OWASP Top 10 Security Guide for Python 3.x and Django Applications

The OWASP Top 10 2021 presents significant security challenges for Python/Django developers, with **security misconfiguration ranking as the highest risk** for Django applications, followed closely by vulnerable dependencies and broken access control. OWASP Django's built-in protections provide strong foundations, but proper implementation and configuration remain critical for production security. MDN Web Docs +3

Django applications face unique vulnerability patterns due to their ORM complexity, template system, and extensive middleware stack. CVE Details While the framework's "secure by default" philosophy protects against many attacks automatically, developers must understand where manual security implementation is required. Semgrep LinkedIn Recent CVEs show that even well-protected Django applications remain vulnerable to SQL injection through ORM edge cases, template injection when debug mode is enabled, and directory traversal in file handling. CVE Details +5

The 2021 OWASP Top 10 introduced three new categories specifically relevant to modern Django applications: Insecure Design (addressing architectural flaws), Software and Data Integrity Failures (including Python's dangerous pickle deserialization), and Server-Side Request Forgery (affecting Django applications making external HTTP requests). OWASP Foundation owasp These additions reflect evolving attack surfaces that traditional security controls don't address.

A01:2021 Broken Access Control - Django Authorization Patterns

Broken Access Control moved from 5th to 1st position in 2021, affecting 3.81% of tested applications with over 318,000 CWE occurrences. owasp In Django applications, this primarily manifests as insufficient authorization checks beyond authentication, allowing users to access resources they shouldn't.

Vulnerable Django Patterns

The most common access control failures occur when developers rely solely on authentication without implementing proper authorization: Sharkbyte

```python
# VULNERABLE - No ownership check
def get_user_profile(request, user_id):
    user = User.objects.get(pk=user_id)
    return render(request, 'profile.html', {'user': user})

# VULNERABLE - Missing permission validation
def edit_project(request, project_id):
    project = get_object_or_404(Project, pk=project_id)
    # No check if user owns this project
    form = ProjectForm(instance=project)
```

Real-world exploitation involves URL manipulation where attackers modify object IDs to access other users' data. For example, changing `/profile/123/` to `/profile/456/` to access another user's profile information. Django Forum +2

Secure Django Access Control Implementation

Django provides multiple layers of access control that should be used together:

```python
```

```
# SECURE - Filter by ownership
def get_user_profile(request, user_id):
    # Only allow users to view their own profile
    if request.user.id != int(user_id):
        raise Http404
    user = get_object_or_404(User, pk=user_id)
    return render(request, 'profile.html', {'user': user})

# SECURE - Using Django permissions
from django.contrib.auth.decorators import permission_required

@permission_required('myapp.change_project')
def edit_project(request, project_id):
    project = get_object_or_404(Project, pk=project_id, owner=request.user)
    form = ProjectForm(instance=project)

# SECURE - Class-based view with permissions
class ProjectDetailView(LoginRequiredMixin, UserPassesTestMixin, DetailView):
    model = Project

    def test_func(self):
        return self.get_object().owner == self.request.user
```

Django Authorization Features

Django's **built-in permissions system** provides granular access control through the `django.contrib.auth` framework: OWASP Cheat Sheet Series Django Documentation

python

```
# Custom permissions in models
class Document(models.Model):
  class Meta:
    permissions = [
      ("view_document", "Can view document"),
      ("share_document", "Can share document"),
    ]

# Permission checking in views
from django.contrib.auth.decorators import login_required, permission_required

@login_required
@permission_required('myapp.view_document')
def view_document(request, doc_id):
  document = get_object_or_404(Document, pk=doc_id)
  return render(request, 'document.html', {'document': document})
```

Django REST Framework provides comprehensive permission classes for API endpoints:

python

```
from rest_framework.permissions import IsAuthenticated, IsOwnerOrReadOnly

class DocumentViewSet(viewsets.ModelViewSet):
  queryset = Document.objects.all()
  serializer_class = DocumentSerializer
  permission_classes = [IsAuthenticated, IsOwnerOrReadOnly]

  def get_queryset(self):
    # Filter objects by user ownership
    return Document.objects.filter(owner=self.request.user)
```

A02:2021 Cryptographic Failures - Django Security Settings

Cryptographic Failures (previously "Sensitive Data Exposure") focuses on root causes of data breaches through weak cryptography. OWASP Django applications commonly fail through misconfigured HTTPS settings, weak session management, and improper handling of sensitive data.

Critical Django Security Configuration

Django requires explicit security configuration for production environments. **The most dangerous default is `DEBUG = True`**, which exposes sensitive information: Sharkbyte OWASP Cheat Sheet Series

```python
```

```
# SECURE production settings
import os
from pathlib import Path

# Never hardcode secrets
SECRET_KEY = os.environ['SECRET_KEY']
SECRET_KEY_FALLBACKS = [os.environ.get('OLD_SECRET_KEY')]

# Disable debug in production
DEBUG = False
ALLOWED_HOSTS = ['yourdomain.com', 'www.yourdomain.com']

# Force HTTPS for all connections
SECURE_SSL_REDIRECT = True
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

# HTTP Strict Transport Security (1 year)
SECURE_HSTS_SECONDS = 31536000
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
SECURE_HSTS_PRELOAD = True

# Secure cookie configuration
SESSION_COOKIE_SECURE = True
SESSION_COOKIE_HTTPONLY = True
SESSION_COOKIE_SAMESITE = 'Lax'
CSRF_COOKIE_SECURE = True
CSRF_COOKIE_HTTPONLY = True
```

Modern Python 3.x Cryptography

Python 3.7+ provides enhanced SSL/TLS 1.3 support and improved cryptographic libraries:

Python +4

python

```python
import ssl
import secrets
import hashlib
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

# Secure random token generation
secure_token = secrets.token_urlsafe(32)
secure_hex = secrets.token_hex(16)

# Modern password hashing
def hash_password(password: str) -> tuple[str, str]:
    salt = secrets.token_hex(32)
    password_hash = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode('utf-8'),
        salt.encode('utf-8'),
        100000  # iterations
    )
    return salt, password_hash.hex()

# Symmetric encryption with cryptography library
def encrypt_sensitive_data(data: bytes, password: bytes) -> bytes:
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
    )
    key = base64.urlsafe_b64encode(kdf.derive(password))
    f = Fernet(key)
    return salt + f.encrypt(data)
```

Testing Cryptographic Configuration

Django provides a built-in security check for deployment configurations: djangoproject

```bash
python manage.py check --deploy
```

This validates critical security settings and identifies misconfigurations that could lead to cryptographic failures. Medium

A03:2021 Injection Attacks - Django ORM and Template Security

Injection remains a critical threat despite dropping from 1st to 3rd position. owasp Django's ORM provides strong protection against SQL injection, but vulnerabilities exist in raw queries, template rendering, and specific ORM edge cases. Sharkbyte +4

SQL Injection Vulnerabilities in Django

While Django's ORM automatically parameterizes queries, several attack vectors remain: Sharkbyte +2

python

```python
# VULNERABLE - String formatting in raw queries
def search_users(request):
    query = request.GET['q']
    sql = f"SELECT * FROM users WHERE username = '{query}'"
    User.objects.raw(sql)  # Direct injection

# VULNERABLE - QuerySet.extra() without parameters
def filter_posts(request):
    category = request.GET['category']
    Post.objects.extra(where=[f"category = '{category}'"])

# Recent CVE-2022-34265 - Trunc/Extract injection
def vuln_extract(request):
    payload = request.GET.get('lookup_name')
    experiments = Experiment.objects.filter(
        start_datetime__year=Extract('end_datetime', payload)
    )
```

Real attack payloads exploit these patterns through crafted input:

- `admin'; DROP TABLE users; --` for basic SQL injection
- `year' FROM start_datetime)) OR 1=1;SELECT PG_SLEEP(5)--` for Extract() vulnerabilities github +3

Secure Django ORM Patterns

Django's ORM provides multiple safe query methods that automatically handle parameterization: StackHawk +3

python

```
# SECURE - Using ORM methods (always parameterized)
User.objects.filter(username=user_input)
User.objects.get(id=user_id)
MyModel.objects.exclude(status='deleted')

# SECURE - Raw SQL with parameters
User.objects.raw("SELECT * FROM users WHERE username = %s", [user_input])

# SECURE - Direct database access with parameters
from django.db import connection
with connection.cursor() as cursor:
    cursor.execute("SELECT * FROM users WHERE name = %s", [user_input])

# SECURE - Using F expressions for database operations
from django.db.models import F
User.objects.update(login_count=F('login_count') + 1)
```

Cross-Site Scripting (XSS) Protection

Django's template system automatically escapes dangerous HTML characters, but developers can bypass these protections: Nvisium +5

```
html
```

```
<!-- SECURE - Default auto-escaping -->
<p>{{ user_input }}</p>  <!-- Automatically escaped -->

<!-- VULNERABLE - Using |safe filter -->
{{ user_content|safe }}  <!-- Renders raw HTML -->
{% autoescape off %}{{ user_content }}{% endautoescape %}

<!-- VULNERABLE - Unquoted attributes -->
<div class={{ css_class }}>Content</div>
<!-- Attack: css_class = "x onmouseover=alert('XSS')" -->
```

Secure template practices involve careful use of escaping and proper context handling:

```python
from django.utils.html import format_html, escape

# SECURE - Using format_html for HTML generation
def safe_html_generation(user_input):
    return format_html('<p>{}</p>', user_input)

# SECURE - Explicit escaping
def process_user_content(content):
    escaped_content = escape(content)
    return mark_safe(format_html('<p>{}</p>', escaped_content))
```

Template Injection Testing

Server-Side Template Injection (SSTI) can occur when user input is processed as template code:

Cobalt +2

```python
# Testing for SSTI vulnerabilities
test_payloads = [
    "{{7*7}}", # Basic arithmetic test
    "{% debug %}", # Debug information disclosure
    "{{settings.SECRET_KEY}}", # Configuration extraction
]
```

A04:2021 Insecure Design - Django Architecture Security

Insecure Design is a new category focusing on fundamental design flaws that cannot be fixed through implementation alone. [owasp] For Django applications, this involves threat modeling, secure architecture patterns, and proper use of Django's security features.

Secure Django Application Architecture

Implementing secure design requires architectural thinking beyond individual vulnerabilities:

```python
```

```
# INSECURE DESIGN - Missing rate limiting
def password_reset(request):
    email = request.POST['email']
    # No rate limiting allows enumeration attacks
    user = User.objects.get(email=email)
    send_reset_email(user)

# SECURE DESIGN - Comprehensive protection
from django_ratelimit import ratelimit
from django.contrib.auth import get_user_model

@ratelimit(key='ip', rate='3/m', method='POST')
def secure_password_reset(request):
    email = request.POST['email']
    # Always behave the same regardless of email validity
    try:
        user = get_user_model().objects.get(email__iexact=email)
        send_reset_email(user)
    except get_user_model().DoesNotExist:
        pass  # Don't reveal if email exists

    # Always show same success message
    return render(request, 'reset_sent.html')
```

Business Logic Security Patterns

Insecure design often manifests in business logic flaws that bypass security controls:

```
python
```

```
# SECURE - Multi-factor transaction validation
class MoneyTransferView(LoginRequiredMixin, View):
    def post(self, request):
        amount = Decimal(request.POST['amount'])
        recipient = request.POST['recipient']

        # Business rule validation
        if amount > request.user.profile.daily_limit:
            return HttpResponse("Exceeds daily limit", status=400)

        # Multi-step verification for large amounts
        if amount > Decimal('1000.00'):
            if not request.session.get('mfa_verified'):
                return redirect('mfa_verify')

        # Atomic transaction with proper error handling
        try:
            with transaction.atomic():
                transfer = MoneyTransfer.objects.create(
                    sender=request.user,
                    recipient_id=recipient,
                    amount=amount,
                    status='PENDING'
                )
                # Additional verification steps...
        except Exception as e:
            logger.error(f"Transfer failed: {e}")
            return HttpResponse("Transfer failed", status=500)
```

A05:2021 Security Misconfiguration - Django Production Setup

Security Misconfiguration affects 4.5% of applications and is particularly critical for Django applications. owasp This includes running with debug mode enabled, default configurations, missing security middleware, and improper HTTPS setup. Sharkbyte +4

Critical Django Production Configuration

The most common Django misconfigurations create severe security vulnerabilities: LearnDjango Django Documentation

python

```python
# DANGEROUS - Debug mode exposes sensitive information
DEBUG = True  # Never use in production

# VULNERABLE - Weak secret key management
SECRET_KEY = 'django-insecure-hardcoded-key'  # Predictable key

# INSECURE - Permissive host configuration
ALLOWED_HOSTS = ['*']  # Allows host header injection

# MISSING - Security middleware not configured
MIDDLEWARE = [
    'django.middleware.common.CommonMiddleware',
    # Missing SecurityMiddleware, CSRFViewMiddleware
]
```

Secure production configuration requires comprehensive security settings: OWASP Cheat Sheet Series Django Documentation

python

```
import os
import environ

# Environment variable management
env = environ.Env(DEBUG=(bool, False))
environ.Env.read_env()

# Secure defaults
DEBUG = False
SECRET_KEY = env('SECRET_KEY')  # From environment
ALLOWED_HOSTS = env.list('ALLOWED_HOSTS')

# Complete security middleware stack
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

# Comprehensive security headers
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
X_FRAME_OPTIONS = 'DENY'
SECURE_REFERRER_POLICY = 'strict-origin-when-cross-origin'

# Database security
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': env('DB_NAME'),
        'USER': env('DB_USER'),
        'PASSWORD': env('DB_PASSWORD'),
        'HOST': env('DB_HOST', default='localhost'),
        'PORT': env('DB_PORT', default='5432'),
        'OPTIONS': {
            'sslmode': 'require',
        },
    }
}
```

Django Security Middleware Configuration

Django provides specialized security middleware that must be properly configured:

Django Documentation Django

```python
# Content Security Policy (Django 5.1+)
SECURE_CSP_DEFAULT_SRC = ["'self'"]
SECURE_CSP_SCRIPT_SRC = ["'self'"]
SECURE_CSP_STYLE_SRC = ["'self'", "'unsafe-inline'"]
SECURE_CSP_REPORT_URI = '/csp-report/'

# CSRF Protection
CSRF_TRUSTED_ORIGINS = ['https://yourdomain.com']
CSRF_COOKIE_AGE = 31449600  # 1 year
CSRF_USE_SESSIONS = False  # Use cookies by default
```

A06:2021 Vulnerable and Outdated Components - Python Dependency Management

Vulnerable Components moved significantly up from 9th position, reflecting the critical importance of dependency management in Python/Django applications. owasp The Python Package Index (PyPI) ecosystem requires active monitoring for vulnerabilities.

Python Dependency Vulnerability Management

Python applications face unique challenges with transitive dependencies and package version conflicts:

```bash
# Check for known vulnerabilities
pip install safety
safety check

# Audit pip packages (Python 3.10+)
pip audit

# Generate vulnerability reports
safety check --json --output safety-report.json
pip audit --format=json --output=audit-report.json
```

Secure Dependency Management Patterns

Requirements management should include version pinning and regular updates:

```text
```

```
# requirements.txt - Pin exact versions
Django==4.2.21
psycopg2-binary==2.9.7
cryptography==41.0.4
requests==2.31.0

# requirements-dev.txt - Development dependencies
bandit==1.7.5
safety==2.3.4
pytest-django==4.5.2
```

Automated dependency scanning in CI/CD pipelines:

```
yaml
```

```yaml
# GitHub Actions security workflow
name: Security Checks
on: [push, pull_request]

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install safety bandit
        pip install -r requirements.txt

    - name: Run Safety check
      run: safety check --json --output safety-report.json

    - name: Run Bandit security scan
      run: bandit -r . -f json -o bandit-report.json
```

Virtual Environment Security

Virtual environment isolation prevents dependency conflicts and supply chain attacks:

```bash
# Create isolated environment
python -m venv django_env
source django_env/bin/activate

# Install from trusted sources only
pip install --trusted-host pypi.org --trusted-host pypi.python.org Django

# Generate locked requirements
pip freeze > requirements-lock.txt
```

A07:2021 Identification and Authentication Failures - Django Auth Security

Authentication Failures dropped from 2nd position but remain critical for Django applications. owasp Django's authentication system provides strong defaults but requires proper configuration and implementation. OWASP Cheat Sheet Series

Django Authentication Security Features

Django's **built-in authentication** provides comprehensive password security: owasp Sharkbyte

```python
```

```
# Secure password validation
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
        'OPTIONS': {'min_length': 12}
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]

# Secure session configuration
SESSION_COOKIE_AGE = 3600  # 1 hour timeout
SESSION_EXPIRE_AT_BROWSER_CLOSE = True
SESSION_SAVE_EVERY_REQUEST = True
SESSION_COOKIE_SECURE = True
SESSION_COOKIE_HTTPONLY = True
```

Multi-Factor Authentication Implementation

Two-factor authentication with django-otp:

```
python
```

```
from django_otp.decorators import otp_required
from django_otp.plugins.otp_totp.models import TOTPDevice

@otp_required
def sensitive_view(request):
    # Only accessible with valid OTP token
    return render(request, 'sensitive.html')

# Custom MFA requirement
def mfa_required_view(request):
    if not request.user.is_verified():
        return redirect('two_factor:setup')
    return render(request, 'protected.html')
```

Rate Limiting and Brute Force Protection

django-axes provides comprehensive authentication attack protection: OWASP Cheat Sheet Series

python

```
# settings.py
INSTALLED_APPS = [
    'axes',
    # ...
]

MIDDLEWARE = [
    'axes.middleware.AxesMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    # ...
]

# Axes configuration
AXES_FAILURE_LIMIT = 5
AXES_COOLOFF_TIME = 1  # Hours
AXES_LOCKOUT_CALLABLE = 'myapp.utils.lockout_response'
AXES_ENABLE_ADMIN = True
```

A08:2021 Software and Data Integrity Failures - Python Serialization Security

Software and Data Integrity Failures is a new category with the highest weighted impact from CVE/CVSS data. owasp For Python applications, this primarily involves **pickle deserialization vulnerabilities** and supply chain security.

Python Pickle Security Vulnerabilities

Pickle deserialization creates remote code execution vulnerabilities when processing untrusted data: Redfox Security +3

python

```
import pickle
import base64

# EXTREMELY DANGEROUS - Never unpickle untrusted data
def vulnerable_deserialize(request):
    data = request.POST['serialized_data']
    # This allows arbitrary code execution
    user_data = pickle.loads(base64.b64decode(data))
    return user_data

# Attack payload creation
class RemoteCodeExecution:
    def __reduce__(self):
        import os
        return (os.system, ('rm -rf /',))

# Attacker sends this payload
malicious_payload = base64.b64encode(pickle.dumps(RemoteCodeExecution()))
```

Real-world pickle exploitation has been documented in Django applications using pickle-based session serializers or caching: GitHub

python

```
# VULNERABLE - Using PickleSerializer
SESSION_SERIALIZER = 'django.contrib.sessions.serializers.PickleSerializer'

# SECURE - Using JSONSerializer (default)
SESSION_SERIALIZER = 'django.contrib.sessions.serializers.JSONSerializer'
```

Secure Serialization Alternatives

JSON serialization provides safe alternatives to pickle: Stack Overflow

```python
import json
from django.core import serializers

# SECURE - JSON serialization
def secure_serialize_data(data):
    return json.dumps(data, default=str)

def secure_deserialize_data(json_data):
    return json.loads(json_data)

# SECURE - Django model serialization
def serialize_queryset(queryset):
    return serializers.serialize('json', queryset)

# SECURE - DRF serializers for APIs
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ['username', 'email', 'first_name', 'last_name']
```

Supply Chain Security for Python

Package integrity verification helps prevent supply chain attacks:

```bash
```

```
# Verify package hashes
pip install Django==4.2.21 --hash=sha256:7e4225ec065e0f354ccf7349a22d209de09cc1c074832be9eb84c51c

# Use pip-audit for supply chain monitoring
pip install pip-audit
pip-audit --format=json --output=audit-results.json
```

A09:2021 Security Logging and Monitoring Failures - Django Logging

Security Logging Failures moved up from 10th position, reflecting the critical importance of detecting and responding to attacks. owasp Django applications require comprehensive logging for security monitoring.

Django Security Logging Configuration

Comprehensive logging setup captures security-relevant events:

python

```
# settings.py - Production logging configuration
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'security': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d} {message}',
            'style': '{',
        },
    },
    'handlers': {
        'security_file': {
            'level': 'WARNING',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': '/var/log/django/security.log',
            'maxBytes': 1024*1024*15,  # 15MB
            'backupCount': 10,
            'formatter': 'security',
        },
        'auth_file': {
            'level': 'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': '/var/log/django/auth.log',
            'formatter': 'security',
        },
    },
    'loggers': {
        'django.security': {
            'handlers': ['security_file'],
            'level': 'WARNING',
            'propagate': False,
        },
        'django.contrib.auth': {
            'handlers': ['auth_file'],
            'level': 'INFO',
            'propagate': False,
        },
    },
}
```

Security Event Monitoring

Custom security logging for application-specific events:

```
python
```

```
import logging

security_logger = logging.getLogger('security.authentication')

def monitor_login_attempt(request, username, success):
  client_ip = get_client_ip(request)
  user_agent = request.META.get('HTTP_USER_AGENT', '')

  if success:
    security_logger.info(
      f'Successful login: user={username} ip={client_ip} ua={user_agent}'
    )
  else:
    security_logger.warning(
      f'Failed login attempt: user={username} ip={client_ip} ua={user_agent}'
    )

# Custom middleware for request monitoring
class SecurityMonitoringMiddleware:
  def __init__(self, get_response):
    self.get_response = get_response
    self.logger = logging.getLogger('security.requests')

  def __call__(self, request):
    response = self.get_response(request)

    # Log suspicious activities
    if response.status_code == 403:
      self.logger.warning(
        f'Access denied: {request.path} from {get_client_ip(request)}'
      )
```

```
        return response
```

A10:2021 Server-Side Request Forgery (SSRF) - Django HTTP Security

Server-Side Request Forgery is the newest OWASP Top 10 category, ranked #1 by the community survey. owasp Django applications making HTTP requests based on user input are vulnerable to SSRF attacks targeting internal networks and cloud metadata services.

SSRF Vulnerabilities in Django Applications

Common SSRF patterns occur when Django applications process user-supplied URLs:

```python
import requests

# VULNERABLE - Direct URL processing
def fetch_external_data(request):
    url = request.POST['url']
    # No validation allows internal network access
    response = requests.get(url)
    return JsonResponse({'data': response.text})

# VULNERABLE - Webhook processing
def process_webhook(request):
    callback_url = request.POST['callback_url']
    # Attacker can target internal services
    requests.post(callback_url, json={'status': 'complete'})
```

Real attack scenarios target internal infrastructure:

- `http://169.254.169.254/latest/meta-data/` - AWS metadata service
- `http://localhost:8080/admin/` - Internal admin interfaces
- `http://192.168.1.1/` - Internal network reconnaissance

Secure HTTP Request Handling

URL validation and filtering prevents SSRF attacks:

```python
```

```
import ipaddress
import socket
from urllib.parse import urlparse

def is_safe_url(url):
    """Validate URL to prevent SSRF attacks"""
    try:
        parsed = urlparse(url)

        # Only allow HTTP/HTTPS
        if parsed.scheme not in ('http', 'https'):
            return False

        # Resolve hostname to IP
        hostname = parsed.hostname
        if not hostname:
            return False

        ip = socket.gethostbyname(hostname)
        ip_obj = ipaddress.ip_address(ip)

        # Block private networks
        if ip_obj.is_private or ip_obj.is_loopback or ip_obj.is_link_local:
            return False

        # Block specific dangerous ranges
        blocked_ranges = [
            ipaddress.ip_network('169.254.0.0/16'),  # AWS metadata
            ipaddress.ip_network('10.0.0.0/8'),      # Private
            ipaddress.ip_network('172.16.0.0/12'),   # Private
            ipaddress.ip_network('192.168.0.0/16'),  # Private
        ]
        for blocked_range in blocked_ranges:
            if ip_obj in blocked_range:
                return False

        return True
    except Exception:
        return False

# SECURE - URL validation before requests
def secure_fetch_data(request):
    url = request.POST['url']

    if not is_safe_url(url):
        return JsonResponse({'error': 'Invalid URL'}, status=400)

    try:
        response = requests.get(
            url,
            timeout=10,
            allow_redirects=False
        )
        return JsonResponse({'data': response.text})
    except requests.exceptions.RequestException as e:
        return JsonResponse({'error': 'Request failed'}, status=500)
```

Comprehensive Security Testing Approaches

Static Analysis Integration

Bandit provides Django-specific security pattern detection: GitHub +2 bash

```
# Install security scanning tools
pip install bandit safety semgrep

# Comprehensive security scan
bandit -r . -f json -o bandit-report.json
safety check --json --output safety-report.json
semgrep --config=python.django --json --output=semgrep-report.json
```

Dynamic Security Testing

Automated vulnerability scanning with OWASP ZAP integration:

python

```
from zapv2 import ZAPv2

class DjangoSecurityTest:
    def __init__(self):
        self.zap = ZAPv2(proxies={'http': 'http://localhost:8080'})

    def test_django_app(self, target_url):
        # Spider crawl
        scan_id = self.zap.spider.scan(target_url)

        # Wait for completion
        while int(self.zap.spider.status(scan_id)) < 100:
            time.sleep(5)

        # Active vulnerability scan
        scan_id = self.zap.ascan.scan(target_url)

        # Get results
        alerts = self.zap.core.alerts(baseurl=target_url)
        return alerts
```

Security Unit Testing

Django security test patterns for each vulnerability type:

```
python
```

```python
import pytest
from django.test import TestCase, Client
from django.contrib.auth import get_user_model

class SecurityTestSuite(TestCase):
    def test_csrf_protection(self):
        """Verify CSRF protection is active"""
        response = self.client.post('/sensitive-action/', {
            'data': 'test'
        })
        self.assertEqual(response.status_code, 403)

    def test_xss_protection(self):
        """Test XSS prevention in templates"""
        malicious_input = '<script>alert("XSS")</script>'
        response = self.client.post('/form/', {
            'content': malicious_input
        })
        self.assertNotContains(response, '<script>')
        self.assertContains(response, '&lt;script&gt;')

    def test_sql_injection_protection(self):
        """Verify ORM injection protection"""
        malicious_query = "'; DROP TABLE users; --"
        response = self.client.get(f'/search/?q={malicious_query}')
        self.assertEqual(response.status_code, 200)
        # Verify users table still exists
        self.assertTrue(get_user_model().objects.exists())

    def test_authentication_required(self):
        """Test access control enforcement"""
        response = self.client.get('/admin/')
        self.assertRedirects(response, '/admin/login/?next=/admin/')
```

Conclusion

The OWASP Top 10 2021 presents evolving challenges for Django developers, with **security misconfiguration, vulnerable dependencies, and broken access control** representing the highest risks. Django's security-by-default philosophy provides strong foundations, but **production deployments require explicit security configuration**, regular dependency updates, and comprehensive testing.

Critical implementation priorities include disabling debug mode, implementing proper HTTPS configuration, maintaining current dependencies, and establishing comprehensive logging. The new categories - Insecure Design, Software Integrity Failures, and SSRF - reflect modern attack surfaces requiring architectural security thinking beyond traditional vulnerability mitigation.

Modern Python 3.x features enhance Django security through improved SSL/TLS support, stronger cryptographic libraries, and better secrets management. However, **pickle deserialization remains a critical vulnerability** that developers must actively avoid in favor of JSON serialization.

Automated security testing through static analysis (Bandit, Safety), dynamic scanning (OWASP ZAP), and CI/CD integration provides continuous protection. Combined with Django's built-in security features and proper production configuration, these practices create robust defense against the OWASP Top 10 vulnerabilities in modern Python/Django applications.

The framework's extensive middleware system, ORM protections, and template security create multiple layers of defense, but **developer awareness and proper implementation remain essential** for maintaining security in production environments.