

Forum:[Hack Forums](#)

Thread: [IDOR Hunting Guide | 2026](#)

Vinavil - January 17, 2026, 6:41 am

????????????????????????????????

IDOR HUNTING

A Practical Hunting Guide

????????????????????????????????

A tutorial by Vinavil

Quick Reference Checklist

Find user owned objects (profiles, documents, orders, settings)

Capture requests in your proxy

Swap object IDs between two test accounts

Test all methods: GET, POST, PUT, DELETE

Verify impact: can you read, modify, delete, or trigger actions?

What Is An IDOR?

IDOR stands for Insecure Direct Object Reference. The concept is stupidly simple: the application uses user supplied input to access objects directly without proper authorization checks. You change a number, you get someone else's data. That's it.

Here's the thing. Developers in 2026 still get this wrong constantly. Authentication is not authorization, and this fundamental confusion leads to IDORs being everywhere. The real world impact is massive: full account takeovers, PII exposure, financial data leaks, admin privilege escalation.

The best part? Most automated scanners can't find IDORs Reliably. These are logic flaws, not signature based vulnerabilities. IDORs reward actual hackers who understand how applications work.

? IDOR vs BOLA: The Terminology

You'll hear both terms thrown around. IDOR is the classic, broad concept: directly referencing objects without proper authorization. BOLA (Broken Object Level Authorization) is the API focused framing from the OWASP API Security Top 10. Same vulnerability class, different lens. If you're testing REST or GraphQL APIs, expect programs to use BOLA in their taxonomy. Know both, report

using whatever the program prefers.

Why Bother Chasing IDORs?

Often High Impact IDORs that expose sensitive data are commonly rewarded well. Payouts vary by program and scope, but access control failures consistently rank among the most impactful findings.

Everywhere, All The Time Every application with user specific data is potentially vulnerable. E commerce, SaaS, fintech, healthcare, social media.

Low Competition Scanners miss these. Junior hunters overlook them. If you develop a solid IDOR methodology, you'll find what others miss.

Easy to Demonstrate Impact Clear to demonstrate when you can show cross account access with screenshots and request/response pairs.

The Technical Deep Dive

? Numeric Sequential IDs

The classic. Auto incrementing integers exposed directly to users.

```
Code GET /api/v1/users/1337/profile HTTP/1.1
Host: api.target.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Now try user 1338 with YOUR token...
GET /api/v1/users/1338/profile HTTP/1.1

If you get user 1338's data, that's an IDOR
? UUID/GUID References

Developers think UUIDs are secure because they're "random." UUIDs reduce guessing, but they don't replace authorization. They're identifiers, not secrets. They leak in API responses, notification emails, WebSocket messages, JavaScript bundles, error messages, and export functionality. Once you have a UUID, the authorization check is what matters.

```
Code GET /api/documents/550e8400-e29b-41d4-a716-446655440000 HTTP/1.1
Authorization: Bearer user_token
```

Find UUIDs anywhere, try to access those objects
? Encoded/Hashed References

Security through obscurity rarely works.

Code Base64 encoded ID
GET /api/invoice/MTMzNw== HTTP/1.1
Decode: MTMzNw== -> 1337
Encode different ID: echo -n "1338" | base64 -> MTMzOA==

GET /api/invoice/MTMzOA== HTTP/1.1
? Parameter Based IDORs

Not all IDORs are in the URL path. Check request bodies, query parameters, headers, and cookies.

Code POST /api/transfer HTTP/1.1
Content-Type: application/json

```
{  
  "from_account": "ACC1337",  
  "to_account": "ACC9999",  
  "amount": 100  
}
```

What happens if you change from_account to someone else's?

The Methodology

? Step 1: Map All User Specific Endpoints

Create two accounts (UserA and UserB). As UserA, use the application while proxying through Burp. Hit every feature: profiles, settings, documents, orders, messages, team management, API endpoints. Export the sitemap.

? Step 2: Identify Object References

Go through every request and highlight any parameter that looks like an object reference:

Code URL paths: /users/1337/profile, /documents/abc123
Query params: ?user_id=1337&file=document_847.pdf
Request body: {"userId": 1337, "documentId": "docabc"}
Headers: X-User-ID: 1337
Cookies: user_context=eyJ1c2VySWQiOjEzMzd9

? Step 3: Collect References From Both Accounts

Map UserB the same way. Get valid object references belonging to UserB to test if UserA can access them.

? Step 4: Cross Reference Testing

For each endpoint, try accessing UserB's objects using UserA's session. Test every HTTP method. Sometimes GET is protected but PUT/DELETE isn't.

Code Read access denied but what about modify?
GET /api/users/1338/profile -> 403 Forbidden

PUT /api/users/1338/profile HTTP/1.1

Content-Type: application/json

```
{"name": "Pwned", "email": "vinavil@hackforums.com"}
```

Sometimes this works when GET doesn't

Pro tip: build a quick matrix mapping endpoint x method x object owner (A vs B) and mark pass/fail. Makes it systematic and you won't miss anything.

? Step 5: Compare Responses

Don't just check status codes. Compare the full response between authorized and unauthorized requests:

Status code differences (200 vs 403 vs 404)

Response body size (different lengths can indicate partial data leakage)

Fields present in JSON responses

Response timing (slower responses might indicate backend processing)

Error message variations ("not found" vs "access denied" vs generic error)

Silent leaks happen when the status code looks normal but the body contains data it shouldn't.

? Step 6: Verify Authorization Is Applied Everywhere

Within your authorized testing scope, verify that access controls are consistently enforced across different request formats. Inconsistent routing or validation can reveal gaps:

Code Confirm consistency with variations:

/users/1338/ # Trailing slash

/USERS/1338 # Case variation

/users/1338.json # Extension handling

Also test how the app handles duplicate parameters or arrays if the API supports them. These variations help confirm whether the authorization logic is applied uniformly or if certain code paths were missed during implementation.

Advanced Techniques

? Blind IDORs

Sometimes you won't see the data in the response, but the action still executes. Check if emails were sent to another user, passwords changed, or settings modified. This is where having two test accounts is crucial.

? Method Override Testing

When testing targets, check if the application handles method overrides consistently with direct method calls:

Code POST /api/users/1338 HTTP/1.1

X-HTTP-Method-Override: DELETE

? Reference Leakage Hunting

UUIDs reduce guessing, but authorization is what prevents access. Hunt for leakage in: API responses listing objects, search functionality, activity feeds, export functionality, error messages, client side storage, and share links.

Real World Examples

? Case 1: E Commerce Invoice Disclosure

Code GET /api/v2/orders/87432/invoice HTTP/1.1
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGc...

Decrement order ID
GET /api/v2/orders/87431/invoice HTTP/1.1

Result: Another customer's invoice with full PII
Impact: Large number of invoices accessible via enumeration

? Case 2: SaaS Admin Panel Escalation

Code Regular user attempting admin action
DELETE /api/team/users/USER_UUID HTTP/1.1
Authorization: Bearer regular_user_token

Response: 200 OK
Regular users could delete ANY user including admins

POST /api/team/users/USER_UUID/role HTTP/1.1
{
 "role": "admin"
}
Could also promote yourself to admin

? Case 3: Mobile App Push Notification Hijack

Code POST /api/devices/register HTTP/1.1
{
 "device_id": "ATTACKER_DEVICE",
 "user_id": "67890", # Another user's ID
 "push_token": "attacker_firebase_token"
}

Result: Push notifications for user 67890 go to attacker
Including 2FA codes and password reset links

What's NOT an IDOR

Public Data If data is intentionally public, accessing it isn't a vulnerability.

Self Referential Access User 1337 accessing /users/1337 with their own token is normal.

User Enumeration Telling if an ID exists without data access is typically low severity at best.

Generic Error Responses If an ID returns "not found" or a generic error without leaking data or side channel information, that's usually enumeration, not IDOR.

Final Thoughts

IDORs are everywhere and they're not going away. Every time a developer writes code that takes an ID from the user and fetches data without checking ownership, they create an IDOR.

The hunters making consistent finds have IDOR testing baked into their methodology. Every endpoint, every parameter, every request gets the treatment. Many endpoints take under a minute to sanity check once you're in the flow, and the potential impact makes it worth testing.

The bugs are there. Go find them.

Happy hunting. Vinavil