APPS BY VIKAS

Meet Vikas



GenzLudo

Ludo with 3D characters and up to 8 players. A first-of-its-kind no-teaming-up feature keeps play fair. Try it - you'll love the difference.

Download on Google Play



Kachu

Too much scrolling straining your neck? Add your addictive apps and complete daily neck exercises to unlock them.

Download on Google Play

SECURE API SERIES

IDOR / BOLA

How changing one object reference can expose, edit, or delete another user's data - and how to prevent it correctly.

AUTHENTICATED USER	REQUESTED OBJECT	QUESTION
User 1001	/api/profiles/**1002**	**Is User 1001 allowed to read User 1002?**

A practical guide for developers, testers, and product teams

1. The 60-second mental model

IDOR and BOLA are authorization failures. A user is allowed to call an endpoint, but the server fails to check whether that user may act on the specific object named in the request.

> **THE CORE RULE** Never trust an object reference merely because it came from your own app. The client can be inspected, modified, automated, or replaced with a custom HTTP request.

Three ingredients

- **An object:** profile, invoice, order, file, message, vehicle, API key, or project.
- **A reference:** numeric ID, UUID, filename, slug, account number, query value, GraphQL variable, or JSON field.
- **A missing object-level check:** the server loads or changes the object without confirming that the current user is permitted to do that action.

Term	Easy meaning	Example
Authentication	Who are you?	A verified session or access token identifies User 1001.
Authorization	What may you do?	User 1001 may read Profile 1001, but not Profile 1002.
IDOR	A direct object reference is used without a correct access check.	Changing /users/1001 to /users/1002 returns another user's data.
BOLA	API-focused name for broken authorization at the object level.	A valid API user acts on an object they do not own or cannot access.

> **MOST IMPORTANT DISTINCTION** A valid login does not automatically make an object safe. Authentication supplies identity. Authorization must connect that identity, the requested action, and the requested object.

2. How an attacker changes 1001 to 1002

The attacker does not edit the button. The button only triggers a request. The attacker observes that request, changes an object reference, and sends it again.

Normal request

```
GET /api/profiles/1001 HTTP/1.1
Host: api.example.com
Authorization: Bearer <token-for-user-1001>
```

Modified request

```
GET /api/profiles/1002 HTTP/1.1
Host: api.example.com
Authorization: Bearer <same-token-for-user-1001>
```

Step	What happens	Security decision
1. Observe	Browser DevTools, a test proxy, app logs, or client code reveals the request structure.	No vulnerability yet.
2. Modify	The object reference changes from 1001 to 1002. The valid login token stays the same.	The client cannot be trusted to preserve the original value.
3. Re-send	The server authenticates User 1001 and loads object 1002.	The server must check permission on object 1002.
4. Result	If allowed incorrectly, the server returns, edits, or deletes another user's object.	This is the authorization failure.

The reference may be anywhere

Location	Example
Path	GET /orders/8421
Query string	GET /download?file_id=8421
JSON body	{"profile_id": 8421, "name": "New name"}
Header	X-Account-ID: 8421
GraphQL variables	{"documentId": "8421"}
Filename or storage key	GET /files/customer-8421.pdf

ETHICAL BOUNDARY Inspect and test only applications you own or systems for which you have explicit permission. Use two controlled test accounts, not real customer records.

3. Why 'the user is logged in' is not enough

A secure decision has at least four inputs:

Input	Question	Example
Subject	Who is making the request?	User 1001 from a verified token.
Action	What do they want to do?	Read, update, delete, export, invite, refund.
Object	Which exact record is targeted?	Profile 1002 or Invoice INV-8421.
Context	What rules apply now?	Ownership, team membership, role, tenant, status, time, consent.

BAD DECISION Token is valid, therefore return whatever ID the request contains.

GOOD DECISION Token identifies User 1001. Policy checks whether User 1001 may perform this action on Object 1002 in the current tenant and state.

Ownership is only one policy

- A private profile may be readable only by its owner.
- A team document may be readable by members, editable by editors, and deletable only by its owner or an administrator.
- A support ticket may be readable by the customer and assigned support agents.
- An order may be cancellable by its owner only before shipment.
- An administrator may need a separate, audited path instead of a blanket bypass.

Related issues that are not exactly the same

Issue	Difference
Broken Function Level Authorization	The user reaches a function or endpoint they should not access, such as /admin/refunds.
Broken Object Property Level Authorization	The user can read or modify fields they should not, such as is_admin or internal_notes.
Mass assignment	Untrusted fields are automatically bound to a model, allowing forbidden properties to change.

4. A vulnerable API, line by line

This simplified Node/Express example authenticates the request but performs an unrestricted database lookup.

```
app.get('/api/profiles/:id', requireAuth, async (req, res) => {
// req.user.id is 1001 because the token is valid
const profile = await db.profile.findUnique({
where: { id: req.params.id } // attacker controls this ID
});

if (!profile) return res.sendStatus(404);
return res.json(profile); // no object authorization check
});
```

Line / behavior	Why it matters
requireAuth	Good, but it proves only that the caller has a valid identity.
req.params.id	This value came from the client and is attacker-controlled.
findUnique by id	The query searches the entire table, not the caller's permitted set.
res.json(profile)	The record is released before an object-level permission decision.

The same bug can modify or destroy data

```
PATCH /api/profiles/1002
{ "phone": "attacker-controlled" }

DELETE /api/documents/1002
```

IMPACT BOLA can cause unauthorized reading, editing, deletion, export, financial action, or device control. The exact impact is business-specific and may be much larger than a simple profile leak.

COMMON REVIEW MISTAKE Reviewers check whether the endpoint requires a token, but do not test the same request using another user's object. A valid-token test is not an authorization test.

5. Secure patterns that scale

Pattern A: do not accept an ID when the object is always 'me'

```
app.get('/api/me/profile', requireAuth, async (req, res) => {
const profile = await db.profile.findUnique({
where: { ownerId: req.user.id }
});
if (!profile) return res.sendStatus(404);
return res.json(selectPublicProfileFields(profile));
});
```

The server derives identity from the verified session, so there is no client-supplied user ID to tamper with.

Pattern B: scope the query to the permitted dataset

```
app.get('/api/projects/:id', requireAuth, async (req, res) => {
const project = await db.project.findFirst({
where: { id: req.params.id, ownerId: req.user.id }
});
if (!project) return res.sendStatus(404);
return res.json(project);
});
```

The requested ID and the ownership rule are evaluated together. A foreign ID behaves like a missing record.

Pattern C: use a policy for shared objects

```
const project = await loadProject(req.params.id);
if (!project) return res.sendStatus(404);

const allowed = await policy.can(req.user, 'project:edit', project);
if (!allowed) return res.sendStatus(403);

await updateProject(project.id, validatedInput);
```

The policy can include membership, tenant, role, ownership, object state, and action. Keep policy logic centralized enough to be consistent, but explicit at every object access boundary.

> **SAFE DEFAULT** Deny unless a policy explicitly allows the subject to perform the action on the object. Apply the rule to read, list, create, update, delete, export, batch, and background-job paths.

6. Authorization beyond simple ownership

Real products often need relationship-based or attribute-based rules. Avoid reducing every decision to currentUser.id == object.userId.

Model	Policy example	Typical use
Ownership	object.owner_id == user.id	Private profiles, personal drafts.
Role-based	user.role in ['support', 'admin']	Operational tools. Still limit tenant and action.
Relationship-based	user is a member of object's workspace	Teams, shared folders, organizations.
Attribute-based	member.role == 'editor' and object.status != 'locked'	Fine-grained business workflows.
Capability-based	A narrow, expiring, signed capability permits one operation	Share links or delegated access.

Multi-tenant example

```
SELECT p.*
FROM projects p
JOIN workspace_members m ON m.workspace_id = p.workspace_id
WHERE p.id = :project_id
AND m.user_id = :authenticated_user_id
AND m.role IN ('owner', 'editor');
```

Avoid the confused-deputy problem

- Do not accept tenant_id, owner_id, or role from the request and treat it as trusted identity.
- Derive the caller from a verified session or token, then load trusted memberships and roles server-side.
- For background jobs, carry an explicit authorization context or re-check policy before the sensitive action.
- For batch endpoints, authorize every object, not only the first ID or the parent container.
- For nested routes such as /teams/T1/projects/P9, verify that P9 actually belongs to T1 and that the user may access T1.

> **TOCTOU NOTE** If permission and mutation are separate operations, access can change between them. For high-risk actions, perform the guarded update atomically in the database or inside a transaction.

7. Where Row-Level Security fits

PostgreSQL Row-Level Security (RLS) can restrict which rows a database role may read or change. It is a strong enforcement layer, especially when several API paths access the same tables.

> **CORRECT STATEMENT** Proper authorization is the solution. RLS is one powerful way to enforce part of that authorization at the database layer.

Illustrative PostgreSQL policies

```
ALTER TABLE profiles ENABLE ROW LEVEL SECURITY;
ALTER TABLE profiles FORCE ROW LEVEL SECURITY;

CREATE POLICY profiles_read_own ON profiles
FOR SELECT
USING (owner_id = app.current_user_id());

CREATE POLICY profiles_update_own ON profiles
FOR UPDATE
USING (owner_id = app.current_user_id())
WITH CHECK (owner_id = app.current_user_id());
```

Here app.current_user_id() represents a trustworthy database function or session context populated from verified server-side identity. Never let the client freely set that value.

Clause	What it controls
USING	Which existing rows are visible or targetable by SELECT, UPDATE, or DELETE.
WITH CHECK	Whether a new row state created by INSERT or UPDATE is allowed.
ENABLE ROW LEVEL SECURITY	Turns policy enforcement on for normal table access.
FORCE ROW LEVEL SECURITY	Also subjects the table owner in normal cases; privileged bypass behavior still needs careful design.

What happens to the 1002 request?

Layer	Result
API	User 1001 asks for Profile 1002.
Database policy	owner_id of Profile 1002 does not equal current authenticated user.
Query result	The row is not visible or modifiable to that runtime role.
API response	Return a safe 404 or 403 according to the product's disclosure policy.

8. RLS pitfalls developers miss

Pitfall	Why it breaks the protection	Safer approach
Privileged runtime role	Superusers and roles with BYPASSRLS bypass policies. Table owners normally bypass unless forced.	Use a restricted runtime role; keep admin/service credentials away from untrusted request paths.
RLS enabled, policy incomplete	Read may be protected while insert, update, delete, RPC, or another table is open.	Map every operation and table. Define USING and WITH CHECK deliberately.
Permissive policy surprise	Multiple permissive policies combine with OR, so one broad policy can reopen access.	Review the effective policy set and add regression tests.
Trusted identity is client-controlled	A policy is only as strong as the identity or tenant context it reads.	Populate identity from verified auth in trusted infrastructure.
Security-definer function or view	Privileged database code can change whose privileges apply or bypass intended filters.	Audit functions, ownership, search_path, grants, and exposed RPCs.
Only database rows are protected	Object storage, caches, search indexes, queues, exports, and external APIs may bypass RLS.	Apply equivalent authorization at every data path.

TEST THE REAL RUNTIME ROLE

A policy that passes in a database console may behave differently through the production API. Test with the same role, claims, connection mode, functions, and query paths used at runtime.

Defense in depth

- Application authorization makes business intent explicit and produces controlled errors and audit events.
- RLS protects against missed filters in many database queries.
- Field allowlists prevent sensitive columns from leaking even from an authorized row.
- Storage rules protect files whose metadata may live in a protected database row.
- Automated tests catch regressions when routes, policies, roles, or relationships change.

DO NOT ASSUME

RLS does not automatically secure every table, every privileged connection, every function, or every non-database system. Treat it as a powerful layer, not a substitute for a complete authorization design.

9. Fixes that look useful but do not solve BOLA

Attempt	Why it is not the real fix	How to use it safely
Hide the ID in the UI	Hidden inputs, disabled fields, and mobile app screens are controlled by the client.	Enforce permission on the server.
Use UUIDs	Harder to guess, but a leaked, shared, logged, or observed UUID can still be replayed.	Use UUIDs only as defense in depth plus authorization.
Encrypt or encode IDs	Encoding is reversible; custom encryption can be fragile; neither proves permission.	Prefer simple identifiers with correct policy checks.
Require login	Authentication answers who, not whether the caller may use this object.	Authorize every object and action.
Validate the ID format	A valid integer or UUID can still identify another user's object.	Validate format, then authorize the referenced object.
Add rate limiting	It may slow enumeration but a single unauthorized request is still a bug.	Rate limit as abuse control, not authorization.
Return 404	A 404 can reduce object-existence leakage, but it does not prevent access by itself.	Authorize first; choose 404 or 403 consistently.

404 or 403?

- **403 Forbidden**
- **404 Not Found**
- Either can be correct. The security control is the denial itself, not the status code.
- Keep internal logs precise even if the external response is intentionally generic.

> **REMEMBER** Unpredictable identifiers reduce easy enumeration. They do not create authority. Possession of an ordinary object ID must not equal permission unless you intentionally designed a narrow capability URL.

10. A practical authorization test plan

Use controlled accounts and objects. The goal is to prove that the same operation is allowed for the right user and denied for the wrong user.

Minimum setup

- Create User A and User B in the same tenant, plus User C in another tenant.
- Create one object owned by each user and one shared team object.
- Give users different roles, such as viewer, editor, owner, and admin.
- Capture known-good requests, then change only the object reference or relationship context.

Operation	Own object	Same tenant, no permission	Other tenant	Expected
Read	A reads A	A reads B	A reads C	Allow first; deny others.
Update	A edits A	A edits B	A edits C	Allow only per policy.
Delete	A deletes test A	A deletes B	A deletes C	Deny unauthorized actions.
List/search	A lists permitted objects	Check foreign rows	Check tenant isolation	No unauthorized row appears.
Export/download	A exports A	A exports B	A exports C	Files and signed URLs follow policy.
Batch/GraphQL	Allowed IDs	Mixed allowed + denied IDs	Cross-tenant IDs	Authorize every object.

Assertions that matter

- The response does not contain unauthorized fields, counts, metadata, filenames, or timing clues beyond the chosen policy.
- The database or downstream system did not perform the forbidden change.
- The denial is logged with subject, action, object type, route, tenant, and correlation ID - without logging secrets.
- Changing HTTP method, content type, API version, mobile/web client, or GraphQL alias does not bypass the rule.
- Regression tests run whenever endpoint, schema, role, policy, or sharing logic changes.

NEGATIVE TESTS ARE FIRST-CLASS

A happy-path test proves that an authorized user can work. A BOLA test proves that another valid user cannot perform the same operation on the same object.

11. Developer review checklist

1	Inventory every endpoint, RPC, GraphQL resolver, job, export, and storage path that accepts or derives an object reference.
2	Treat every path parameter, query value, header, body field, filename, and GraphQL variable as untrusted.
3	Derive caller identity from verified authentication, never from user_id supplied by the client.
4	Define subject + action + object + context for each sensitive decision.
5	Scope database queries to the caller's permitted dataset or call an explicit policy before release or mutation.
6	Authorize read, list, create, update, delete, export, batch, and admin actions separately.
7	Validate tenant and parent-child relationships, not only object ownership.
8	Allowlist response fields and writable fields to prevent property-level leaks and mass assignment.
9	Use UUIDs only as defense in depth, never as the authorization mechanism.
10	If using RLS, verify policies, USING, WITH CHECK, runtime role, bypass roles, functions, views, and all affected tables.
11	Apply matching authorization to object storage, caches, search, queues, analytics exports, and third-party systems.
12	Add two-user and cross-tenant negative tests for every sensitive operation.
13	Log denied attempts safely and alert on repeated cross-object access patterns.
14	Fail closed: missing identity, policy errors, or ambiguous context should deny access.

ONE-LINE SUMMARY Never ask only, 'Is this user logged in?' Ask, 'May this user perform this action on this exact object right now?'

12. Reference architecture and sources

Recommended enforcement chain

Layer	Responsibility
1. Authentication	Verify the token/session and establish a trusted subject.
2. Input boundary	Parse and validate object references and allowed fields.
3. Application policy	Decide subject + action + object + context; deny by default.
4. Scoped data access	Query only the permitted dataset; avoid unrestricted find-by-ID patterns.
5. Database RLS	Enforce row visibility/modification as defense in depth where appropriate.
6. Output control	Return only fields the caller may see.
7. Audit and tests	Record denials safely and continuously test cross-user and cross-tenant cases.

Primary references

OWASP API Security Top 10 - API1:2023 Broken Object Level Authorization

Defines BOLA, explains that every endpoint receiving an object ID should perform object-level authorization, and provides prevention guidance.

OWASP IDOR Prevention Cheat Sheet

Covers object references in paths and request bodies, identifier complexity, access verification, scoped lookups, and testing with multiple accounts.

PostgreSQL Documentation - Row Security Policies

Documents RLS behavior, policy expressions, default deny, command-specific policies, policy combination, and privileged bypass behavior.

SCOPE NOTE

Examples are intentionally simplified for learning. Adapt them to your framework, threat model, tenancy model, database roles, and business rules. Security testing must be authorized.

More things worth exploring

Find more free guides at vikasbuilds.com/resources