vulnerability Database, 2024a].
- **Resource Allocation Without Limits (High Severity).** Django does not enforce limits on certain operations, such as file uploads or request processing, which could lead to denial of service if an attacker forces the server to handle excessive load [National Vulnerability Database, 2025c].
- **SQL Injection (Critical Severity).** Django's ORM in version 4.2.16 may allow attackers to inject malicious SQL through untrusted data if query construction is not properly sanitized, risking full database compromise [National Vulnerability Database, 2024b].

Evidence

Figure 7 shows Snyk flagging the Django package with three identified security risks, including one critical SQL injection vulnerability.

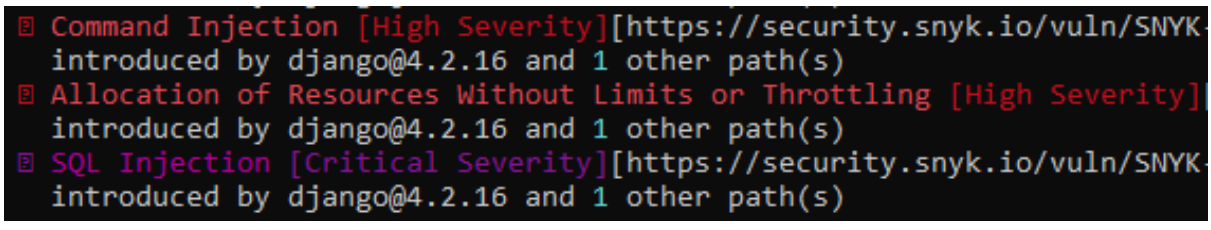

Figure 7: Snyk flags Django 4.2.16 with multiple high and critical severity issues

Remediation

In order to patch these vulnerabilities, I upgraded Django to the latest patched release to resolve these issues. The updated `requirements.txt` now specifies Django 5.1.9, as shown in Figure 8 and the snyk dependency scan highlights no vulnerabilities (see Figure 9 [Django Software Foundation, 2025a].

```
asgiref==3.8.1
certifi==2024.12.14
charset-normalizer==3.4.1
colorama==0.4.6
Django==5.1.9
```

Figure 8: Updated `requirements.txt` specifying Django 5.1.9

```
Organization:      stanly363
Package manager:   pip
Target file:       requirements.txt
Project name:      securecart
Open source:       no
Project path:      C:\Users\stanl\Downloads\Secure-ecomerse-website-main
Licenses:          enabled

✔ Tested 18 dependencies for known issues, no vulnerable paths found.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.
```

Figure 9: Successful snyk dependency scan highlighting no known vulnerabilities

Impact reduction

By upgrading Django to the latest version, the risk of command injection, denial of service, and SQL injection is significantly reduced. This ensures the application runs on a secure, supported framework with up-to-date security patches.

2.3 Theoretical Vulnerabilities

Django's secure-by-default framework helps prevent many common vulnerabilities such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), SQL Injection, and Broken Authentication by providing built-in protections. Features like CSRF tokens, automatic input sanitization in templates, ORM query parameterization, and strong password hashing with PBKDF2 reduce the attack surface significantly. As a result, during manual review, no additional confirmed major vulnerabilities beyond those identified by automated tools were found as seen in Figure 10.

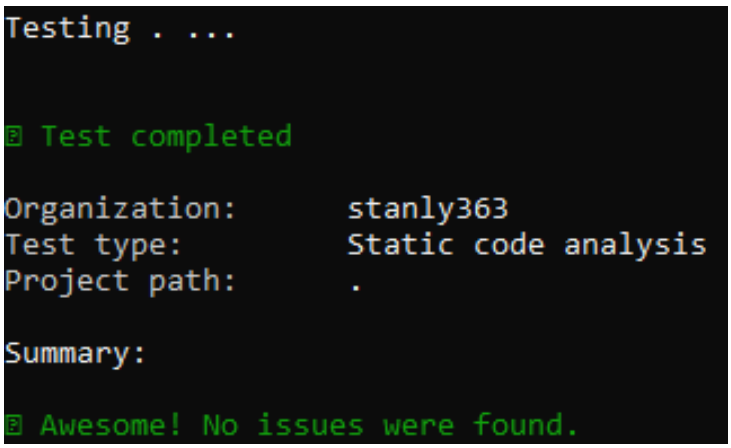

Figure 10: Successful snyk scan highlighting no known vulnerabilities

However, two theoretical vulnerabilities based on design patterns and common developer mistakes are worth noting:

2.3.1 Insufficient Object-Level Authorisation Checks

Django provides model-level and view-level permission checks, but it does not enforce object-level access control by default. Developers are responsible for validating that a user has permission to access or modify a specific object instance.

Risk

If object-level checks are missing, a user could potentially manipulate URLs or form data to view, edit, or delete objects that they do not own. This could include orders, user profiles, or administrative records [OWASP Cheat Sheet, 2024a].

Example Vulnerable Code

Figure 12 shows a vulnerable implementation where object ownership is not checked, allowing attackers to access other users' orders by modifying the URL.

```python
from django.shortcuts import get_object_or_404, render
from .models import Order

def view_order(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request, 'order_detail.html', {'order': order})
```

Figure 11: Vulnerable view function missing user ownership validation

Example Scenario

A user changes a URL like `/orders/5` to `/orders/6` and successfully accesses another user's order details because no object-level permission check is enforced in the view.

Remediation

Use Django's `get_object_or_404` or custom access checks combined with `request.user` validation to ensure that the logged-in user owns the object being accessed. Implement `check_object_permissions()` or similar logic to enforce object-level security [Django Software Foundation, 2025b].

```python
from django.shortcuts import get_object_or_404, render
from .models import Order

def view_order(request, order_id):
    order = get_object_or_404(Order, id=order_id, customer=request.user)
    return render(request, 'order_detail.html', {'order': order})
```

Figure 12: Fixed view function including user ownership validation

2.3.2 Insecure Custom Script Execution (Code Injection Risk)

While Django itself provides secure execution environments, developers sometimes introduce custom admin tools, management commands, or evaluation scripts that rely on `eval()`, `exec()`, or `importlib.import_module()` using user-supplied or environment-provided input [OWASP Cheat Sheet, 2024b].

Risk

If user input or environment variables are passed into dangerous Python functions like `exec()` or `eval()` without proper validation, attackers could inject and execute arbitrary Python code on the server.

Example code

Figure 13 shows a vulnerable implementation that uses `exec()` on user-supplied input, allowing an attacker to execute arbitrary Python commands.

```python
from django.http import HttpResponse

def run_code(request):
    user_code = request.GET.get('code')
    exec(user_code)
    return HttpResponse("Code executed.")
```

Figure 13: Vulnerable code using `exec()` on untrusted user input

Example Scenario

A developer writes a custom feature that loads modules dynamically based on user input or environment variables without proper validation. An attacker manipulates this input to load malicious modules or execute arbitrary commands. Figure 13 shows a vulnerable implementation that uses `exec()` on user-supplied input, allowing an attacker to execute arbitrary Python commands.

Remediation

Avoid using `eval()`, `exec()`, or dynamic imports with untrusted input. Validate and whitelist any allowed module or command names, and use safer alternatives such as dictionary lookups or predefined mappings [OWASP Cheat Sheet, 2024b]. Figure 14 shows a fixed implementation that restricts execution to predefined safe commands using a dictionary lookup.

```python
from django.http import HttpResponse

SAFE_COMMANDS = {
    'ping': lambda: 'pong',
    'status': lambda: 'All systems operational',
}

def run_code(request):
    command = request.GET.get('code')
    result = SAFE_COMMANDS.get(command, lambda: 'Invalid command')()
    return HttpResponse(result)
```

Figure 14: Secure code using a dictionary lookup to limit allowed commands

These theoretical risks emphasise the need for secure coding practices, thorough code reviews, and awareness of Django's security model to ensure robust protection beyond built-in defaults.

3 CI/CD Pipeline Implementation

3.1 Implementing CI/CD Pipelines

My CI/CD pipeline for SecureCart is implemented using GitHub Actions, in line with DevSecOps best practice guidelines to automate the build, test, and deployment stages [GitHub Docs, 2025]. I began by forking the codebase from into a dedicated `ci-cd` branch and creating a `.github/workflows/ci-cd.yml` file. My key motivations for adopting CI/CD include:

- **Rapid feedback:** Automated checks on each commit surface failures immediately, reducing the time between defect introduction and detection.
- **Reproducibility:** Defining the steps as code ensures every runner executes the same sequence, eliminating "works on my machine" issues.
- **Developer confidence:** A passing pipeline signals stability, enabling more frequent merges and shorter release cycles.

3.2 Pipeline Overview

The pipeline consists of four major phases: *Source Control*, *Build & Test*, *Security Scanning*, and *Deployment*. Each phase corresponds to one or more jobs in GitHub Actions. The pipeline is triggered by both `push` and `pull_request` events, specifically when targeting the `main` branch, as shown in Figure 15. In the following sections, I walk through each stage of my GitHub Actions workflow in turn—showing how I trigger the pipeline, install dependencies, run tests and linters, perform security scans, build and run the Docker image, and finally deploy to production.

Source Control Trigger

All pipeline runs are triggered on both `push` and `pull_request` events, but only when targeting the `main` branch. Figure 15 shows the GitHub Actions UI confirming these settings.

```yaml
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

Figure 15: GitHub Actions "on: [push, pull_request]" filtered to "branches: [main]".

Dependency Installation

Dependencies are installed via a two-step script that first upgrades `pip` and then installs from `requirements.txt`, ensuring reproducible builds [PIP Team, 2025]. See Figure 16 for the workflow screenshot.

```yaml
- name: Checkout repository
  uses: actions/checkout@v3

- name: Set up Python
  uses: actions/setup-python@v4
  with:
    python-version: '3.11'

- name: Install dependencies
  run: |
    pip install --upgrade pip
    pip install -r securecart/requirements.txt
```

Figure 16: "Install dependencies" step in GitHub Actions.

Unit Testing & Linting

Code style is enforced with `flake8`, and business logic validated via Django's test suite [Flake8 Development Team, 2025]. Figure 17 shows both the lint and test steps in the workflow. While Figures 18, 19, and 20 show the unit tests that are run by the pipeline [Django Software Foundation, 2025c].

```yaml
- name: Run Linter (flake8)
  run: flake8 .

- name: Run Unit Tests
  run: python securecart/manage.py test store --keepdb
```

Figure 17: "Lint with flake8" and "Run Django tests" steps.

```python
def test_cart_item_total_price(self):
    item = self.cart.items.first()
    self.assertEqual(item.get_total_price(), 20.00)

def test_cart_total_price(self):
    self.assertEqual(self.cart.get_total_price(), 20.00)

def test_order_total(self):
    self.assertEqual(self.order.get_total(), 30.00)
```

Figure 18: Tests verifying item, cart, and order total price computations.

```python
def test_totp_secret_length(self):
    secret = generate_totp_secret()
    self.assertEqual(len(secret), 32)

def test_totp_uri_and_qr(self):
    user = User.objects.create_user(
        'bob', 'bob@example.com', 'pw'
    )
    secret = generate_totp_secret()
    uri = get_totp_uri(user, secret)
    # percent-encoded email is expected
    self.assertIn(quote(user.email), uri)
    img_b64 = generate_qr_code(uri)
    # should be valid base64 PNG header
    header = base64.b64decode(img_b64)[:8]
    self.assertTrue(header.startswith(b'\x89PNG\r\n\x1a\n'))

def test_verify_totp(self):
    secret = generate_totp_secret()
    import pyotp
    totp = pyotp.TOTP(secret)
    token = totp.now()
    self.assertTrue(verify_totp(token, secret))
```

Figure 19: Tests checking secret length, URI/QR code generation, and token verification.

```python
def test_user_register_form_valid(self):
    data = {
        'username': 'charlie',
        'email': 'charlie@example.com',
        'password1': 'complexpass123',
        'password2': 'complexpass123',
    }
    form = UserRegisterForm(data)
    self.assertTrue(form.is_valid())

def test_user_register_form_mismatch(self):
    data = {
        'username': 'charlie',
        'email': 'charlie@example.com',
        'password1': 'a',
        'password2': 'b',
    }
    form = UserRegisterForm(data)
    self.assertFalse(form.is_valid())
    self.assertIn('password2', form.errors)
```

Figure 20: Tests validating correct input and detecting password mismatches.

Static Application Security Testing (SAST)

A Snyk SAST scan runs against `requirements.txt`, flagging vulnerable dependencies and insecure code patterns [Snyk Ltd., 2025] (See Figure 21).

```yaml
- name: Run Snyk SAST Scan
  uses: snyk/actions/python@master
  with:
    args: --file=securecart/requirements.txt --skip-unresolved
```

Figure 21: Snyk SAST scan step in GitHub Actions.

Build Docker Image with Layer Caching

After verifying that tests and linting have passed, I package my Django app into a Docker image using the project's Dockerfile [Docker, Inc., 2024]. Figure 22 highlights the Dockerfile used to build the docker container. While, Figure 23 illustrates the build process, which leverages docker/build-push-action with layer caching to significantly reduce build times on subsequent runs.

```dockerfile
# syntax=docker/dockerfile:1
FROM python:3.11-slim

WORKDIR /app

RUN apt-get update \
 && apt-get install -y --no-install-recommends \
      libpq-dev gcc build-essential openssl \
 && rm -rf /var/lib/apt/lists/*

COPY requirements.txt ./
RUN pip install --upgrade pip \
 && pip install --no-cache-dir -r requirements.txt

COPY . .

RUN python manage.py collectstatic --noinput

RUN openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem \
    -days 365 -nodes -subj "/CN=localhost"

EXPOSE 8000
ENV DJANGO_SETTINGS_MODULE=securecart.settings \
    PYTHONUNBUFFERED=1

CMD ["gunicorn","securecart.wsgi:application",\
     "--bind","0.0.0.0:8000",\
     "--workers","3",\
     "--config","gunicorn_conf.py",\
     "--certfile","cert.pem",\
     "--keyfile","key.pem"]
```

Figure 22: Dockerfile used to create the application image.

```yaml
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3

- name: Build Docker Image with Cache
  uses: docker/build-push-action@v5
  with:
    context: ./securecart
    file: ./securecart/Dockerfile.ci
    tags: securecart-app:latest
    push: false
    load: true
    cache-from: type=gha
    cache-to: type=gha,mode=max
```

Figure 23: Executing `docker build` to create the application image.

Run Docker Container

With the image ready, I create up a detached container bound to port 8000. As displayed in Figure 24, this command sets environment varirables and launches the service so I can interact with a live instance without polluting my local environment [Docker, Inc., 2025b].

```yaml
- name: Run Docker Container
  env:
    SECRETKEY: ${{ secrets.SECRETKEY }}
    DB_NAME: defaultdb
    DB_USER: avnadmin
    DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
    DB_HOST: ${{ secrets.DB_HOST }}
    DB_PORT: 18833
    STRIPE_PUBLISHABLE_KEY: ${{ secrets.STRIPE_PUBLISHABLE_KEY }}
    STRIPE_SECRET_KEY: ${{ secrets.STRIPE_SECRET_KEY }}
    EMAIL: ${{ secrets.EMAIL }}
    EMAIL_PASSWORD: ${{ secrets.EMAIL_PASSWORD }}
  run: |
    docker run -d \
      -p 8000:8000 \
      --name securecart-app-container \
      -e SECRETKEY="$SECRETKEY" \
      -e DB_NAME="$DB_NAME" \
      -e DB_USER="$DB_USER" \
      -e DB_PASSWORD="$DB_PASSWORD" \
      -e DB_HOST="$DB_HOST" \
      -e DB_PORT="$DB_PORT" \
      -e STRIPE_PUBLISHABLE_KEY="$STRIPE_PUBLISHABLE_KEY" \
      -e STRIPE_SECRET_KEY="$STRIPE_SECRET_KEY" \
      -e EMAIL="$EMAIL" \
      -e EMAIL_PASSWORD="$EMAIL_PASSWORD" \
      securecart-app
```

Figure 24: Starting the container in detached mode on port 8000.

Wait for App to Be Ready

To ensure the application is initialised, the pipeline actively polls the service before scanning. A 'curl' script awaits a successful response from the container, as shown in Figure 25.

```
- name: Wait for App to be Ready
  run: |
    echo "Waiting for application to be ready..."
    timeout 180s bash -c 'until curl -k --output /dev/null --silent --head --fail https://localhost:8000; do
      printf "."
      sleep 5
    done'
```

Figure 25: Actively polling the container to confirm it is initialised.

Dynamic Application Security Testing (DAST)

Once the app is up at `https://localhost:8000`, I initiated a full OWASP ZAP scan to uncover runtime vulnerabilities [OWASP, 2022]. The configuration for this active scan is depicted in Figure 26.

```
- name: Run OWASP ZAP DAST Scan
  uses: zaproxy/action-full-scan@v0.12.0
  with:
    target: 'https://localhost:8000'
```

Figure 26: OWASP ZAP full-scan step against the running container.

Collect Docker Logs

To aid in troubleshooting, I capture the container's stdout and stderr into `django_debug.log` [Docker, Inc., 2025a]. Figure 27 shows the logging command that preserves any error traces for later review.

```
- name: Collect Docker Logs
  run: docker logs securecart-app-container > django_debug.log
```

Figure 27: Redirecting container logs into a file for post-scan analysis.

Upload Logs

Regardless of success or failure, I upload `django_debug.log` as an artefact so it's accessible in the Actions UI [GitHub Actions Team, 2025]. The upload step is captured in Figure 28.

```yaml
- name: Upload Logs
  if: always()
  uses: actions/upload-artifact@v4
  with:
    name: docker-logs
    path: ./django_debug.log
```

Figure 28: Using `actions/upload-artifact` to store the log file.

Stop Docker Container

To clean up the runner environment, I terminate the container once logs are stored [Docker, Inc., 2023]. Figure 29 demonstrates the stop command in the workflow.

```yaml
- name: Stop Docker Container
  run: docker stop securecart-app-container
```

Figure 29: Stopping the Docker container to free up resources on the runner.

Deploy to Production

When all previous steps succeed on the `main` branch, I then trigger the production deployment using a Render deploy hook [Render, Inc., 2025]. The hook URL is stored as the GitHub secret `RENDER_DEPLOY_HOOK`, and the final workflow step sends a HTTPS POST to that endpoint. This initiates a zero-downtime release which builds the image from the production `Dockerfile`. This version serves standard HTTP traffic to work behind Render's reverse proxy, as opposed to the separate `Dockerfile.ci` used for HTTPS-based testing earlier in the pipeline. The Render integration performs the following tasks:

- Builds the latest production Docker image from `securecart/Dockerfile`.
- Applies production environment variables (e.g. `DJANGO_SECRET_KEY`, `DATABASE_URL`, `STRIPE_SECRET_KEY`).
- Runs health checks, swaps traffic to the new container, and retains previous releases for one-click rollback.

Figure 30 shows the conditional GitHub Actions step executing the Render deploy hook, and Figure 31 shows the corresponding API call to render.

```yaml
- name: Trigger Render Deploy Hook
  if: github.ref == 'refs/heads/main' && success()
  env:
    RENDER_DEPLOY_HOOK: ${{ secrets.RENDER_DEPLOY_HOOK }}
  run: |
    echo "👷 Triggering Render deploy…"
    curl --fail --silent --show-error -X POST "$RENDER_DEPLOY_HOOK"
    echo "✅ Deploy hook fired."
```

Figure 30: Conditional production deployment step executed for `main`.

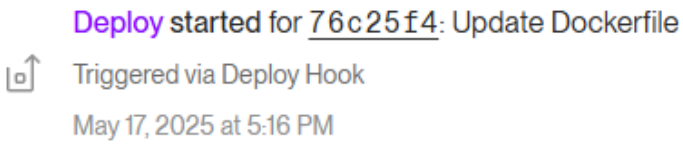

Figure 31: Render API log confirming the Render deploy hook trigger.

Finally, Figure 32 presents the live production landing page hosted using Render at `https://securecart-staging.onrender.com/`; as the application is on a free hosting tier, the service may take a moment to load if it has been inactive.

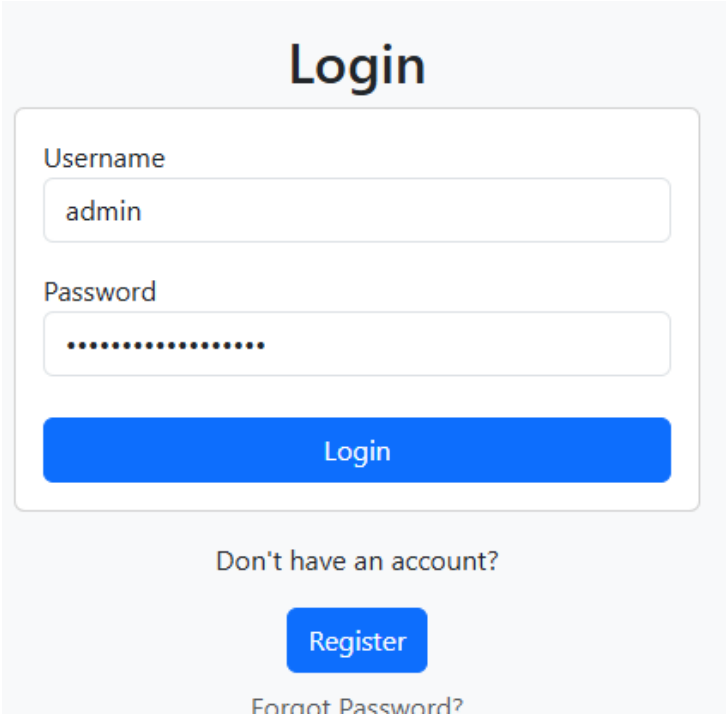

Figure 32: SecureCart production landing page on Render.

3.3 Summary

The final CI/CD pipeline automates building, testing, securing, and deploying the Django application. After dependencies are installed, the code undergoes linting, unit testing, and security scanning (SAST and DAST). The application is then packaged into a Docker container, with logs captured for troubleshooting. Finally, after successful tests and scans, the application is deployed to production with zero downtime. This streamlined process ensures that only secure, tested code reaches production. Figure 33 illustrates the completed pipeline, highlighting each of the steps involved.

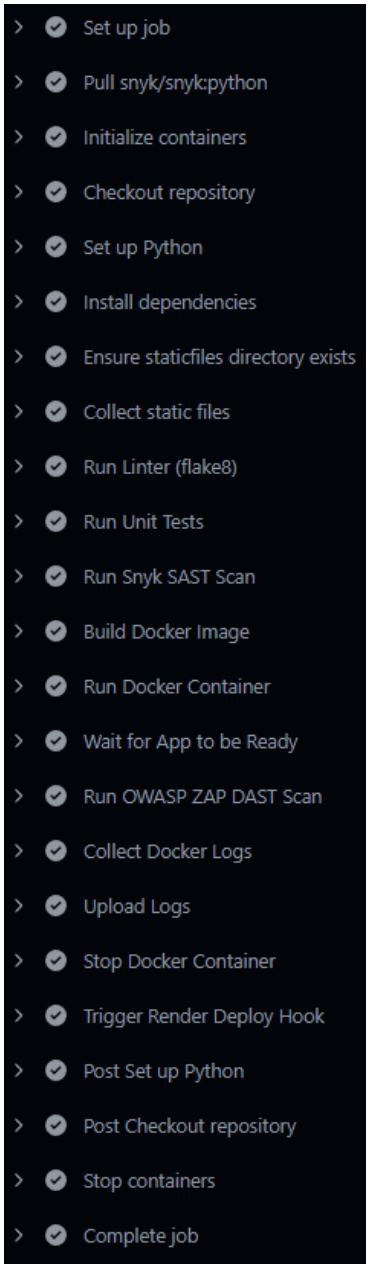

Figure 33: Overview of the completed CI/CD pipeline.

3.4 CI/CD Pipeline Security Testing Reflection

Differences in vulnerabilities

During OWASP ZAP's pre-deployment scan, missing security headers (`Content-Security-Policy`) and an outdated bootstrap library were flagged. Post-deployment, additional issues emerged:

- **Forms blocked by CSP sandbox**: the `sandbox` directive lacked `allow-forms`, causing all form submissions (login/register) to be silently blocked by the browser [Mozilla Contributors, 2024].
- **SRI mismatch on Bootstrap**: the `integrity` attribute on `bootstrap.bundle.min.js` did not match the served file, so the script was blocked entirely [Mozilla Contributors, 2025b].
- **Wildcard CORS on static assets**: `Access-Control-Allow-Origin:*` on CSS responses exposed static content to any origin, violating least-privilege CORS policy [Mozilla Contributors, 2025a].

3.5 Post-deployment Vulnerability Remediation

To address these post-deployment findings, I implemented the following remediations in order to fix the vulnerabilities:

3.5.1 Forms Blocked by CSP Sandbox

The browser refused all form submissions because the CSP `sandbox` directive omitted `allow-forms`, as shown by the network error blocking the login form (Figure 34), and the missing header screenshot (Figure 35). After adding `allow-forms`, the forms post normally (Figure 36) [Mozilla Contributors, 2024].

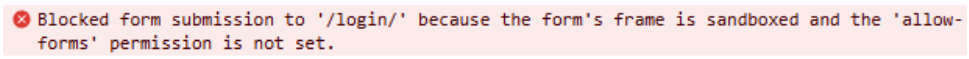

Figure 34: Network error blocking form submissions

```
CONTENT_SECURITY_POLICY = {
    'DIRECTIVES': {
        'default-src':     [SELF],
        'script-src':      [SELF, 'https://cdn.jsdelivr.net'],
        'style-src':       [SELF, 'https://cdn.jsdelivr.net'],
        'img-src':         [SELF, 'data:'],
        'object-src':      [NONE],
        'base-uri':        [SELF],
        'form-action':     [SELF],
        'frame-ancestors': [NONE],
        'sandbox': ['allow-scripts', 'allow-same-origin'],
        'report-to':       ['csp-reports'],
    },
}
```

Figure 35: Before: CSP `sandbox` missing `allow-forms`, blocking forms

```
CONTENT_SECURITY_POLICY = {
    'DIRECTIVES': {
        'default-src':     [SELF],
        'script-src':      [SELF, 'https://cdn.jsdelivr.net'],
        'style-src':       [SELF, 'https://cdn.jsdelivr.net'],
        'img-src':         [SELF, 'data:'],
        'object-src':      [NONE],
        'base-uri':        [SELF],
        'form-action':     [SELF],
        'frame-ancestors': [NONE],
        'sandbox': ['allow-scripts', 'allow-same-origin', 'allow-forms'],
        'report-to':       ['csp-reports'],
    },
}
```

Figure 36: After: CSP `sandbox` updated with `allow-forms`, forms allowed

3.5.2 SRI Mismatch on Bootstrap

The `integrity` hash for `bootstrap.bundle.min.js` did not match the file version, so the script was blocked (Figures 37 and 38) [Mozilla Contributors, 2025b]. Upgrading to Bootstrap 5.3.6 and using the correct SHA-384 hash allowed the script load (Figure 39).

Figure 37: SRI hash mismatch HTML error

```
<script src="{% static 'js/bootstrap.bundle.min.js' %}"
        integrity="sha384-ONB+SnjmmjdZc5tpXzfdRtB4ElGyA1Tw2BdvWVEu6xE7W8uXxAbBxgLg0bEU9jvc"
        crossorigin="anonymous"></script>
```

Figure 38: Before: SRI hash mismatch blocking Bootstrap 5.3.6 bundle

```
<script src="{% static 'js/bootstrap.bundle.min.js' %}"
        integrity="sha384-j1CDi7MgGQ12Z7Qab0qlWQ/Qqz24Gc6BM0thvEMVjHnfYGF0rmFCozFSxQBxwHKO"
        crossorigin="anonymous"></script>
```

Figure 39: After: updated to Bootstrap 5.3.6 with matching integrity hash

3.5.3 Strict CORS on Static Assets

A post-deployment ZAP scan revealed that WhiteNoise [Willison, 2025] was serving static assets with `Access-Control-Allow-Origin:*` (see Figure 40). I therefore updated `securecart/whitenoise_headers.py` (Figure 41) to restrict `Access-Control-Allow-Origin` to my secure domain (see Figure 42) [Mozilla Contributors, 2025a].

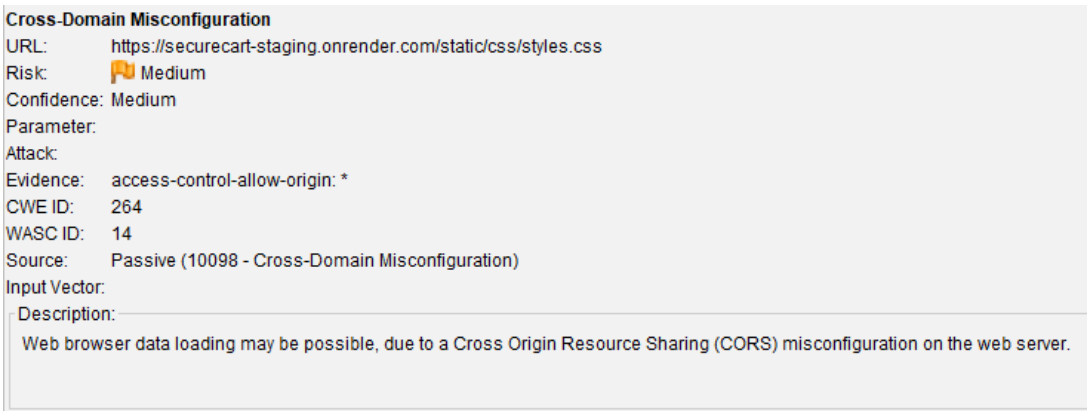

Figure 40: The ZAP warning highlights the Cross-Domain Misconfiguration

```
def add_security_headers(headers, path, url):
    headers['Strict-Transport-Security'] = (
        'max-age=31536000; includeSubDomains; preload'
    )
    if 'Server' in headers:
        del headers['Server']
    return headers
```

Figure 41: The code before injecting a restrictive CORS header

```
def add_security_headers(headers, path, url):
    headers['Strict-Transport-Security'] = (
        'max-age=31536000; includeSubDomains; preload'
    )
    if 'Server' in headers:
        del headers['Server']
    headers['Access-Control-Allow-Origin'] = (
        'https://securecart-staging.onrender.com'
    )
    return headers
```

Figure 42: The code after injecting secure `Access-Control-Allow-Origin`

4 Lessons Learned

Through the SecureCart DevSecOps journey, several key insights emerged that highlight how embedding security into every phase of development enhances both code quality and delivery speed.

4.1 Early Integration of Security (Shift-Left)

Integrating SAST and lightweight DAST scans into feature branches reduced defect detection time, minimised remediation overhead, and reinforced security ownership across the development team, aligning with OWASP's shift-left recommendation and industry standards [OWASP, 2025; OWASP, 2022].

4.2 Defence in Depth through Declarative Policies

Embedding CSP, HSTS and strict CORS rules in code and infrastructure configurations created continuous multi-layer enforcement that persists across environments, preventing reliance on ad-hoc headers. This defence-in-depth approach ensures that if one control fails, others mitigate exploitation and elevate baseline security [Django CSP, 2024; OWASP CSP, 2025].

4.3 Continuous Monitoring and Feedback Loops

Versioning security middleware and automating the collection of policy violations established a closed feedback loop with robust automated alerts, enabling rapid threat detection and iterative policy refinement in line with DevSecOps principles [Snyk Ltd., 2025; Render, Inc., 2025].

4.4 Least Privilege and Immutable Infrastructure

Enforcing least privilege for runtime components and treating builds as disposable further hardened the attack surface, as rebuild-over-patch deployments ensured unauthorised changes could not persist and reduced drift between releases [Docker, Inc., 2025b; Docker, Inc., 2024].

5 Future Work

Building on my pipeline, the following future enhancements will strengthen SecureCart's security posture and resilience by addressing emerging threats, enforcing compliance, and automating advanced security checks as the platform scales.

Enhancement	**Description**	**Potential Threats Mitigated**
Container image scanning	Integrate tools like Trivy or Clair into the CI pipeline to scan base images and dependencies for OS-level vulnerabilities.	Unpatched CVEs in base images; privilege escalation; supply-chain attacks.
Infrastructure-as-Code analysis	Add Terraform/CloudFormation security checks (e.g. Checkov, tfsec) to detect misconfigurations before provisioning.	Over-permissive IAM roles; open security groups; insecure default settings.
Runtime monitoring & alerting	Deploy a SIEM or EDR solution (e.g. Sentry, Falco) in staging and enforce WAF/CSP policies in production.	Zero-day exploit activity; anomalous traffic patterns; undetected XSS or injection attempts.
Automated rollback & canary releases	Enhance the Render/deploy hook workflow to support gradual traffic shifts and automatic rollback on health-check failures.	Faulty releases causing downtime; configuration drift; broken security controls in production.
Performance & load testing	Incorporate Gatling or k6 into the pipeline to simulate real-world loads and detect DoS vectors.	Denial-of-Service attacks; resource exhaustion; scale-out failures.

Table 1: Future Enhancements and Potential Threats Mitigated

6 Conclusion

In this project I carried out end-to-end security testing—using OWASP ZAP, Snyk and flake8—to find and fix missing CSP headers, an outdated Bootstrap library, vulnerable Django dependencies and other design issues. I then codified those remediations in a GitHub Actions pipeline that installs dependencies, runs lint and unit tests, performs SAST and DAST scans, builds and runs a Docker image, captures logs and deploys only when every security gate passes. By shifting security checks left and automating each stage, I have created a repeatable workflow that prevents vulnerable code from reaching production. Going forward, I plan to add container-image scanning, Infrastructure-as-Code analysis and continuous monitoring to keep SecureCart resilient as new threats arise.

References

Bootstrap Team [2024]. *Bootstrap: The most popular HTML, CSS, and JS library*. Version 5.3.6; Accessed: 2025-05-19. URL: https://getbootstrap.com/.

Django CSP [2024]. *django-csp: Content Security Policy for Django*. https://github.com/mozilla/django-csp. Accessed: 2025-05-19.

Django Software Foundation [May 2025a]. *Django 5.1.9 Release Notes*. https://docs.djangoproject.com/en/5.2/releases/5.1.9/. Accessed: 2025-05-19.

— [2025b]. *Django shortcut functions*. https://docs.djangoproject.com/en/5.1/topics/http/shortcuts/. Documentation for version 5.1; Accessed: 2025-05-19.

— [2025c]. *Testing in Django*. Accessed: 2025-05-19.

Docker, Inc. [2023]. *docker container stop*. Accessed: 2025-05-19.

— [2024]. *docker buildx build*. Accessed: 2025-05-19.

— [2025a]. *docker container logs*. Accessed: 2025-05-19.

— [2025b]. *docker container run*. Accessed: 2025-05-19.

Flake8 Development Team [2025]. *User Guide*. Accessed: 2025-05-19.

GitHub Actions Team [2025]. *actions/upload-artifact*. https://github.com/actions/upload-artifact. Accessed: 2025-05-19.

GitHub Docs [2025]. *Getting started with GitHub Actions*. https://docs.github.com/articles/getting-started-with-github-actions. Accessed: 2025-05-19.

Mozilla Contributors [2024]. *Content Security Policy `sandbox` Directive*. Accessed: 2025-05-19. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/sandbox.

— [2025a]. *HTTP `Access-Control-Allow-Origin` Header*. Accessed: 2025-05-19. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin.

— [2025b]. *Subresource Integrity (SRI) Implementation Guide*. Accessed: 2025-05-19. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Practical_implementation_guides/SRI.

National Vulnerability Database [2024a]. *CVE-2024-53907 Detail: Command Injection via strip_tags/striptags*. https://nvd.nist.gov/vuln/detail/CVE-2024-53907. Accessed: 2025-05-19.

— [2024b]. *CVE-2024-53908 Detail: SQL Injection in HasKey JSONField lookup on Oracle*. https://nvd.nist.gov/vuln/detail/CVE-2024-53908. Accessed: 2025-05-19.

— [2025a]. "CVE-2018-20677 Detail". In: Accessed: 2025-05-19.

— [2025b]. "CVE-2019-8331 Detail". In: Accessed: 2025-05-19.

National Vulnerability Database [2025c]. *CVE-2025-26699 Detail: Allocation of Resources Without Limits or Throttling in wrap()/wordwrap*. https://nvd.nist.gov/vuln/detail/CVE-2025-26699. Accessed: 2025-05-19.

OWASP [2022]. *zaproxy/action-full-scan*. https://github.com/zaproxy/action-full-scan. Accessed: 2025-05-19.

— [2025]. *OWASP Zed Attack Proxy (ZAP)*. Accessed: 2025-05-19. OWASP Foundation. URL: https://www.zaproxy.org/.

OWASP Cheat Sheet [2024a]. *Authorization Cheat Sheet*. Accessed: 2025-05-19. OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html.

— [2024b]. *Code Injection Cheat Sheet*. Accessed: 2025-05-19. OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Code_Injection_Cheat_Sheet.html.

OWASP CSP [2025]. *Content Security Policy Cheat Sheet*. Accessed: 2025-05-19. OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html.

PIP Team [2025]. *Repeatable Installs*. https://pip.pypa.io/en/stable/topics/repeatable-installs/. Accessed: 2025-05-19.

Render, Inc. [2025]. *Deploy Hooks*. https://render.com/docs/deploy-hooks. Accessed: 2025-05-19.

Retire.js Community [2020]. *Retire.js: JavaScript library vulnerability scanner*. Accessed: 2025-05-19. URL: https://retirejs.github.io/retire.js/.

Snyk Ltd. [2025]. *Snyk Code and Open Source Analysis*. Accessed: 2025-05-19. URL: https://snyk.io/product/code/.

Willison, Simon [2025]. *WhiteNoise Documentation*. Accessed: 2025-05-19. URL: http://whitenoise.evans.io/.