Sample Pentest Findings

Four real findings, written for engineers

Senior-led penetration testing. Reports that close on first read.

Four real findings, written for engineers

A draft lead magnet for engineering-leader cold-outreach prospects.

This document is the canonical source for the PDF at `public/downloads/cyberguards-engineering-leader-sample-findings.pdf`. Regenerate the PDF after edits with: `python3 scripts/generate-lead-magnet-pdfs.py`

Why this exists

Engineering leaders ask one question more than any other when they evaluate a pentest vendor: *"Will my engineers respect the report?"*

This document is four redacted, anonymized real-world findings from CyberGuards engagements, presented exactly the way they appear in our client reports. No marketing rewrites, no executive summaries dressed up to look like findings — just the working artifact your engineers will actually open.

Each finding shows:

- The title line your engineering team will see first (with severity and the

control reference, if applicable).

- The reproduction steps a developer can replay locally.
- The working proof we captured during the test.
- The severity rationale — why we rated it the way we did.
- The remediation we recommended, with a paste-ready code or

configuration snippet where applicable.

These are real engagements. Company names, hostnames, user IDs, and other identifying details have been redacted or replaced with neutral placeholders.

Finding 1 — IDOR in admin invoice export endpoint

Severity: Critical **Affected component:** `/admin/api/invoices/export?org_id={N}` **Discovered during:** Authenticated web application pentest, Series B SaaS client, scope week 2 of 3.

Summary

The admin invoice export endpoint accepts an `org_id` query parameter and returns a CSV of all invoices for that organization. The endpoint validates that the caller has an *admin* role on *some* organization, but does not validate that the caller is an admin of the organization they are requesting.

Any authenticated admin of any tenant could enumerate `org_id` and download the invoices of every other tenant on the platform — including invoice line items, customer names, and ACH return references.

Reproduction

- Sign in to the platform as an admin of any organization (we used a

self-signed-up free-tier account).

- Capture the session cookie.
- Request:

`GET /admin/api/invoices/export?org_id=42` with that cookie.

- The server returns a 200 response with `Content-Type: text/csv` and the

full invoice export for `org_id=42`, regardless of whether the caller is an admin of that organization.

Working proof

During the engagement we captured invoice exports for three test tenants we do not have legitimate access to. Exports include columns for customer email, billing address, amount, and bank-return references. Full request / response pair attached in Appendix A of the engagement report.

Severity rationale — Critical

We rate Critical because all of the following are simultaneously true:

- No additional privilege escalation is required — any authenticated admin

of any tenant can exploit.

- Cross-tenant data exposure is the textbook definition of broken access

control (OWASP A01:2021).

- The exposed records include financial PII that this client's customer

contracts explicitly forbid sharing with other tenants.

- The endpoint is reachable from the public internet without IP allowlist

or VPN.

If only one of these were true we would rate High. With all four simultaneously present, the realistic blast radius justifies Critical.

Recommended remediation

The handler must validate that the caller is a member of the requested organization with the *admin* role, before reading any invoice record.

A minimal Express middleware sketch that captures the intent (adapt to your framework):

```
  function requireAdminOf(req, res, next) {
    const requestedOrgId = Number(req.query.org_id);
    if (!Number.isInteger(requestedOrgId)) return res.sendStatus(400);

    const membership = req.session.memberships
      .find(m => m.org_id === requestedOrgId);
```

```
    if (!membership) return res.sendStatus(403);
    if (membership.role !== 'admin') return res.sendStatus(403);
    next();
  }

  app.get('/admin/api/invoices/export', requireAdminOf, exportInvoicesHandler);
```

The membership lookup must read from the request session, never from a client-controlled header or token. The 403 responses should be structurally identical regardless of whether the org exists or whether the caller is a non-admin member — otherwise the response shape becomes a side channel for enumeration.

We also recommend adding a regression test that asserts the endpoint returns 403 when the caller is admin of org A and requests the export for org B.

Finding 2 — SSRF via image proxy on profile-import flow

Severity: High **Affected component:** `/api/profile/import-avatar` **Discovered during:** External web application pentest, fintech client, scope week 1 of 2.

Summary

The profile-import flow accepts a `source_url` parameter and fetches the URL server-side to mirror the avatar to internal object storage. The fetch honours arbitrary URLs without scheme allowlist, host allowlist, or IP- range exclusion.

We demonstrated server-side request forgery to the AWS instance metadata service (`http://169.254.169.254/latest/meta-data/iam/security-credentials/`), which returned the instance role name. Combined with the IAM role policy in use, this would allow an attacker to retrieve short-lived credentials for the role and act as the application server from outside the cluster.

Reproduction

- Sign in as any authenticated user.
- Issue:

`POST /api/profile/import-avatar` with body: `{"source_url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/"}`

- The server fetches the URL, base64-encodes the response, and returns it

as the "avatar preview." The response body is the instance role name.

- Follow up with:

`{"source_url": "http://169.254.169.254/latest/meta-data/iam/security-credentials/{role-name}"}` to retrieve the temporary credential document.

We did not exfiltrate the credentials — the engagement scope authorized demonstration only, not credential extraction. Screenshots and timing captured in Appendix B.

Severity rationale — High

We rate High rather than Critical because IMDSv2 was not enforced on this client's instances, which materially increases impact. We would have rated Critical had we successfully chained to a full cloud takeover during the engagement window, but the engagement scope terminated SSRF demonstration at the credentials-document return.

The Critical-to-High distinction matters here because the next step is real cloud takeover, and the remediation work is materially different (enforcing IMDSv2 + fixing the SSRF, versus just fixing the SSRF). We separate them so the engineering team can sequence the work correctly.

Recommended remediation

Two changes, both required:

- Constrain the image-fetch to an explicit scheme + host allowlist. HTTPS

only, public DNS only, no IP-literal hosts, no private RFC 1918 ranges, no link-local (169.254/16), no loopback. Resolve the hostname before the fetch and reject if the resolved IP lands in any excluded range.

- Enforce IMDSv2 cluster-wide

(`HttpTokens=required` on the EC2 instance metadata options). This is a defense-in-depth control: even if a future SSRF is introduced, IMDSv2 requires a session token that the SSRF cannot obtain.

A sketch of the host validation (Node, using the `dns` and `ipaddr.js` modules):

```
async function fetchSafe(url) {
  const u = new URL(url);
  if (u.protocol !== 'https:') throw new Error('scheme not allowed');
  const ips = await dns.resolve(u.hostname);
  for (const ip of ips) {
    const addr = ipaddr.parse(ip);
    if (addr.range() !== 'unicast') throw new Error('non-public IP');
  }
  return fetch(url);
}
```

`addr.range()` returns `'unicast'` for public addresses and one of `'private'`, `'loopback'`, `'linkLocal'`, `'reserved'`, etc. for the ranges you want to reject. Validate every resolved IP, not just the first one — DNS rebinding attacks can return different addresses across resolves.

Finding 3 — Authentication bypass via reused JWT signing key

Severity: High **Affected component:** Identity service `idp.internal.example.com` **Discovered during:** Internal network pentest, healthcare client, scope week 2 of 2.

Summary

The client's identity service signs both their access tokens and their refresh tokens with the same HS256 secret. A refresh token can be replayed as an access token because the access-token verifier accepts any

token with a valid HS256 signature from the shared secret, regardless of the `typ` claim.

We obtained a refresh token through legitimate authentication, then used it directly as an access-token bearer credential against the application API. The API accepted it and returned the requesting user's data.

Reproduction

- Authenticate to the identity service as any user. Capture both the

access token and the refresh token from the response.

- Issue an API request:

`GET /api/me` `Authorization: Bearer <refresh_token>`

- The API returns the user's profile. The refresh token was accepted as an

access token.

Severity rationale — High

We rate High because:

- Refresh tokens are typically issued with longer expiration than access

tokens (this client used 30 days vs. 15 minutes), so a captured refresh token has a much wider exploitation window.

- Refresh tokens are often stored in less-protected locations (longer-term

cookies, mobile-app secure storage) than access tokens.

- A captured refresh token bypasses any access-token-shortening posture the

client put in place to limit blast radius from access-token theft.

We do not rate Critical because exploitation requires that the attacker already have an authenticated user's refresh token — there is no remote unauthenticated path. The risk profile is "if any user's refresh token leaks, full account takeover for 30 days."

Recommended remediation

The fundamental fix is to verify the `typ` (or equivalent) claim on every token before honouring it as the credential for that endpoint:

- Access tokens must have `typ: "access"` and the access-token verifier

must reject anything else.

- Refresh tokens must have `typ: "refresh"` and the refresh endpoint must

reject anything else.

A separate, stronger fix is to use distinct signing keys per token type (or move to asymmetric signing with separate kid-rotation for access vs. refresh). This makes the cross-type acceptance physically impossible rather than merely policy-prohibited.

A minimal verifier sketch (Node, using `jsonwebtoken`):

```
function verifyAccess(token) {
  const claims = jwt.verify(token, ACCESS_SECRET, {
    algorithms: ['HS256'],
    issuer: 'idp.example.com',
    audience: 'api.example.com',
  });
  if (claims.typ !== 'access') {
    throw new Error('not an access token');
  }
  return claims;
}
```

The `typ` check must happen *after* `jwt.verify` succeeds — checking it before verification leaves a window where an unsigned or wrongly-signed token could be inspected. Verify first, then check claims.

Finding 4 — Information disclosure via debug header on error responses

Severity: Low **Affected component:** API gateway, all `/api/*` endpoints **Discovered during:** External web application pentest, fintech client, scope week 2 of 2.

Summary

The API gateway includes an `X-Debug-Internal-Trace` header on 5xx error responses that contains the stack frame of the upstream service where the error originated. The header was left on after a debugging session and the gateway configuration was not reverted.

The leaked information includes internal hostnames, file paths inside the container image, package versions, and in some cases SQL fragments from ORM errors.

Reproduction

- Send a request that triggers a 5xx response — we used a malformed JSON

body to a JSON-required endpoint.

- Inspect the response headers. The `X-Debug-Internal-Trace` header

contains the upstream stack frame.

Severity rationale — Low

We rate Low because:

- The disclosed information aids reconnaissance but does not by itself

enable a foothold.

- The leaked SQL fragments did not contain user data in any of the cases

we observed during the engagement.

- The leaked internal hostnames are not directly reachable from the

internet.

We do not rate Informational because the disclosed package versions could materially shorten the time-to-exploit window for a future CVE in those packages. If, six months from now, a CVE drops against the disclosed ORM version, an attacker who has captured these headers from your error responses already knows you are vulnerable before you have a chance to patch.

Recommended remediation

Remove the `X-Debug-Internal-Trace` header from production gateway configuration. Add a regression check in the gateway config-test suite that asserts the header is absent on all response codes.

A gateway middleware sketch (Express style, adapt to your gateway):

```
app.use((err, req, res, next) => {
  // Log full trace internally, never in headers
  logger.error({ err, route: req.path }, 'upstream error');
  res.removeHeader('X-Debug-Internal-Trace');
  res.removeHeader('X-Powered-By');
  return res.status(500).json({ error: 'internal_error' });
});
```

If you genuinely need a debug header for triage, route it through a correlation ID that is logged internally and returned in the response, but that does not contain stack data. Engineers can resolve the correlation ID in your logging system; attackers cannot.

How we present findings in client reports

Every finding in a CyberGuards report ships with the five sections you saw above:

- **Summary** — what the finding is, in two to three sentences any

engineer on the team can read.

- **Reproduction** — numbered steps a developer can replay locally without

needing the tester present.

- **Working proof** — the artifact (response body, screenshot, captured

request) that demonstrates the finding actually works. No theoretical findings, no scanner output without verification.

- **Severity rationale** — *why* the severity is what it is, not just the

label. Engineering leaders need this to sequence the remediation queue.

- **Recommended remediation** — a paste-ready snippet (code, config,

policy, whichever applies) plus an explanation of why that's the right fix rather than a different one. We name the trade-offs.

Findings include the relevant control reference in the title line where applicable (SOC 2 CC, ISO 27001 Annex A, PCI DSS requirement, HIPAA safeguard), so compliance leaders can pull the same report and use it as audit evidence without rework.

If this is the engineering-respect shape you want from your next pentest, we'd like to talk.

Book a 30-minute scoping call: https://cyberguards.ai/contact/

Read the full For Engineering Leaders page: https://cyberguards.ai/for-engineering-leaders/