I'm setting up some docker containers in WSL2 that want to use a Kerberos ticket cache based on my user to authenticate to a remote SQL Server. It isn't working, and I need to get some detailed diagnostics as to what might be the root cause. Are there any industry-standard packages that provide such diagnostics, or do I need to roll my own?

You don't necessarily need to roll *everything* yourself — there *are* mature and industry-accepted techniques (and partial tools) to help you diagnose Kerberos issues — but in a complex environment (Docker + WSL2 + SQL Server + cross-platform), you'll probably end up combining those tools with some custom instrumentation. Below is a breakdown of what you can use (and build) to get good diagnostics and root-cause visibility.

What you want to observe / diagnose

Before tooling, it helps to enumerate *what needs to break / be visible* in your scenario. Some of the things you'll want to see:

- **Ticket cache state** — is your container seeing a valid TGT in the credential cache? Does it see the right cache file (path, ownership, permissions)?
- **Which principal / realm** is being used for the authentication.
- **Service ticket request / SPN resolution** — when the SQL client requests a service ticket, what SPN is it asking for, and is the KDC giving you a ticket?
- **KDC communication** — DNS lookups for KDC discovery, queries to the KDC (UDP/TCP 88) or fallback ports, error messages from the KDC (e.g. "no support for encryption type," "principal unknown," "preauth failed," etc.).
- **Encryption types / key types** — whether the client and KDC agree on acceptable etypes.
- **Time / clock skew** — whether the client, container, host, and KDC clocks are within allowed skew.
- **Authorization / delegation / impersonation / S4U / cross-realm if relevant**
- **Errors from the SQL Server side / SPN mismatch / mutual authentication mismatch**
- **Networking / firewall / port reachability** (especially inside containers or WSL2 bridges)

If you can trace all those, you'll likely find the root cause.

Off-the-shelf / standard facilities & tools

Here are existing, well-accepted mechanisms and tools (open source or part of the Kerberos ecosystem) that you should use first (and often combine):

Tool / Mechanism	What it gives you	How useful in y
KRB5_TRACE (MIT Kerberos trace logging)	Many MIT / krb5 library-based clients will emit extremely verbose trace logs (internal library operations) when you set `KRB5_TRACE=/path/to/log` (or `KRB5_TRACE=/dev/stdout`) before launching the binary. Massachusetts I... +1	If your SQL client o (or a wrapper that internal step: cache
Kerberos / KDC logs	On the KDC side, logs will show ticket requests, rejections, error codes.	If you have access/ correlate client-side
System call tracing (e.g. strace, ltrace)	You can see what files are being opened (e.g. which credential cache file), what DNS lookups are done, what network syscalls are made.	Useful to confirm " socket / port." (K5W k5wiki.kerberos....
Network packet capture (tcpdump / Wireshark / tshark)	Capture traffic to/from the KDC, see DNS, see Kerberos KDC exchanges, see which ports/addresses are used.	Particularly valuabl the KDC traffic is n
klist / kvno / kinit / kdestroy / kpasswd	Standard command-line tools in the Kerberos ecosystem for viewing, requesting, validating tickets.	You can inject test sidecar) to assert th
Kerberos debugging options / logging sections in `krb5.conf`	The `[logging]` section in krb5.conf allows you to route general logs, KDC logs, admin logs, etc.	Helps turn on mod processes.
K5Wiki / upstream krb5 debugging options	The Kerberos community has documented many compile-time or runtime flags, debugging switches, etc. k5wiki.kerberos....	Use them when you logging.

So yes — the baseline is "use standard trace / logging / capture tools." Many Kerberos deployments and rollouts do exactly this.

However, in your environment (Docker + WSL2), there are complicating factors (filesystem isolation, user namespace mapping, networking translation) which likely hide or intercept some of the expected behavior. That's why you may need some custom glue.

What you might need to build / instrument (or wrap) yourself

Because of your particular stack, you'll likely need some additional "bridging" instrumentation to glue together the off-the-shelf bits. Here are some suggestions of things you might build or adapt:

1. **Wrapper "probe" container / sidecar**
   - A small container in the same pod / network namespace that you can drop into your Docker setup whose sole purpose is to run diagnostic commands (klist, kvno, kinit with trace, tcpdump, etc).
   - That probe container can share the credential cache volume with the main container (read-only) so you can introspect whether the ticket cache is what you expect.
   - You could even mount the same `/etc/krb5.conf` so you can test "what happens *inside* the container" without interfering with your real application.
2. **Instrument the SQL client / Kerberos library layer**
   - If your client library has hooks (e.g. via logging, or you can compile it with debug flags), turn on verbose logging around GSSAPI / SPNEGO / Kerberos steps.
   - Wrap or intercept the GSSAPI negotiation calls (if possible) to dump what service principal it's requesting, or which ticket it's selecting.
   - In Java / .NET / other languages, many Kerberos or GSSAPI libraries already support debug or verbose flags — enable them.
3. **Log propagation / correlation**
   - Correlate timestamps across container, host, KDC logs.
   - Add correlation IDs or context (e.g. container ID, principal) to logs so you know which request you're looking at.
   - Use logging aggregation so you can search through multiple sources (host, container, KDC) in one place.
4. **Health check / assertion hooks**

- At container startup, run a diagnostic check: e.g. “kinit / kvno to SQL SPN / GSSAPI handshake test” and dump success/failure before your app runs.
- Monitor for expired credentials / ticket renewal failures and log proactively.

5. **Custom “debug mode” that increases verbosity**
   - Consider bundling a “diagnostic mode” with your container images that, when turned on, enables full trace logging, packet capture, and dumps of credential cache.
   - That avoids having to rebuild entirely separate debug vs production images.

Practical approach for your scenario (WSL2 + Docker + SQL Server)

Given your environment, here is an approach I’d follow (and recommend) to home in on the problem:

1. **Validate baseline Kerberos from the host**
   - On your WSL2 host (outside containers), make sure you can obtain a ticket (`kinit`), see it (`klist`), and use `kvno` or a simple Kerberoast-like request for the SQL SPN to confirm the environment is working.
   - Use `KRB5_TRACE=/dev/stdout <command>` to see detailed trace.
   - Use `tcpdump` or `wireshark` on the host to confirm KDC traffic.
2. **Reproduce inside a plain container / sidecar**
   - Launch a minimal container (e.g. Debian / Ubuntu) inside your Docker network that has the same krb5 config, network, and volumes (for credential cache) as your target container.
   - Inside it, run `kinit`, `klist`, `kvno` to the SQL SPN, using KRB5_TRACE, and observe whether you can reach the KDC and get a service ticket.
   - If that fails, use `tcpdump` inside the container to see if packets go out / fail.
3. **Compare environment differences**
   - Are you using a non-default credential cache type (e.g. KEYRING, DIR, etc) that the container runtime or kernel doesn’t support?
   - Are file permissions / UID/GID mapping preventing the container from reading the cache file?

- Is network routing / nameserver / DNS resolution inside WSL2 / container different (e.g. DNS servers, SRV record lookup, etc)?
- Is the SQL driver requesting a non-default SPN or doing name canonicalization differently inside container vs host?

4. **Enable debug / verbose mode in your SQL / Kerberos library**
   - Many SQL / ADO / JDBC / ODBC drivers support verbose Kerberos / GSSAPI logs. Turn those on.
   - Configure the client library's Kerberos / GSS module (if exposed) to emit full trace (or use a debug build).
   - If the library is open source, you can instrument or patch it to dump which SPN it's asking and which credential cache handle it's opening.
5. **Cross-check KDC logs**
   - On your domain / KDC side, inspect the KDC logs for the client's IP / principal / time. You'll likely see either a request (with error) or nothing (which hints at client network issue).
   - Match up error codes (for instance, "KDC_ERR_ETYPE_NOSUPP", "KDC_ERR_PREAUTH_FAILED", "KDC_ERR_S_PRINCIPAL_UNKNOWN", etc).
6. **Iterate with narrowed scoping**
   - If things succeed in the minimal container but fail in your real container, compare environment: library versions, config, user permissions, mounted volumes, SELinux / AppArmor / container security profiles, etc.
   - If even the minimal container fails, the issue is at a lower level (network, DNS, routing, credential cache access). Use packet tracing / syscall tracing to drill in.

Over time, you'll "see" the failure point: "no ticket in cache," "couldn't contact KDC," "principal not found," "SPN mismatch," etc.

Known pitfalls & "gotchas" in your kind of setup

In a Docker + WSL2 + Kerberos environment, some common tricky failure modes to watch out for:

- **Credential cache type / kernel support** — modern krb5 uses e.g. `KEYRING:` or `DIR:` cache types instead of plain `FILE:`. If your container/kernel

doesn't support the chosen type (or mapping from host), the cache may be invisible or inaccessible.

- **Filesystem / permission / UID mapping** — the container may not have permission to read the ticket cache file, or may map UIDs differently.
- **DNS / SRV lookup differences** — inside WSL / containers, DNS servers may differ, or DNS names (e.g. _kerberos._tcp) may not resolve properly.
- **Network isolation / firewall / NAT** — the container may not reach the KDC (UDP 88, TCP 88, fallback ports) because of Docker/WSL bridging rules.
- **Time sync / clock skew** — if host, container, or KDC clocks are out of sync beyond allowed skew, tickets will be rejected.
- **SPN canonicalization / name resolution** — the SQL client may canonicalize hostnames differently in container vs host, leading to a mismatched SPN (e.g. FQDN vs short name).
- **Multiple Krb5 libraries / conflicts** — you might have conflicting versions of krb5 (e.g. in host vs container vs driver-embedded) and behavior differences.
- **Keytab / delegation / constrained delegation** — if your scenario requires delegation or impersonation, missing delegation flags or constrained S4U settings may block things.
- **Cross-realm / trust relationships** — more complex AD setups may require cross-realm configuration or correct name mappings.

Summary & recommendation

- **Yes**, there are industry-standard diagnostic tools and techniques in the Kerberos ecosystem (trace logging, packet sniffing, KDC logs, command-line tools, syscall tracing) that you should use first.
- But in a complex containerized + WSL2 environment, you'll almost certainly need to "glue" them with lightweight custom probes/wrappers/sidecars or instrument your client library to expose the missing visibility.
- The best approach is *incremental*: start with known-working environment (host), then move inside a minimal container, then your real container, comparing differences, and using the standard diagnostic tools at each step.

- If you like, I can help you build a diagnostic “probe container” or sample instrumentation for your SQL driver + Kerberos library in your stack — would you like me to sketch that out for your specific language/framework?