/sentry-django-access-review

Source: `~/.claude/skills/sentry-django-access-review/SKILL.md`

---

name: django-access-review description: 'Django access control and IDOR security review. Use when reviewing Django views, DRF viewsets, ORM queries, or any Python/Django code handling user authorization. Trigger keywords: "IDOR", "access control", "authorization", "Django permissions", "object permissions", "tenant isolation", "broken access".' allowed-tools: Read Grep Glob Bash Task license: LICENSE

Django Access Control & IDOR Review

Find access control vulnerabilities by investigating how the codebase answers one question:

Can User A access, modify, or delete User B's data?

Philosophy: Investigation Over Pattern Matching

Do NOT scan for predefined vulnerable patterns. Instead:

1. **Understand** how authorization works in THIS codebase
2. **Ask questions** about specific data flows
3. **Trace code** to find where (or if) access checks happen
4. **Report** only what you've confirmed through investigation

Every codebase implements authorization differently. Your job is to understand this specific implementation, then find gaps.

Phase 1: Understand the Authorization Model

Before looking for bugs, answer these questions about the codebase:

How is authorization enforced?

Research the codebase to find:

```
□ Where are permission checks implemented?
  - Decorators? (@login_required, @permission_required, custom?)
  - Middleware? (TenantMiddleware, AuthorizationMiddleware?)
  - Base classes? (BaseAPIView, TenantScopedViewSet?)
  - Permission classes? (DRF permission_classes?)
```

```
   - Custom mixins? (OwnershipMixin, TenantMixin?)

□ How are queries scoped?
   - Custom managers? (TenantManager, UserScopedManager?)
   - get_queryset() overrides?
   - Middleware that sets query context?

□ What's the ownership model?
   - Single user ownership? (document.owner_id)
   - Organization/tenant ownership? (document.organization_id)
   - Hierarchical? (org -> team -> user -> resource)
   - Role-based within context? (org admin vs member)
```

Investigation commands

```
# Find how auth is typically done
grep -rn "permission_classes\|@login_required\|@permission_required" --include="*.py" | head -
20

# Find base classes that views inherit from
grep -rn "class Base.*View\|class.*Mixin.*:" --include="*.py" | head -20

# Find custom managers
grep -rn "class.*Manager\|def get_queryset" --include="*.py" | head -20

# Find ownership fields on models
grep -rn "owner\|user_id\|organization\|tenant" --include="models.py" | head -30
```

Do not proceed until you understand the authorization model.

Phase 2: Map the Attack Surface

Identify endpoints that handle user-specific data:

What resources exist?

```
□ What models contain user data?
□ Which have ownership fields (owner_id, user_id, organization_id)?
□ Which are accessed via ID in URLs or request bodies?
```

What operations are exposed?

For each resource, map:

- List endpoints - what data is returned?
- Detail/retrieve endpoints - how is the object fetched?
- Create endpoints - who sets the owner?
- Update endpoints - can users modify others' data?
- Delete endpoints - can users delete others' data?
- Custom actions - what do they access?

Phase 3: Ask Questions and Investigate

For each endpoint that handles user data, ask:

The Core Question

"If I'm User A and I know the ID of User B's resource, can I access it?"

Trace the code to answer this:

```
1. Where does the resource ID enter the system?
   - URL path: /api/documents/{id}/
   - Query param: ?document_id=123
   - Request body: {"document_id": 123}

2. Where is that ID used to fetch data?
   - Find the ORM query or database call

3. Between (1) and (2), what checks exist?
   - Is the query scoped to current user?
   - Is there an explicit ownership check?
   - Is there a permission check on the object?
   - Does a base class or mixin enforce access?
```

```
4. If you can't find a check, is there one you missed?
   - Check parent classes
   - Check middleware
   - Check managers
   - Check decorators at URL level
```

Follow-Up Questions

```
□ For list endpoints: Does the query filter to user's data, or return everything?

□ For create endpoints: Who sets the owner - the server or the request?

□ For bulk operations: Are they scoped to user's data?

□ For related resources: If I can access a document, can I access its comments?
  What if the document belongs to someone else?

□ For tenant/org resources: Can User in Org A access Org B's data by changing
  the org_id in the URL?
```

Phase 4: Trace Specific Flows

Pick a concrete endpoint and trace it completely.

Example Investigation

```
Endpoint: GET /api/documents/{pk}/

1. Find the view handling this URL
   → DocumentViewSet.retrieve() in api/views.py

2. Check what DocumentViewSet inherits from
   → class DocumentViewSet(viewsets.ModelViewSet)
   → No custom base class with authorization

3. Check permission_classes
```

```
   → permission_classes = [IsAuthenticated]
   → Only checks login, not ownership

4. Check get_queryset()
   → def get_queryset(self):
   →     return Document.objects.all()
   → Returns ALL documents!

5. Check for has_object_permission()
   → Not implemented

6. Check retrieve() method
   → Uses default, which calls get_object()
   → get_object() uses get_queryset(), which returns all

7. Conclusion: IDOR - Any authenticated user can access any document
```

What to look for when tracing

```
Potential gap indicators (investigate further, don't auto-flag):
- get_queryset() returns .all() or filters without user
- Direct Model.objects.get(pk=pk) without ownership in query
- ID comes from request body for sensitive operations
- Permission class checks auth but not ownership
- No has_object_permission() and queryset isn't scoped

Likely safe patterns (but verify the implementation):
- get_queryset() filters by request.user or user's org
- Custom permission class with has_object_permission()
- Base class that enforces scoping
- Manager that auto-filters
```

Phase 5: Report Findings

Only report issues you've confirmed through investigation.

Confidence Levels

Level	Meaning	Action
HIGH	Traced the flow, confirmed no check exists	Report with evidence
MEDIUM	Check may exist but couldn't confirm	Note for manual verification
LOW	Theoretical, likely mitigated	Do not report

Suggested Fixes Must Enforce, Not Document

Bad fix: Adding a comment saying "caller must validate permissions" **Good fix**: Adding code that actually validates permissions

A comment or docstring does not enforce authorization. Your suggested fix must include actual code that:

- Validates the user has permission before proceeding
- Raises an exception or returns an error if unauthorized
- Makes unauthorized access impossible, not just discouraged

Example of a BAD fix suggestion:

```
def get_resource(resource_id):
    # IMPORTANT: Caller must ensure user has access to this resource
    return Resource.objects.get(pk=resource_id)
```

Example of a GOOD fix suggestion:

```
def get_resource(resource_id, user):
    resource = Resource.objects.get(pk=resource_id)
    if resource.owner_id != user.id:
        raise PermissionDenied("Access denied")
    return resource
```

If you can't determine the right enforcement mechanism, say so - but never suggest documentation as the fix.

Report Format

```
## Access Control Review: [Component]

### Authorization Model
[Brief description of how this codebase handles authorization]
```

```
### Findings

#### [IDOR-001] [Title] (Severity: High/Medium)
- **Location**: `path/to/file.py:123`
- **Confidence**: High - confirmed through code tracing
- **The Question**: Can User A access User B's documents?
- **Investigation**:
  1. Traced GET /api/documents/{pk}/ to DocumentViewSet
  2. Checked get_queryset() - returns Document.objects.all()
  3. Checked permission_classes - only IsAuthenticated
  4. Checked for has_object_permission() - not implemented
  5. Verified no relevant middleware or base class checks
- **Evidence**: [Code snippet showing the gap]
- **Impact**: Any authenticated user can read any document by ID
- **Suggested Fix**: [Code that enforces authorization - NOT a comment]

### Needs Manual Verification
[Issues where authorization exists but couldn't confirm effectiveness]

### Areas Not Reviewed
[Endpoints or flows not covered in this review]
```

Common Django Authorization Patterns

These are patterns you might find - not a checklist to match against.

Query Scoping

```
# Scoped to user
Document.objects.filter(owner=request.user)

# Scoped to organization
Document.objects.filter(organization=request.user.organization)

# Using a custom manager
Document.objects.for_user(request.user)  # Investigate what this does
```

Permission Enforcement

```
# DRF permission classes
permission_classes = [IsAuthenticated, IsOwner]

# Custom has_object_permission
def has_object_permission(self, request, view, obj):
    return obj.owner == request.user

# Django decorators
@permission_required('app.view_document')

# Manual checks
if document.owner != request.user:
    raise PermissionDenied()
```

Ownership Assignment

```
# Server-side (safe)
def perform_create(self, serializer):
    serializer.save(owner=self.request.user)

# From request (investigate)
serializer.save(**request.data)  # Does request.data include owner?
```

Investigation Checklist

Use this to guide your review, not as a pass/fail checklist:

```
□ I understand how authorization is typically implemented in this codebase
□ I've identified the ownership model (user, org, tenant, etc.)
□ I've mapped the key endpoints that handle user data
□ For each sensitive endpoint, I've traced the flow and asked:
  - Where does the ID come from?
  - Where is data fetched?
  - What checks exist between input and data access?
□ I've verified my findings by checking parent classes and middleware
□ I've only reported issues I've confirmed through investigation
```

Revision #4
Created 2026-02-18 08:39:59 UTC by John
Updated 2026-05-31 20:01:32 UTC by John