Remote Support Tools & Techniques

A Complete 10-Chapter Technical Support Course

Topics covered:

SSH key-based auth · port forwarding & jump hosts

RDP, VNC & relay-based remote desktop tools

Screen sharing's own trust model · least privilege, even for yourself

Working safely on systems you don't own · diagnosing connection failures

Session logging & audit trails

Capstone: three fresh scenarios — SSH, RDP, and screen sharing

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Before You Can Diagnose, You Have to Connect: What This Course Adds

Remote Support Tools & Techniques

Chapter 1 · Before You Can Diagnose, You Have to Connect: What This Course Adds

Every command in this subject's other five courses — a `pg_stat_activity` query, a `top` snapshot, a `curl -v` — assumed a terminal was already open on the right machine. This course is about the step that made that possible in the first place: getting there, safely, and knowing what genuinely changes when the machine — or the person sitting at it — isn't yours.

What the Other Five Courses Silently Assume

COURSE	WHAT IT ASSUMES YOU ALREADY HAVE
Logging & Log Analysis (`log1`)	A way to read the logs — access to the machine or its log aggregator
Network Troubleshooting (`netdiag1`)	A terminal to run `ping`, `dig`, `nc` from
System Monitoring & Performance Diagnosis (`perfdiag1`)	A shell already open on the machine being diagnosed
Web & Application Troubleshooting (`appdiag1`)	Database and application access already granted
Incident Response & Ticketing Workflows (`incident1`)	That the whole team already has whatever access the incident needs

None of them teach how that access actually gets established. This course fills exactly that gap.

Three Genuinely Different Kinds of "Getting There"

SCENARIO	TOOL	COVERED IN
SSH into a production Linux server	SSH, key-based auth, jump hosts	Chapters 2–3
Reach a Windows server's GUI to run a diagnostic tool	RDP / VNC	Chapter 4
Help a confused end user on their own laptop	Screen-sharing software	Chapter 5

Genuinely different tools, genuinely different trust models, genuinely different risks — the capstone applies all three to fresh scenarios of their own, mirroring this course's own three-part structure directly.

Why "Just Connect" Isn't as Simple as It Sounds

Remote access tools make a costly mistake trivially easy — connecting to the wrong host because two sessions look identical, carrying more privilege than a purely diagnostic task ever needed, or, for end-user screen sharing, taking control of someone's machine without them fully understanding what you can see and do. This course is as much about doing remote support *safely* as it is about doing it at all.

> **REMOTE ACCESS IS OFTEN THE ACTUAL ATTACK SURFACE, NOT "JUST INFRASTRUCTURE"**
>
> A compromised SSH key or an overly permissive remote-desktop policy is a classic, real way systems actually get breached — treating remote access tooling as beneath worrying about, in a course otherwise focused on real diagnosis, would be a genuine mistake. It deserves the same care as everything else in this subject, not less.

> **KNOW YOUR ORGANIZATION'S OWN REMOTE-ACCESS POLICY BEFORE YOU NEED IT**
>
> The same principle `incident1` applied to escalation paths applies here directly: figuring out what access you're authorized to use, and through which tool, is much better done calmly in advance than scrambled together in the middle of an urgent ticket.

This course doesn't replace this site's own **Remote Access With SSH** (`ssh1`), **Remote Desktop: RDP & VNC** (`rdp1`), or **VPN** (`vpn1`) courses — it applies and extends their material specifically for the support-diagnostic context those courses don't focus on.

What This Course Covers

SSH key-based authentication done right, port forwarding and jump hosts, RDP/VNC and modern remote-desktop alternatives, the different trust model end-user screen sharing requires, least privilege applied to your own access, working safely on a system you don't own, diagnosing a connection that won't even establish, and session logging. The capstone applies all of it to three fresh scenarios.

Hands-On Exercises

EXERCISE 1

Explain what this chapter says all five of the other Technical Support courses silently assume, and why that leaves a genuine gap this course fills.

View solution

EXERCISE 2

Explain why this chapter treats remote-access tooling as a genuine security concern in its own right, rather than as neutral infrastructure beneath separate attention.

View solution

EXERCISE 3

Name this chapter's three "getting there" scenarios and explain why they're described as genuinely different, not just three examples of the same underlying task.

View solution

CHAPTER 1 QUICK REFERENCE

- All five other Technical Support courses assume access already exists — this course covers how it's actually established
- Three genuinely different scenarios: **SSH to a server, RDP/VNC to a GUI, screen sharing with an end user** — different tools, different trust models
- Remote access tools make costly mistakes trivially easy — wrong host, excess privilege, unclear consent
- Remote access is a genuine security concern, not neutral infrastructure — deserves the same care as anything else in this subject
- This course extends, not replaces, `ssh1` / `rdp1` / `vpn1` — applied specifically to the support-diagnostic context
- **Next chapter:** SSH Fundamentals for Support: Key-Based Auth Done Right

CHAPTER 2 OF 10

SSH Fundamentals for Support: Key-Based Auth Done Right

Remote Support Tools & Techniques

Chapter 2 · SSH Fundamentals for Support: Key-Based Auth Done Right

This site's own **Remote Access With SSH** (`ssh1`) course covers SSH's full mechanics. This chapter stays narrowly focused on the practices a support engineer needs to get right every day: why key-based authentication is the standard, one rule that's genuinely non-negotiable, and reading a system's own access list as real diagnostic evidence.

Why Key-Based Auth Over Passwords

SSH key-based authentication uses a public/private key pair — the private key never leaves your own machine, and authentication happens through a cryptographic challenge rather than transmitting a secret over the connection at all. Two real, concrete advantages: nothing secret ever crosses the wire to be intercepted or brute-forced, and access is revocable per person — removing one person's public key from a system doesn't affect anyone else's access, unlike a shared password that has to be rotated for everyone the moment one person leaves.

The Mechanics, Briefly

```
$ ssh-keygen -t ed25519 -C "j.lee@company.com"
Generating public/private ed25519 key pair.
Enter passphrase (empty for no passphrase): ********
Your identification has been saved in /home/jlee/.ssh/id_ed25519
Your public key has been saved in /home/jlee/.ssh/id_ed25519.pub
```

The public key (`.pub`) is what gets added to a target system's `~/.ssh/authorized_keys` . The private key stays exactly where it was generated.

The Cardinal Rule: Never Share a Private Key

A SHARED "TEAM KEY" DESTROYS EXACTLY THE EVIDENCE THIS SUBJECT IS BUILT ON

Handing out one shared private key to an entire team — "here's the deploy key, everyone uses it" — is a genuinely damaging, common mistake. It destroys per-user accountability entirely: a log showing "the shared key connected" tells you nothing about which actual person was at the keyboard, directly undermining the same evidence-based diagnosis this whole subject has built around since `log1`'s own first chapter. It also makes revocation all-or-nothing — if the key leaks, or one person leaves, everyone's access breaks and has to be re-issued at once. Each person should have their own keypair; access is granted by adding their own public key to a system, never by distributing a private one.

Passphrase-Protecting Your Private Key

A private key file with no passphrase means anyone who obtains a copy of the file itself — a stolen laptop, a leaked backup — can use it immediately, with no further barrier at all. A passphrase adds a genuine second layer, at the real cost of needing to enter it (or caching it for a session via an `ssh-agent`). This isn't an absolute mandate for every scenario — some automated, non-interactive use cases genuinely can't use a passphrase-protected key and need other compensating controls instead — but for a support engineer's own personal, interactive key, a passphrase is the sensible default.

Reading authorized_keys Like Evidence

A system's own `~/.ssh/authorized_keys` file is itself a legitimate diagnostic artifact, in the exact same spirit as `log1`'s own "read it" discipline — each line is one authorized public key, and checking who currently has access to a system can be as simple as reading this one file directly. Genuinely useful during onboarding/offboarding verification, or when investigating "how did they get in" during a security incident.

WORKING EXAMPLE: AN AUTHORIZED_KEYS FILE, BEFORE AND AFTER

Before — unlabeled, effectively unauditable:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIxxxxxx...
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIyyyyyy...
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQD...
```

After — a clean, identifying comment on every key:

```
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIxxxxxx... j.lee@company.com — added 2026-03-14
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIyyyyyy... m.patel@company.com — added 2026-05-02
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQD... a.rossi@company.com — added 2026-07-19 (legacy RSA, scheduled rotatio
```

The "before" version is three anonymous, uninterpretable lines — nobody could confidently answer "who is this?" months later. The "after" version answers it directly, on sight, for anyone who ever needs to.

Hands-On Exercises

EXERCISE 1

Explain why key-based SSH authentication is considered more secure than password authentication, using this chapter's own two stated advantages.

View solution

EXERCISE 2

Explain both problems this chapter identifies with a team sharing one private key, and why the first one connects directly back to `log1`'s own evidence-based diagnosis theme.

View solution

EXERCISE 3

Using this chapter's before/after `authorized_keys` example, explain what specifically makes the "after" version more useful, and in what real situation that difference would actually matter.

View solution

CHAPTER 2 QUICK REFERENCE

- Key-based auth: nothing secret crosses the wire, and access is revocable **per person**, not all at once
- **Never share a private key** — one keypair per person, access granted by adding a public key, not distributing a private one
- A shared key destroys per-user accountability and makes revocation all-or-nothing
- Passphrase-protect your own interactive key as the sensible default — a bare key file is usable by anyone who obtains it
- `authorized_keys` is itself a diagnostic/audit artifact — read it directly to see who currently has access
- Label every key with a clear, identifying comment — an unlabeled key is effectively unauditable months later
- **Next chapter:** SSH Beyond a Plain Shell: Port Forwarding & Jump Hosts

CHAPTER 3 OF 10

SSH Beyond a Plain Shell: Port Forwarding & Jump Hosts

Remote Support Tools & Techniques

Chapter 3 · SSH Beyond a Plain Shell: Port Forwarding & Jump Hosts

A plain shell is only one thing SSH is good for. This chapter covers two techniques a support engineer runs into constantly: reaching a service that was deliberately never exposed directly, and connecting through a hardened gateway to internal servers you can't reach straight from the internet.

Local Port Forwarding: Reaching a Service That Isn't Exposed

```
$ ssh -L 5432:localhost:5432 db-server.internal
```

This forwards your own machine's local port 5432 through the SSH connection to port 5432 *as seen from the remote side* — commonly its own `localhost`, reaching a service that's only listening internally on that server, never exposed to the outside at all. Once the tunnel is open, connecting a local client to `localhost:5432` on your own machine transparently reaches the remote database.

Why Services Are Often Not Directly Exposed — And Why That's Good

A database listening only on `localhost`, reachable by nothing outside that one machine, is a deliberate security practice — the same reasoning **Network Troubleshooting** covers for firewall rules: reducing the attack surface by not exposing anything that doesn't genuinely need to be reachable from outside. SSH port forwarding is the sanctioned way to reach such a service for legitimate diagnostic work, without weakening that security posture by opening the port more broadly to make things convenient. The "obstacle" of not being directly reachable is the control working exactly as intended — tunneling through SSH works with it, not around it.

Remote Port Forwarding: The Reverse Direction

`ssh -R` reverses the direction — exposing a port on your own machine so the remote side can reach it. Less common for this course's own use cases than local forwarding, but worth knowing it exists: it's the right tool when a remote system needs to temporarily reach something running locally on your machine, rather than the other way around.

Jump Hosts and Bastion Hosts

Many organizations don't allow direct SSH access to internal servers from the general internet at all — instead, one hardened, carefully monitored "jump host" (or bastion host) is the only machine reachable from outside. You connect to it first, then reach the actual target from there. The modern, clean way to do this in one step:

```
$ ssh -J bastion.company.com db-server.internal
```

Concentrating the exposed attack surface onto one well-monitored machine, rather than every internal server needing to individually defend itself against internet-facing attacks, is exactly the same logic behind not exposing the database port directly.

Combining Both: ProxyJump Plus Port Forwarding

```
$ ssh -J bastion.company.com -L 5432:localhost:5432 db-server.internal
```

One command: connect through the bastion, land on the database server, and forward its localhost-only database port to your own machine — all in a single line.

Simplifying Repeated Use With ~/.ssh/config

```
Host db-server
    HostName db-server.internal
    User jlee
    ProxyJump bastion.company.com
    LocalForward 5432 localhost:5432
```

With this saved, `ssh db-server` alone does everything the full command above did.

> **A FORGOTTEN OR OVER-PERMISSIVE TUNNEL IS ITSELF A SECURITY HOLE**
>
> Port forwarding, used carelessly, can create the exact problem it's meant to avoid: forwarding a sensitive internal service and then forgetting the tunnel is still open, or explicitly binding a forward to `0.0.0.0` instead of the (usual, safe) localhost-only default — accidentally exposing the forwarded port to your entire local network rather than just your own machine. Close tunnels when the diagnostic work is done, and don't override the default bind address without a real, specific reason.

Working Example: A Production Database, Reachable Only Through a Bastion

A diagnostic query needs to run directly against a production database that's only reachable via an internal bastion, and only listens on `localhost` on the database server itself:

```
$ ssh -J bastion.company.com -L 5432:localhost:5432 db-server.internal
# tunnel now open in this terminal — leave it running

# in a second terminal:
$ psql -h localhost -p 5432 -U readonly_user production_db
production_db=> SELECT pid, now() - query_start AS duration, state FROM pg_stat_activity WHERE state != 'idle';
# — the exact same appdiag1-style read-only diagnostic query, now running against production directly

# when finished:
$ Ctrl+C  # closes the tunnel in the first terminal
```

A read-only diagnostic query, run against a database that was never directly exposed, using exactly the sanctioned path — and the tunnel explicitly closed the moment the work is done.

Hands-On Exercises

EXERCISE 1

Explain why this chapter frames a database only listening on `localhost` as a deliberate security control, not an obstacle to work around.

View solution

EXERCISE 2

Explain why organizations use a jump/bastion host instead of allowing direct SSH access to every internal server, using this chapter's own reasoning.

View solution

EXERCISE 3

Explain how a forgotten or over-permissive port forward can become a security hole, using the two specific examples this chapter names.

View solution

CHAPTER 3 QUICK REFERENCE

- **Local forwarding** (`-L`) reaches a service on the remote side that isn't directly exposed to you
- A service not being directly reachable is often a **deliberate security control** — port forwarding works with it, not around it
- **Remote forwarding** (`-R`) exposes something local to the remote side — the reverse direction, less common here
- **Jump/bastion hosts** (`-J` / `ProxyJump`) concentrate exposed attack surface onto one hardened, monitored machine
- `~/.ssh/config` with `ProxyJump` / `LocalForward` collapses a repeated multi-flag command into `ssh <host>`
- A forgotten open tunnel, or one bound to `0.0.0.0` instead of localhost, is a real, avoidable security risk — close tunnels when done
- **Next chapter:** Remote Desktop Protocols: RDP, VNC & Modern Alternatives

CHAPTER 4 OF 10

Remote Desktop Protocols: RDP, VNC & Modern Alternatives

Remote Support Tools & Techniques

Chapter 4 · Remote Desktop Protocols: RDP, VNC & Modern Alternatives

Chapters 2 and 3 covered SSH — command-line access. This chapter covers the scenarios that genuinely need a GUI instead: a Windows admin tool with no CLI equivalent, a legacy application only usable through its desktop interface, or visually confirming a display issue no remote command can fully capture.

RDP: Windows-Native, Encrypted, Efficient

RDP (Remote Desktop Protocol) is built into Windows, and encrypted by default (TLS-based) in modern versions. A real, genuine technical advantage: RDP sends drawing instructions and screen deltas rather than raw video frames, generally making it more bandwidth-efficient than VNC's typical full-framebuffer approach. Practically, the target account usually needs explicit RDP access granted — membership in the Remote Desktop Users group — a standard, deliberate admin step, not something that happens by default.

VNC: Cross-Platform, But Check the Implementation

VNC works across Windows, macOS, and Linux — genuinely cross-platform, unlike RDP's Windows-native server side. But VNC's own security posture varies significantly by implementation: some VNC servers have historically shipped with weak encryption, or none at all, by default. "VNC" isn't one single security guarantee the way modern RDP roughly is — it depends heavily on which specific server and client you're using and how it's configured.

A GENUINELY GOOD MITIGATION: TUNNEL VNC THROUGH SSH

Reusing Chapter 3's own port-forwarding technique directly: tunneling VNC traffic through an SSH port forward gets VNC's real cross-platform reach combined with SSH's own well-understood encryption and authentication guarantees — a practical, common way to close the gap a bare VNC connection can leave open.

Modern Alternatives: TeamViewer, AnyDesk, Chrome Remote Desktop

These tools are generally not what you'd reach for to administer infrastructure you already have RDP/VNC/SSH access to — they solve a genuinely different problem: an ad-hoc session with an end user's

own machine, often sitting behind NAT or a firewall with no direct network path available at all. Both sides connect *outward* to the vendor's own relay servers, meeting there — avoiding any need for inbound firewall rules on either end. This relay model is exactly what makes them practical for helping a random end user, and it's also what sets up Chapter 5's own "different trust model" material directly.

A Clear Decision Table

SITUATION	REACH FOR
Infrastructure you administer — a Windows server	RDP, through a VPN or jump host
Infrastructure you administer — cross-platform	VNC, tunneled through SSH
An end user's own machine, ad-hoc, no existing access path	A relay-based tool — TeamViewer, AnyDesk, Chrome Remote Desktop

NEVER EXPOSE RDP DIRECTLY TO THE INTERNET

RDP has been a major, persistent attack target specifically because it's so often left directly internet-facing — credential-stuffing campaigns and real, serious exploits (BlueKeep among them) have targeted exposed RDP endpoints repeatedly. Exactly the same principle from Chapter 3's own database example applies here: RDP should be reached through a VPN or a jump host, never opened directly to the internet just for convenience.

Working Example: A GUI-Only Disk Management Tool

A storage issue on a Windows server needs investigating with its own Disk Management tool — no convenient CLI equivalent exposes the same information, the kind of scenario **System Monitoring & Performance Diagnosis**'s own disk-space material assumes a shell for, but this specific tool genuinely doesn't have one. The correct path: connect through the organization's VPN or jump host first, then RDP to the target server from there — never RDP directly to a server exposed on the open internet, confirming the connection is going through the sanctioned path before doing anything else.

Hands-On Exercises

EXERCISE 1

Explain why this chapter says "VNC" alone isn't a reliable security guarantee the way modern RDP roughly is, and what practical technique it recommends to close that gap.

View solution

EXERCISE 2

Explain what makes relay-based tools like TeamViewer or AnyDesk genuinely different from RDP/VNC's own connection model, and why that difference makes them better suited to ad-hoc end-user support.

View solution

EXERCISE 3

Explain why this chapter says RDP should never be exposed directly to the internet, and what real-world evidence it cites for that risk.

View solution

CHAPTER 4 QUICK REFERENCE

- **RDP**: Windows-native, encrypted by default, efficient delta-based rendering, needs explicit access granted
- **VNC**: genuinely cross-platform, but security depends entirely on the specific implementation — tunnel through SSH for real safety
- **Relay-based tools** (TeamViewer/AnyDesk/Chrome Remote Desktop): both sides connect outward to a relay, no inbound firewall rules needed — built for ad-hoc end-user sessions, not infrastructure you administer
- Decision table: infrastructure you own → RDP/VNC-over-SSH through a VPN or jump host; someone else's machine, ad-hoc → a relay tool
- **Never expose RDP directly to the internet** — a real, historically exploited attack target; always go through a VPN or jump host
- **Next chapter:** Screen Sharing for End-User Support: A Different Trust Model

CHAPTER 5 OF 10

Screen Sharing for End-User Support: A Different Trust Model

Remote Support Tools & Techniques

Chapter 5 · Screen Sharing for End-User Support: A Different Trust Model

Chapters 2 through 4 all connected to infrastructure — the "user" of the system, in every meaningful sense, was the system itself, with nobody watching. This chapter is different in kind: there's an actual person present, often non-technical, whose own machine and whose own visible desktop are being accessed while they watch. Everything in this chapter follows from that one fact.

Informed Consent Isn't Just "They Clicked Allow"

Technically possible isn't the same as genuinely consented to. A user clicking "Allow" because a support agent told them to, without actually understanding what that agent can now see or do, isn't informed consent in any meaningful sense. Good practice: explain plainly, *before* starting — will you see their whole screen or just one window? Will you only be able to look, or will you be able to move their mouse and type? Say it before they grant access, not after.

View-Only vs. Full Control

Most screen-sharing tools offer both: view-only (you can see, not interact) and full remote control (you can move the mouse, click, type). Default to the least access that accomplishes the task — if you only need to see an error message or confirm a setting, view-only is enough and far less invasive. Only escalate to full control when you genuinely need to act yourself, ideally after explaining why. This is Chapter 6's own least-privilege principle, applied directly to a human-facing context rather than an infrastructure one.

Transparency During the Session

Narrate what you're doing as you do it — "I'm going to open your Control Panel now to check a setting" — rather than silently clicking around on someone else's machine. This keeps the user genuinely informed and comfortable, and it has a real secondary benefit: it helps them understand what actually happened, rather than leaving them a passive bystander on their own computer.

Ending the Session Cleanly

Fully disconnect at the end — don't just close your own window while a remote-control agent is still technically running and listening on their machine. Confirm the session shows as ended on both sides. If a persistent helper agent was installed specifically for this one session, uninstall it afterward if it isn't meant to remain.

THE SAME "FORGOTTEN OPEN TUNNEL" WARNING, IN A HUMAN-FACING FORM

Leaving unnecessary remote-access software running on someone's machine after a support session ends is a genuine security concern — an unused-but-active remote access path is itself an attack surface, exactly the same category of risk Chapter 3 flagged for a forgotten SSH tunnel, just installed on an end user's own personal machine this time.

NEVER ASK TO TYPE A USER'S OWN PASSWORD YOURSELF

If a password genuinely needs to be entered during a session, have the user type it themselves — many tools support briefly handing control back, or simply asking them to type it while the screen is momentarily obscured — rather than asking for it so you can type it in. A real, important practice specific to this human-facing scenario, distinct from anything in Chapter 2's own SSH-key material.

What to Do If Something Seems Off

If, during a session, something looks like a sign of a pre-existing compromise — unfamiliar browser extensions, obviously malicious software, other signs the machine may already be infected — this shifts the situation from routine support into something that needs escalating differently, per **Incident Response & Ticketing Workflows**' own escalation material. Worth acknowledging honestly rather than pretending it never happens, without turning this chapter into the still-outstanding, separately-scoped Security Basics for Support Technicians course.

Working Example: A Complete Support Session

BEFORE CONNECTING

"I'll be able to see your whole screen, and with your permission I can also move your mouse and type — I'll ask before doing that specifically."

STARTING VIEW-ONLY

The user shares their screen in view-only mode first, enough to see the error message and confirm the actual problem.

ESCALATING TO FULL CONTROL, BRIEFLY, WITH NARRATION

"I'll take control now to show you exactly where the setting is — opening Settings, then Display, then here." Each step narrated as it happens.

HANDING CONTROL BACK

Control returns to the user, who repeats the steps themselves to confirm they can now do it unassisted.

ENDING CLEANLY

The session is ended explicitly on both sides, confirmed as disconnected — no lingering agent left running afterward.

Hands-On Exercises

EXERCISE 1

Explain why this chapter says clicking "Allow" on a screen-share prompt isn't automatically the same as informed consent.

View solution

EXERCISE 2

Explain how the view-only vs. full-control choice in this chapter is described as an application of Chapter 6's own least-privilege principle.

View solution

EXERCISE 3

Explain why leaving a remote-access agent installed and running on an end user's machine after a session ends is described as the same category of risk as Chapter 3's forgotten SSH tunnel.

View solution

CHAPTER 5 QUICK REFERENCE

- An actual person is present and watching — this is genuinely different from Chapters 2–4's own infrastructure access
- Explain what you'll see and do **before** connecting — a technical "Allow" click isn't automatically informed consent
- Default to **view-only**; escalate to full control only when genuinely needed, and say why
- **Narrate what you're doing** as you do it — keeps the user informed, not a passive bystander
- **Disconnect cleanly** and remove any leftover access agent — an unused remote-access path is a real attack surface
- **Never type a user's own password yourself** — have them enter it, or use a proper reset flow
- Signs of a pre-existing compromise shift the situation into escalation territory, per `incident1`
- **Next chapter:** The Golden Rule: Least Privilege, Even for Yourself

CHAPTER 6 OF 10

The Golden Rule: Least Privilege, Even for Yourself

Remote Support Tools & Techniques

Chapter 6 · The Golden Rule: Least Privilege, Even for Yourself

Chapters 1 and 5 both previewed this: use the minimum access that accomplishes the current task, nothing more, even when broader access is technically sitting right there and available to you. "Even for yourself" is the important part — this applies to your own habits as a support engineer, not just what gets granted to other people.

Why Broad Access Is a Risk Even If You'd Never Misuse It

Least privilege isn't (only) about not trusting yourself. A broadly-privileged account is a bigger target and a bigger blast radius the moment something goes wrong with *your* access specifically — a stolen credential, a phished session, a compromised laptop — entirely independent of your own intentions. The real question isn't "would I misuse this," it's "what's the maximum damage if something goes wrong with my access, regardless of why."

Read-Only Where Read-Only Suffices

Many diagnostic tasks genuinely only need read access — checking logs, checking metrics, running a `SELECT`. Using a dedicated read-only account or role for these, rather than a full admin account that happens to also be capable of it, is a real, practical default. **Web & Application Troubleshooting**'s own worked examples already modeled this directly — its `readonly_user` connecting for a diagnostic query, not the application's own full database credentials.

Time-Limited Credentials Over Standing Access

Where the infrastructure supports it, short-lived credentials — a bastion host issuing a certificate valid for a few hours rather than a permanent keypair, a cloud provider's own temporary session tokens — are genuinely stronger than permanent standing access. If a short-lived credential leaks, the exposure window is naturally bounded; a permanent one keeps working until someone notices and explicitly revokes it. Not every organization has this kind of infrastructure built out yet, and standing access used with good discipline is still far better than no discipline at all — this describes a genuinely stronger option where it's available, not a judgment on anyone whose organization hasn't built it.

Elevation on Demand, Not Standing Root

```
$ systemctl status app-service
# read-only, no elevation needed

$ sudo systemctl restart app-service
# elevated for exactly this one command, then back to the normal account
```

Using `sudo` for the specific command that genuinely needs elevated privilege, rather than logging in as root directly or holding a permanent root shell, is the same "escalate only for the moment you need it" principle applied to everyday Linux access.

The Temptation to Over-Provision "Just in Case"

A genuinely understandable, common failure mode: reaching for broader access than a specific task actually needs because asking again later feels like friction, or "I might need it soon anyway." This undermines the whole principle through a thousand individually-reasonable-feeling small decisions, not one dramatic one. If a genuine, recurring need for broader access exists, that's worth requesting as real, properly-scoped standing access through the normal process — not quietly worked around by defaulting to an overpowered account out of convenience.

> **EXCESS PRIVILEGE DURING A ROUTINE TASK IS PURE, UNNECESSARY RISK**
> Using a full admin account for a routine, read-only diagnostic task means that if your session is compromised at that exact moment — a phishing attack succeeding on your own machine while that admin session happens to be open — the attacker inherits full admin access, not just the read access the task actually needed. The entire gap between what was used and what was actually necessary becomes pure downside, with zero corresponding benefit to the work being done.

Working Example: Diagnosing a Slow Service

BAD PRACTICE	GOOD PRACTICE
SSH in as root; run every check as root, including ones that never needed write access at all	SSH in with a standard, key-based account
Same root session used for database checks	A dedicated read-only database role for the diagnostic queries
Standing root shell held for the entire investigation	`sudo` reached for only the one specific command that genuinely needed it — restarting the service, if that turns out to actually be necessary — nothing more

Same diagnostic work, same actual outcome — a dramatically smaller blast radius the entire time it was happening.

Hands-On Exercises

EXERCISE 1

Explain why this chapter says least privilege matters even for someone who would genuinely never misuse their own access.

View solution

EXERCISE 2

Explain why time-limited credentials are described as genuinely stronger than standing access, and why this chapter is careful not to treat organizations without them as doing something wrong.

View solution

EXERCISE 3

Using this chapter's own bad/good comparison table, explain what changes about the actual risk if the support engineer's session were compromised during the investigation, under each approach.

View solution

CHAPTER 6 QUICK REFERENCE

- **Least privilege, even for yourself** — use the minimum access a task needs, not the broadest you happen to have
- The real risk isn't misuse — it's the **blast radius if your own access is compromised**, regardless of intent
- Use a **read-only role** where read-only genuinely suffices, rather than a full admin account out of convenience
- **Time-limited credentials** bound the exposure window if they leak — genuinely stronger where available, not a requirement everywhere
- `sudo` for the **one specific command** that needs it, not a standing elevated shell
- Watch for **"just in case" over-provisioning** — a real recurring need should become properly-scoped standing access, not a workaround
- **Next chapter:** Working Safely on a System You Don't Own

CHAPTER 7 OF 10

Working Safely on a System You Don't Own

Remote Support Tools & Techniques

Chapter 7 · Working Safely on a System You Don't Own

Earlier chapters covered getting access. This one is about what happens once you have it — specifically on a system that belongs to someone else's team, where the ordinary confidence of working on your own infrastructure doesn't automatically apply.

Confirming Authorization and Scope Before Touching Anything

Before making any change — not just diagnosing — on a system that isn't yours to freely administer, confirm explicitly: who actually owns this system, what are you authorized to do on it (read-only diagnosis? specific fixes? anything at all?), and is there a change-approval process that applies here. This is a genuinely different question from "do I have access" — having access and being authorized for a specific action are not the same thing.

The Trivially-Easy Mistake: Wrong Host, Wrong Session

Remote access tools make it genuinely easy to lose track of which session is connected to which host, especially with several near-identical terminal or RDP windows open side by side. A command meant for staging, accidentally run against production because the wrong window happened to have focus, is a classic and genuinely damaging mistake — not a rare, exotic failure.

A CHEAP, REAL MITIGATION: MAKE SESSIONS VISUALLY DISTINCT

A different terminal background color per environment — red or orange for production, green or blue for staging is a common real practice — or a shell prompt that prominently shows the hostname, gives your eyes something to catch before a mistake happens, not after.

A Practical Habit: Read the Prompt Before You Type

Literally look at the shell prompt — typically `user@hostname` — immediately before running any command that changes state, not just once at the start of a session. This is the single cheapest check against the wrong-host mistake, and it's worth building as a deliberate habit precisely because it's easy to stop doing once a session feels routine.

Dry Runs and Confirmation Prompts

Many tools support a dry-run or "show what would happen" mode — `rsync --dry-run`, a package manager showing its plan before applying it. Using these before the real action, especially on an unfamiliar or high-stakes system, catches a mistake before it happens rather than after. Related: don't habitually bypass built-in confirmation prompts (scripting `-y` or `--force` by default) specifically on systems you don't own or aren't deeply familiar with — the extra friction of a confirmation prompt is doing genuinely useful work in exactly this scenario.

When in Doubt, Ask — Even If It Feels Slow

On an unfamiliar system, especially one owned by another team, asking "can you confirm it's safe to do X here" before acting isn't a sign of incompetence — it's good practice, the same discipline **Incident Response & Ticketing Workflows** already applied to escalation timing: acting alone past the point where checking would be faster costs more than the moment spent asking.

Documenting What You Did, Even Outside a Formal Incident

Even for routine work — not a formal incident — briefly noting what you changed on a system you don't own, and when, and why, is genuinely valuable for the team who actually owns it, since they'll otherwise have no record of why their system's state changed. A lightweight version of Chapter 5's own live-timeline discipline, applied to any change on someone else's system, not only formal incident response.

> **THE CLASSIC "I THOUGHT I WAS ON STAGING" DISASTER**
>
> Running a destructive command — a database truncate, a service restart, a config overwrite — intended for a test environment, but actually connected to production because two similarly-named sessions were open side by side, is a genuinely common, real, damaging mistake. This entire chapter exists to prevent exactly this.

Working Example: A Close Call, Caught in Time

An engineer is about to run a cleanup script, and pauses — the practical habit from earlier in this chapter — to read the prompt first: `root@prod-db-3`, not the expected `root@staging-db-1`. The mistake is caught before anything runs. A brief, blameless note goes to the team anyway: what nearly happened, and why — not to assign fault, but because a near-miss is exactly the kind of information worth sharing, in the same spirit as **Incident Response & Ticketing Workflows**' own blameless review. Hiding it out of embarrassment would only mean the same near-miss is more likely to happen again, to someone else, with no warning.

Hands-On Exercises

EXERCISE 1

Explain why "having access" and "being authorized for a specific action" are described as genuinely different things in this chapter.

View solution

EXERCISE 2

Explain why this chapter recommends against habitually bypassing confirmation prompts on systems you don't own, even if doing so is normal practice on your own infrastructure.

View solution

EXERCISE 3

In this chapter's working example, explain why the engineer reported the near-miss even though no actual mistake occurred, and what principle from `incident1` this connects to.

View solution

CHAPTER 7 QUICK REFERENCE

- Confirm **authorization and scope**, not just access, before acting on a system you don't own
- Near-identical sessions make the **wrong-host mistake** genuinely easy — visually distinguish environments (color-coded prompts/backgrounds)
- **Read the prompt before every state-changing command**, not just once per session — the cheapest real safeguard
- Use **dry-run modes** where available; don't habitually bypass confirmation prompts on unfamiliar systems
- **Ask when in doubt** — the same anti-"escalating is failure" discipline as `incident1`
- **Document changes**, even routine ones, on systems you don't own — a lightweight version of Chapter 5's own timeline habit
- A caught near-miss is worth reporting **blamelessly**, not hiding — it's real, valuable information
- **Next chapter:** When the Connection Itself Is the Problem

CHAPTER 8 OF 10

When the Connection Itself Is the Problem

Remote Support Tools & Techniques

Chapter 8 · When the Connection Itself Is the Problem

Network Troubleshooting already taught the general diagnostic toolkit — DNS, ping, port checks, firewalls. This chapter isn't new technical territory so much as that same toolkit aimed at one specific, narrower question: "I'm trying to reach an SSH, RDP, or VNC service on a specific host, and it isn't working."

The First Question: Is It Me, or Is It Them?

The same scoping instinct `netdiag1` opens with, applied here directly: can you reach other things fine? If yes, the problem is specific to this one host or service, not your general connectivity. Testing from a genuinely different network — your phone's own mobile data, for instance — isolates whether the problem is your specific location or something about the target itself.

Common SSH Connection Failures, Precisely Read

Reusing `netdiag1`'s own "refused vs. timed out" distinction (Chapter 6), applied specifically to SSH:

```
$ ssh jumphost.company.com
ssh: connect to host jumphost.company.com port 22: Connection refused

$ ssh jumphost.company.com
ssh: connect to host jumphost.company.com port 22: Operation timed out

$ ssh jumphost.company.com
jlee@jumphost.company.com: Permission denied (publickey).
```

ERROR	WHAT IT ACTUALLY MEANS
Connection refused	Nothing is listening on port 22, or a firewall explicitly rejected it — `netdiag1`'s own network-level distinction
Operation timed out	A firewall silently dropping the connection, or a genuine network-path problem — also `netdiag1`'s territory
Permission denied (publickey)	The network connection and SSH handshake both actually succeeded — this is an authentication failure, not a connectivity one

A GENUINELY DIFFERENT CATEGORY, EASY TO CONFLATE

"Permission denied" means you reached the server just fine — the problem is specifically that your key isn't in `authorized_keys`, the wrong key is being offered, or the wrong username was used. This is Chapter 2's territory, not `netdiag1`'s — treating it as a network problem sends the investigation in a completely wrong direction, even though it also happened "at connection time."

A Practical Checklist, In Order

1. Can I reach anything else at all? (scope)
2. Can I reach the host at the network level — ping, or a route that completes? (`netdiag1` Chapter 5)
3. Is the specific port actually open? (`netdiag1` Chapter 6 — `nc -zv host 22` for SSH, `nc -zv host 3389` for RDP)
4. If the port is open but the connection still fails — is this actually an authentication problem, not a network one? (Chapter 2's own `authorized_keys` material)

RDP-Specific Connection Failures

RDP has its own distinct failure modes worth recognizing on sight, rather than mistaking for a network problem: "The remote computer requires Network Level Authentication, which your computer does not support" is a real, specific, named error indicating a client/server security-mode mismatch — not a connectivity issue at all. Similarly, a connection refused specifically due to a server already at its concurrent RDP session limit is a licensing/capacity condition, again nothing to do with the network path.

DON'T BLAME THE FIREWALL FIRST

Jumping straight to "the firewall must be blocking me" is a common overreaction. The actual cause is very often something far more mundane — a typo in the hostname, the service genuinely being down, the wrong port, an expired VPN session. The same "check first, don't reach for the most dramatic-sounding explanation" discipline this entire subject has built on applies here too.

Working Example: "SSH to the Jump Host Stopped Working This Morning"

Working the checklist: other services reach fine — not a general network problem. A ping to the jump host times out. A traceroute shows the path dying at the exact same hop that always used to pass through cleanly — genuine network-path evidence, not an SSH-specific issue at all. This gets escalated as a plain `netdiag1`-style network finding to the network team, rather than treated as an SSH problem — "I can't SSH in" often turns out to be an ordinary network-layer problem wearing an SSH-shaped disguise, and the diagnostic techniques from that course apply completely unchanged.

Hands-On Exercises

EXERCISE 1

Explain why "Permission denied (publickey)" is described as a genuinely different category of failure from "Connection refused" or a timeout, even though all three happen at connection time.

View solution

EXERCISE 2

Explain why the RDP "Network Level Authentication" error shouldn't be diagnosed the same way as a network connectivity failure.

View solution

EXERCISE 3

In this chapter's working example, explain why the "SSH stopped working" ticket was escalated as a network finding rather than an SSH-specific one, and what evidence supported that conclusion.

View solution

CHAPTER 8 QUICK REFERENCE

- This chapter applies `netdiag1`'s own toolkit to one narrow question: why can't I connect *in*
- Scope first: can you reach other things fine? Test from a different network to isolate location vs. target
- **Refused/timeout = network-level** (`netdiag1` territory); **"Permission denied (publickey)" = authentication succeeded reaching the server, then failed** — Chapter 2's territory
- Ordered checklist: reach anything → reach the host → the specific port is open → is this actually auth, not network
- RDP has its own distinct errors (NLA mismatch, session-limit refusal) — neither is a network problem
- Don't jump to "the firewall" first — the mundane explanation is usually right
- **Next chapter:** Session Logging & Audit Trails for Remote Access

CHAPTER 9 OF 10

Session Logging & Audit Trails for Remote Access

Remote Support Tools & Techniques

Chapter 9 · Session Logging & Audit Trails for Remote Access

This chapter closes the content chapters where it makes sense to: access itself is worth logging, for exactly the same "evidence over guesswork" reason this whole subject has been built on since `log1`'s own first chapter — not surveillance of the support engineer, the same principle applied one layer earlier, to how a session even started.

What Gets Logged, Typically

SSH connections land in `auth.log` or `secure` — `log1`'s own territory for where these actually live. RDP connections show up in Windows Event Viewer's own Security log, with specific event IDs for logon and logoff. For genuinely privileged or sensitive access, some organizations go further with full session recording — keystroke logging or video-style capture — through dedicated privileged-access-management tooling. Worth being honest: full session recording is a more invasive, higher-scrutiny practice, typically reserved for genuinely sensitive access, not blanket surveillance of all routine remote work.

Reading Your Own Access History Like Evidence

```
$ journalctl -u ssh --since "2026-08-09 08:00" --until "2026-08-09 10:00"
```

Checking your own recent connections is a legitimate, useful diagnostic step in its own right — confirming exactly when you connected relative to some other event, or confirming a session actually ended cleanly, the same "confirm it's actually disconnected" discipline Chapter 5 covered for screen sharing, now applied to SSH and RDP as well.

Using Session Logs During an Incident

During a genuine incident, remote-access session logs are themselves timeline evidence, per `log1`'s own material — "who connected to this system, and when" can directly corroborate or contradict a hypothesis, the same way any other log source does. Worth noting this cuts both ways: confirming *no* unauthorized session

occurred during a suspicious window is itself genuinely valuable, reassuring evidence — session logs are useful whether or not something bad is actually found.

Session Logging Is a Deterrent Too, Not Just a Record

Knowing that access is logged is itself a mild, healthy deterrent against exactly the kind of "just this once" shortcuts this course has warned against — sharing a private key, using a shared account, skipping least privilege. Not because anyone assumes bad faith by default, but because visibility genuinely does change behavior for the better, the same way any other accountability mechanism does. Worth stating honestly, rather than pretending logging exists purely for forensics after something has already gone wrong.

Privacy and Proportionality

Session logging should scale with the actual sensitivity of the access involved — routine read-only diagnostic access on a low-sensitivity system doesn't need the same scrutiny as privileged access to a system handling sensitive data. Organizations should be transparent with engineers about what's actually being logged and why — the same "know your organization's own policy" advice Chapter 1 opened this course with.

CLOSING THE LOOP ON CHAPTER 2: SHARED KEYS MAKE LOGS MEANINGLESS

A session log is only as trustworthy as the identity behind each connection. This is yet another, final reason a shared private key is so damaging: with proper per-person keys, "who connected" in the log is real, reliable evidence. With a shared key, the log entry is present, but it tells you nothing meaningful about which actual person was behind it — the exact same accountability gap Chapter 2 warned about, showing up again here in its most direct form.

Working Example: "Nobody Remembers Changing This"

A sensitive configuration file changed overnight; nobody remembers doing it. Checking session logs for the exact time window shows exactly one SSH session, from one specific engineer's own key, at precisely the right time. Confirmed directly with that engineer — who then remembers making a quick, legitimate change and simply forgetting to mention it. A benign resolution, not a security incident — a realistic reminder that most mysteries logs resolve end this way, not with "aha, a breach," but with "oh right, that was me."

Hands-On Exercises

EXERCISE 1

Explain why this chapter says session logging is not surveillance of the support engineer, and what principle it says it's actually the same as.

View solution

EXERCISE 2

Explain why this chapter says a shared private key makes a session log meaningless, connecting it back to Chapter 2's own reasoning.

View solution

EXERCISE 3

Explain why this chapter's worked example is described as a realistic reminder, given that it didn't turn out to be a security incident at all.

View solution

CHAPTER 9 QUICK REFERENCE

- SSH connections log to `auth.log` / `journalctl`; RDP logs to Windows' own Security event log
- Full session recording is more invasive, reserved for genuinely privileged/sensitive access, not blanket practice
- Reading your own access history is legitimate, useful evidence — confirming timing, confirming a clean disconnect
- Session logs are **incident timeline evidence** too — useful whether or not something bad is actually found
- Logging is a **deterrent**, not just a record — visibility genuinely changes behavior
- Logging should be **proportional** to access sensitivity, and organizations should be transparent about it
- A **shared private key makes "who connected" meaningless** — the final, clearest payoff of Chapter 2's own warning
- **Next chapter:** Capstone: Three Remote Support Scenarios, Start to Finish

CHAPTER 10 OF 10

Capstone: Three Remote Support Scenarios, Start to Finish

Remote Support Tools & Techniques

Chapter 10 · Capstone — Three Remote Support Scenarios, Start to Finish

Nine chapters built the toolkit — SSH keys, tunneling, jump hosts, RDP/VNC, screen sharing's own trust model, least privilege, working safely on systems you don't own, diagnosing connection failures themselves, and session logging. This capstone returns to the three-scenario shape this subject's other diagnostic courses use, since server SSH access, GUI remote desktop, and end-user screen sharing are genuinely different tools that don't chain into one continuous story.

Scenario 1: Diagnosing a Slow Nightly Report Job via SSH

The nightly report-generation service is running unusually slowly, on a production server reachable only through a jump host.

CONNECTING (CHAPTERS 2–3)

Connecting with a personal, key-based account — never a shared credential — through the bastion: `ssh -J bastion.company.com -L 5432:localhost:5432 report-db.internal`, tunneling the report database's own localhost-only port to run diagnostic queries directly.

A BRIEF CONNECTION HICCUP (CHAPTER 8)

The first attempt to reach the bastion times out. Rather than assuming an SSH problem, the checklist runs first: other services reach fine, and the cause turns out to be a quietly-expired VPN session — reconnecting the VPN resolves it immediately, confirming this was never an SSH-specific issue at all.

LEAST PRIVILEGE AND A SAFETY CHECK (CHAPTERS 6–7)

Diagnostic queries run through a dedicated read-only database role, not the application's own credentials. Before running anything that changes state, the prompt is read to confirm `report-db-prod-1`, not `report-db-staging-1`.

RESOLVING AND CONFIRMING (CHAPTER 9)

The slowdown traces to a connection-pool issue resolved with a targeted fix. Afterward, session logs confirm exactly when the diagnostic connection started relative to the slowdown itself, for the ticket's own record.

Scenario 2: A GUI-Only Disk Check via RDP

A Windows file server shows intermittent disk errors; confirming the specific issue needs its own Disk Management tool, with no convenient CLI equivalent.

REACHING IT THE SANCTIONED WAY (CHAPTER 4)

Connecting through the organization's VPN first, never RDP exposed directly to the internet — the same reasoning as Chapter 3's own database example, applied to RDP.

A RECOGNIZED, NON-NETWORK ERROR (CHAPTERS 4 AND 8)

The first attempt fails: "The remote computer requires Network Level Authentication, which your computer does not support." Recognized immediately as a client-side security-mode mismatch, not a connectivity problem — the client is reconfigured to support NLA, and the retry succeeds.

SCOPED ACCESS AND A HOSTNAME CHECK (CHAPTERS 6–7)

Logged in with a standard, admin-capable account only for the duration needed. Before touching any disk settings, the session's own title bar is checked to confirm it's genuinely connected to the intended server.

The Disk Management tool confirms a specific drive is degrading — handed off to the hardware team with the exact diagnostic evidence, rather than a vague "something's wrong with storage."

Scenario 3: Helping an End User Find a Settings Menu

A non-technical user can't locate a specific setting in their email client.

CONSENT AND LEAST ACCESS (CHAPTERS 5–6)

Before connecting: "I'll be able to see your screen, and only take control if you'd like me to — I'll ask first." The session starts view-only, enough to see exactly where the user is stuck.

NARRATED FULL CONTROL, BRIEFLY (CHAPTER 5)

With permission, control is taken briefly, narrating each step: "I'm opening Settings now, then Accounts, then here." Control is handed back immediately afterward so the user can repeat it themselves.

A PASSWORD MOMENT, HANDLED CORRECTLY (CHAPTER 5)

The setting requires re-entering an account password. The agent explicitly does not ask for it — the user types it themselves while the screen is briefly obscured.

CLEAN TEARDOWN (CHAPTERS 5 AND 9)

The session ends explicitly, confirmed disconnected on both sides. A lightweight helper the tool installed for this one session is removed afterward. The vendor tool's own session log — even for this ad-hoc, relay-based connection — records the start and end time, available if anything about the session is ever questioned later.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Framing: three genuinely different "getting there" scenarios	Chapter 1
Personal, key-based auth — never a shared credential (Scenario 1)	Chapter 2
Jump host plus port forwarding to reach a localhost-only database (Scenario 1)	Chapter 3
RDP through a VPN, never exposed directly; recognizing the NLA error (Scenario 2)	Chapter 4

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Informed consent, view-only default, narrated full control, never typing the user's password, clean disconnection (Scenario 3)	Chapter 5
Read-only DB role and standard accounts, never broader than the task needs (Scenarios 1–3)	Chapter 6
Reading the prompt/session title before acting (Scenarios 1–2)	Chapter 7
Diagnosing a VPN-expiry connection failure and a non-network RDP error correctly (Scenarios 1–2)	Chapter 8
Checking session logs for timing confirmation and session metadata (Scenarios 1 and 3)	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No specific vendor-tool tutorials (exact TeamViewer/AnyDesk UI walkthroughs) — the underlying principles transfer, specific interfaces change too often to document here
- No enterprise privileged-access-management (PAM) tool setup or administration — a genuinely separate, deeper discipline
- No VPN client configuration or troubleshooting in depth — this site's own `vpn1` course covers that directly
- No compliance-framework-specific audit or retention requirements (SOC2, HIPAA, and similar) — organization- and regulation-specific, out of scope here
- No physical security considerations for on-site support — this course is specifically about *remote* access

Each is a legitimate, separate topic — not silently assumed solved by what this course actually covers.

Hands-On Exercises

EXERCISE 1

Explain why Scenario 1's connection failure was diagnosed as a VPN issue rather than an SSH problem, and which chapter's own checklist that reasoning came from.

View solution

EXERCISE 2

Explain why Scenario 2's NLA error was recognized immediately as not a network problem, and why treating it as one would have wasted time.

View solution

EXERCISE 3

Explain why the agent in Scenario 3 didn't ask the user for their password, and what they did instead.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Scenario 1: a jump host, port forwarding, a VPN-expiry connection issue correctly diagnosed, and least privilege throughout an SSH-based investigation
- Scenario 2: RDP reached only through a VPN, a non-network NLA error recognized on sight, a hostname check before touching anything
- Scenario 3: informed consent, view-only by default, narrated full control, a password never typed by the agent, a clean teardown
- The recurring theme across all ten chapters: **getting there is its own real skill, and doing it safely matters as much as doing it at all**
- This closes *Remote Support Tools & Techniques*, 10/10 chapters — the sixth complete course under the Technical Support subject