The Windows Enterprise Ecosystem — Why It Exists This Way

The Core Problem: Trust at Scale

Imagine a company with 5,000 employees and 3,000 computers. Every employee needs to log into potentially any machine, access shared files, print documents, and use internal applications — and IT needs to manage all of this centrally rather than configuring each machine by hand.

This creates a few hard problems that Windows had to solve simultaneously:

1. **Where is the "truth" about who exists and what they're allowed to do?** (Directory problem)
2. **How does a user prove who they are, repeatedly, across many different services, without typing a password every time or sending it over the network constantly?** (Authentication problem)
3. **How does a machine find the right server to talk to for any of this?** (Discovery problem)
4. **How do administrators (and services) remotely manage/query things across the network?** (Remote management problem)
5. **How do people share files/printers without treating every folder as its own island?** (Resource sharing problem)

Active Directory isn't one thing — it's the umbrella architecture that stitches together a separate solution to each of these five problems. That's why it "relies on" LDAP, Kerberos, DNS, SMB, and RPC: each one is the answer to one specific piece of the puzzle, not an arbitrary add-on.

Problem 1 — Where Is the Truth? (Directory Service → LDAP)

Before directory services, every machine had its own local list of users (think of the old Windows "Local Users and Groups"). That doesn't scale — you can't manually keep 3,000 machines in sync every time someone joins or leaves the company.

The solution: put all user/computer/group data in **one central database**, on servers dedicated to holding it (Domain Controllers), and give every machine a standard way to *query* that database.

LDAP is that standard query language. It wasn't invented by Microsoft — it's an open protocol for reading/writing hierarchical directory data (this is why other systems, like OpenLDAP on Linux, speak it too). Microsoft built Active Directory's actual database around this pre-existing, well-understood protocol instead of inventing a proprietary one, so that querying "who is this user, what groups are they in, what's their SID" has one consistent interface.

Why this matters to you as an attacker: LDAP is *designed* to be queryable — that's its entire purpose. The vulnerability isn't a bug, it's the intended function operating without proper access control. When anonymous LDAP bind works, you're not exploiting a flaw in LDAP — you're exploiting the fact that the directory's core job (answer questions about who exists) was left open to anyone. That's why LDAP enumeration is often the single highest-value early step: it's not a side-channel leak, it's the front door left unlocked.

Problem 2 — Proving Identity, Over and Over, Without Leaking It (Kerberos)

This is the big one, so let's build it up properly.

The Problem With the Naive Approach

The simplest way to prove your identity to a service is: send your username and password to it, every single time. But in an enterprise, a user might touch 10 different services in a day (file server, email, print server, internal web app...). Sending your actual password to 10 different servers is a security nightmare — any one of those servers being compromised or logging traffic exposes your real credential.

Older Windows used **NTLM** to work around this — a challenge-response scheme where you never send the password itself, just proof you know it (a hash-based response to a challenge). It works, but it has a structural weakness: the server itself has to be trusted to *ask* the right challenge and *verify* the answer, and there's no way to verify the identity of the *server itself* to the client. It's also vulnerable to relay — if someone captures your NTLM handshake in flight, they can potentially replay it to authenticate as you elsewhere.

Kerberos's Actual Design Idea

Kerberos was built on a different philosophy: **don't have the user prove their identity to each service individually — have one trusted third party issue reusable, time-limited proof, and let services validate that proof without ever contacting the user directly.**

Here's the rationale, step by step:

1. **You log in once**, and your machine proves your identity to a central authority — the **KDC (Key Distribution Center)**, which runs on every Domain Controller. This happens *once*, not once-per-service.
2. In exchange, you get a **TGT (Ticket Granting Ticket)** — think of it as a wristband from a festival: it doesn't tell venues who you are, but it proves the festival's security already checked your ID, so nobody else needs to check it again. You hold onto this ticket instead of your password.
3. When you actually want to use a specific service (say, a file server), you don't send your password there either. Instead, you show your TGT back to the KDC and say "give me a ticket specifically for this file server." The KDC issues you a **service ticket**

(TGS), encrypted with *that specific service's* password hash — meaning only that service can actually decrypt and use it.

4. You hand that service ticket to the file server. The file server decrypts it with a key only it and the KDC know, confirms it's valid and not expired, and now trusts you — **without ever talking to the KDC directly at that moment**, and without you ever sending a password anywhere in this whole exchange.

Why This Design Choice Matters

- **Your actual password never travels across the network at all**, not even in hashed form, during normal day-to-day authentication. Compare this to NTLM, where a hash-based exchange still happens with every single service you touch.
- **Mutual trust, not just one-directional proof** — the ticket system means services can verify you without contacting a central server every time, which also scales much better across a huge enterprise network.
- **Time-limited tickets** — a TGT typically lasts ~10 hours by default, limiting how long a stolen ticket is useful, unlike a stolen password which is useful until it's changed.

Why This Design Creates Exactly the Attacks You've Been Reading About

Now here's the part that connects rationale to offense — **every major Kerberos attack is a direct consequence of a specific design tradeoff:**

- **Kerberoasting exists because of a deliberate compatibility decision.** Any user is allowed to request a service ticket for *any* service in the domain — that's necessary, because otherwise the system couldn't work (you need to be able to ask for a ticket to a service you haven't used yet). But that service ticket is encrypted with the *service account's own password hash*. If that account has a weak password, and the ticket is encrypted with a hash derived from it, you can request the ticket freely (that's allowed by design) and then crack it offline forever, with no further contact with the DC. The vulnerability isn't a flaw in Kerberos — it's what happens when the "anyone can request a ticket" design meets a human who chose a weak service account password.
- **AS-REP Roasting exists because Kerberos has an optional step called "pre-authentication"** — normally, before issuing you a TGT, the KDC requires you to first encrypt a timestamp with your password hash and send it, proving you actually know the password *before* getting a ticket. This stops someone from just requesting a TGT for any username with no proof at all. But this check is a per-account setting that can be **disabled** (`DONT_REQUIRE_PREAUTH`) — meaning, if it's off, anyone can request a TGT for that account with zero proof of identity, and the KDC will happily encrypt part of the response with that account's password hash. This opens a hash-cracking opportunity that shouldn't exist, purely because someone disabled a safety check that exists for exactly this reason.

- **Golden Tickets exist because of what the trust model requires.** Every TGT issued by the KDC is validated because it's encrypted with one specific account's hash — the `krbtgt` account, a special built-in account whose only job is signing TGTs. If you ever obtain that one hash, you can forge a TGT for *any* user, including ones that don't exist, with any group memberships you want, and every DC in the domain will accept it as legitimate — because the entire trust system boils down to "if it's encrypted correctly with the krbtgt hash, it's real." This is the direct consequence of centralizing trust into one signing key: incredible efficiency and scalability, but a single point of catastrophic failure if that one hash leaks.
- **Unconstrained delegation exists because some services legitimately need to act on your behalf against a *second* service they don't know about in advance** (e.g., a web server that needs to query a database as you). The "solution" that got built for this was: let the front-end server cache a copy of your full TGT so it can request whatever tickets it needs later. That's incredibly convenient — and also means that if an attacker compromises that front-end server, they now have a copy of every TGT of everyone who's ever logged into it, including Domain Admins who checked in once. The convenience *is* the vulnerability.

Each of these isn't "a bug in Kerberos." Kerberos works exactly as designed in each case — the attacks exploit the necessary tradeoffs of a system that has to balance usability, scalability, and security across an entire enterprise.

Problem 3 — How Does Anything Find Anything? (DNS)

None of the above works if a workstation doesn't know **which server is the KDC**, **which server holds the LDAP directory**, or **which server is even a Domain Controller** in the first place. In a large enterprise, this can't be hardcoded — DCs get added, removed, moved, replaced.

Active Directory solves this by **piggybacking on DNS**, using a special record type called **SRV records** that don't just say "here's an IP for this hostname," but say "here's which service is available, on which host, on which port." For example, a query like `_ldap._tcp.dc._msdcs.domain.local` literally answers the question "which host(s) are Domain Controllers offering LDAP?"

This is why DNS is considered core AD infrastructure rather than just generic name resolution — **every single AD operation starts with a DNS lookup** to figure out where to even send the Kerberos/LDAP traffic. It's the "phone book" that makes the rest of the system discoverable without manual configuration.

Why this matters to you: if DNS is misconfigured or a zone transfer is allowed, you can sometimes map the entire internal service topology (every DC, every service that registers itself) without touching a single Windows-specific protocol — because AD's discovery mechanism is, underneath, just DNS records doing what DNS records do.

Problem 4 — Remote Management (RPC Over SMB Named Pipes)

Once you're authenticated (Kerberos) and you know who's who (LDAP), you still need an actual mechanism for a client to say "query the list of domain users" or "reset this user's password" or "tell me what shares this server has." This is a *remote function call* problem: a client wants to invoke functionality that physically executes on another machine.

RPC (Remote Procedure Call) is the general solution to that problem, and Microsoft's implementation (MSRPC) is what powers almost every administrative interaction in Windows — user management, service control, event logs, printing, you name it.

The reason this rides on **SMB named pipes** specifically (rather than always using its own dedicated ports) goes back to a very practical constraint: **firewalls**. If every RPC interface needed its own dynamically assigned port, managing enterprise firewalls would be a nightmare — you'd need to open a huge range of high ports just to let basic admin tools function. By tunneling RPC calls through SMB (which only needs port 445), Windows collapsed a whole category of "which ports do I open for this to work" problems into one well-known port.

Why this matters to you: this is precisely why so much of Windows/AD enumeration can happen through **just port 445** — `samr`, `lsarpc`, `srvsvc`, `netlogon` are all RPC interfaces, but because they're tunneled through SMB, one open port gives you access to a huge amount of the same functionality that would otherwise need direct RPC (port 135 + dynamic ports). The design choice made for firewall-friendliness is the same choice that makes SMB such a rich single target for enumeration.

Problem 5 — Sharing Files and Printers Without Chaos (SMB)

This is the most "boring" of the five problems, but still foundational: users need to save files to shared network locations and print to shared printers, and this needs to respect the same permission model as everything else (the same users/groups defined in AD, the same authentication via Kerberos).

SMB solves this — and because Microsoft needed a transport for *both* file sharing and the RPC-over-named-pipes trick above, it made sense architecturally to unify them under one protocol and one port. That's why SMB feels like it "does everything" — file shares, printer shares, AND the pipe/RPC mechanism for admin queries — it's not coincidence, it's the same transport layer being reused for multiple related jobs.

Putting It All Together: What Actually Happens When Someone Logs In

This is the story that ties every piece above into one flow — worth internalizing because it's the mental model that makes "where do I even start on this box" click:

1. **DNS lookup** — workstation asks "where's a Domain Controller?" via an SRV record query.
2. **Kerberos AS-REQ/AS-REP** — workstation sends proof of password knowledge (pre-auth) to the KDC, gets back a **TGT**.
3. User wants to access a file share. Workstation sends the TGT back to the KDC, asking for a **service ticket** for that specific file server (**TGS-REQ/TGS-REP**).
4. Workstation presents that service ticket to the file server **over SMB (port 445)**.
5. The file server validates the ticket (it can decrypt it, because it shares a secret with the KDC), and now trusts the connection.
6. Underneath that same SMB session, if any administrative query needs to happen (e.g. checking group membership via `samr`), it happens as an **RPC call over a named pipe**, still riding on that same port 445 connection.
7. If any of this needs to check "is this user actually in the Finance OU / Domain Admins group," that's an **LDAP** query against the directory.

Every protocol here is a different chapter of this exact same story — they're not separate systems bolted together, they're a purpose-built pipeline where each piece hands off to the next.

Why This Mental Model Makes You Faster on HTB

Once you see it this way, an HTB box stops being "which random tool do I throw at this open port" and becomes a diagnostic process:

- **Port 53 open?** → This is the discovery layer. Ask: what does the domain's SRV records reveal about topology?
- **Port 88 open?** → This is the trust-issuing layer. Ask: can I get a ticket without proof (AS-REP roast)? Can I request tickets for services with weak passwords (Kerberoast)?
- **Port 389/636 open?** → This is the "ground truth" layer. Ask: can I just read the directory to learn everything about the domain's structure, users, and permissions?
- **Port 445 open?** → This is the resource-sharing + remote-management layer. Ask: what shares exist, what RPC interfaces are reachable through IPC$, and does the SMB configuration itself have exploitable weaknesses (signing disabled, null sessions allowed)?

Instead of memorizing "run enum4linux, then run GetNPUsers, then run BloodHound," the reasoning becomes: *"This is an authentication system that trades passwords for tickets — where in that trade could the check have been skipped or weakened?"* That question applies to a box you've never seen before, which is exactly the difference between a 10-minute solve and 4 hours of trial and error.