SqlCertForge Adoption Guide

TLS certificates for SQL Server, Reporting Services, and Power BI Report Server — provisioned, bound, verified, and renewed. A guide and complete command reference for management, DBAs, sysadmins, and compliance.

Detent Point LLC · detentpoint.com · September 6, 2026

Why this exists

I'm a DBA, and I never planned on becoming the certificate guy. It just kept landing on me.

Most of the time it started on the reporting side. We run a farm of Power BI Report Servers, and every so often I'd have to put new certificates on thirty of them. I'd get one done, then a second, then a third, and then some server would flat refuse the binding, with an error that told me almost nothing. So I'd start deleting bindings by hand with netsh, run it again, hope I'd cleared the right one, because the web service and the web portal share the certificate on the same port and clearing one takes out the other. I've lost hours on a single box that way, sometimes most of a day, with another twenty-seven still waiting. I opened a ticket with Microsoft once just to get the order of operations right, and the guy on the phone told me this comes up all the time, it isn't new, and they'd never found a way to hand customers the fix without it driving them up a wall.

The database engine handed me a different flavor of the same thing. When we needed encrypted connections, a couple of us DBAs ended up in territory nobody had asked us to learn: SANs, private-key permissions, secure versus less secure versus most secure, none of it lining up the way the docs implied. It went from one DBA reading the instructions, to a second DBA reading them, to the two of us sitting down together to work out what they actually meant. We got it working, but it was never made easy. And we still had the plain question nobody answered cleanly: one certificate for the whole cluster, or one per node? Then do that across thirty or forty clusters, with customers to schedule around.

I gave up on getting it right by hand pretty fast, and I couldn't get it to script cleanly either, so for a long time I just found my way through it. Later in my career I decided to take it on for real, as part of my own company. That's what this is.

It isn't only for DBAs, even though that's what I am. Years as a federal contractor taught me a certificate never moves on its own. There's a sysadmin or a certificate admin, an engineer who wires it into an automation or a release pipeline, a manager who signs off on it, and a compliance reviewer who has to be able to trust it. So the guide is written for them too. Find the part that's yours and read that.

What I wanted out of it isn't complicated. A DBA shouldn't have to become a PKI engineer just to put TLS on a SQL Server or a report farm. I didn't want to lose another day to a binding that won't take. And when a manager or an auditor asks whether the connection is really encrypted, you ought to be able to show them instead of giving your word.

It won't make the decisions for you. You still pick the certificate, the servers, and the policy that fits your shop. What it handles is the part that kept going wrong on me: the steps in the right order, on one box or fifty, with a check at the end that the endpoint is actually serving over TLS.

— Keith

SqlCertForge — Adoption Guide

How to take SQL Server and Reporting Services from unencrypted to verified TLS, and keep them there — written for the people who decide, the people who do, and the people who check.

What SqlCertForge is, in one paragraph

SqlCertForge provisions and binds TLS certificates for the **SQL Server database engine**, **SQL Server Reporting Services (SSRS)**, and **Power BI Report Server (PBIRS)**. It does the whole certificate chore — request, issue, install, permission, bind, and *confirm it actually serves* — as a small set of PowerShell commands, or as an MCP tool. It runs entirely inside your network: nothing about your certificates or servers is transmitted anywhere, and the licence is checked offline. Read-only audit commands are free; the commands that change a server are licensed.

Who each part of this guide is for

If you are…	Read	You care about
Management / a budget owner	"Why" and "The phases at a glance"	What it removes (a recurring, error-prone task and its outages), and what it costs
A DBA	"The phases," SQL-engine track	Getting the database engine encrypted correctly, including Always On
A sysadmin	"The phases," Reporting Services track	The cert store, URL reservations, service accounts, WinRM, restarts
Compliance / an auditor	"Evidence and controls"	What's enforced, and what evidence each step leaves

---

Why: the problem this removes

Encrypted connections stopped being optional. For most of SQL Server's life an unencrypted connection was the default, and encryption was something you switched on deliberately. That has been reversed, one client at a time:

Client	From version	Default
Microsoft ODBC Driver for SQL Server	18	`Encrypt` = Mandatory
Microsoft OLE DB Driver (`MSOLEDBSQL19`)	19	`Encrypt` = Mandatory
`Microsoft.Data.SqlClient`	4.0	`Encrypt` = True — Mandatory from 5.0
SQL Server Management Studio	20	Encryption = Mandatory

SQL Server Native Client, the last widely deployed driver that didn't behave this way, no longer ships with SQL Server 2022 or with SSMS 19 and later. Nothing on your servers has to change for this to arrive: a patch cycle, a driver update, a refreshed workstation image, a developer bumping a NuGet package — and connections that worked for years start failing with "*the certificate chain was issued by an authority that is not trusted.*" The server is exactly as it was. The client simply stopped assuming.

Microsoft's own guidance gives two ways out, in this order: provision a certificate the client trusts, or set `TrustServerCertificate=yes` — which the same guidance calls a temporary measure that leaves the connection open to an adversary in the middle. The second is the one most of us reach for first, because it takes ten seconds and the correct answer takes an afternoon per server. Closing that gap is what this tool is for.

The certificate work itself is a six-step job, and every step has a quiet failure mode. Two of them cause real incidents:

1. **The private-key permission is missed.** The certificate is installed and looks bound, but the account SQL runs as can't read the key — so SQL ignores it and traffic stays unencrypted, with no obvious error.
2. **"Force encryption" is turned on before clients are checked.** The server starts rejecting every unencrypted connection, and an application that wasn't ready goes dark.

Neither announces itself, so done by hand across an estate they are easy to hit more than once. SqlCertForge encodes the correct order, refuses the dangerous shortcuts by default, and — for Reporting Services — proves the site actually serves before calling a node done.

How much work this is depends entirely on how you are deployed. A standalone server, on a name that already matches its certificate, is a short job; if that describes your estate, you may not need much help. A cluster is a different exercise. On the engine side, a Failover Cluster Instance or an availability group needs the same certificate present on every node, a SAN covering the listener and every replica name clients use, the private-key permission granted per node, and the binding written per node. On the Reporting Services side, a scale-out behind a vanity name or a load balancer needs the certificate identical on every node, a URL reservation on every node, and — the part nobody warns you about — the web service and the web portal share a certificate on the same port, so clearing one binding takes the other down with it.

That last one is where the days go. It is the problem described in the foreword, and together with the driver shift above it is why the database engine and Reporting Services were built first, before anything else on the certificate surface. The Scope section later in this guide lists the rest of that surface honestly, including what SqlCertForge doesn't touch yet, and says how we decide what comes next.

The phases at a glance

```
1 ASSESS  →  2 PLAN  →  3 PROVISION  →  4 APPLY  →  5 VERIFY  →  6 OPERATE
(free)       (decide)   (issue/import)  (grant+bind) (prove)      (renew)
```

Phase 1 — Assess *(free, read-only, safe on any server)*

Inventory where you stand before changing anything. Every command here is free and read-only.

- **SQL engine:** `Test-SqlCertBinding -SqlInstance <name> -Node <n1,n2>` — bound thumbprint, Force Encryption state, expiry, per node.
- **Reporting Services / PBIRS:** `Get-RsHttpConfig -RsInstance <SSRS|PBIRS>` — reserved URLs, SSL bindings, live URLs; `Test-RsCertBinding` — the bound cert and its expiry.

Output of this phase: a list of servers, their current TLS state, and which certificates (if any) are expiring.

Phase 2 — Plan *(a decision, not a command)*

Two choices decide the rest:

1. **Where do certificates come from?** Your own AD Certificate Services CA (SqlCertForge requests and retrieves them), or an external provider / existing certs handed over as `.pfx`. Either is first-class; there's no third-party service to sign off.
2. **What are the targets?** Which instances, which report servers, which vanity names, standalone vs cluster vs Always On.

Phase 3 — Provision *(licensed)*

Turn a decision into an installed certificate.

- **With a CA:** `New-SqlCertRequest` → `Submit-SqlCertRequest`. A CA that queues for approval returns *Pending* — resume later; it's not an error.
- **Bring your own:** `Import-SqlCert -PfxPath … -Node …` installs the same certificate consistently on every node.

Phase 4 — Apply *(licensed)*

- **SQL engine:** `Add-SqlCertPrivateKeyAccess` (grant the service account Read on the key — the step that is easiest to miss, because nothing fails loudly when it is), then `Set-SqlCertBinding`.
- **Reporting Services / PBIRS:** `Set-RsUrlReservation` then `Set-RsCertBinding` (or do reserve + bind + restart + verify in one with `Install-RsConnectionCertificate`).

> **Or skip the atoms:** `Install-SqlConnectionCertificate` runs the whole SQL-engine chain (request → verify), and `Install-RsConnectionCertificate` runs the whole Reporting Services chain, on one server or many.

Phase 5 — Verify *(free)*

- **SQL engine:** `Test-SqlCertBinding` reads back the binding; confirm the *running* instance loaded it via the SQL error log line "successfully loaded for encryption."
- **Reporting Services / PBIRS:** `Test-RsHttpsEndpoint` makes a real HTTPS request and inspects the served certificate — proof the site works, not just that it's configured.

Phase 6 — Operate *(renewal)*

Certificates expire — typically once a year. **Scheduled renewal ships in v1.3.0.** Register one job per certificate type — SQL connection, SSRS, PBIRS — with `Register-SqlCertRenewalJob`, under Windows Task Scheduler, cron, or SQL Server Agent, renewing on the interval you set. The runner (`Invoke-SqlCertRenewalJob`) acts only when a certificate is within its expiry threshold, verifies the new certificate before it replaces the live one, and leaves the working certificate in place if anything fails. `Test-SqlCertRenewalJob` previews a run without changing anything.

Two worked paths

Path A — You run AD Certificate Services (one SQL server, end to end):

```
Install-SqlConnectionCertificate -CommonName sql01.corp.com `
    -CaConfig 'CA01\Corp Issuing CA' -CertificateTemplate WebServer `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER'
# then restart the SQL service in your window
```

Path B — You buy certificates (a PBIRS scale-out behind one vanity name):

```
Import-SqlCert -PfxPath \\share\reports.pfx -PfxPassword $pw -Node n1,n2,n3
Install-RsConnectionCertificate -Thumbprint $tp -VanityUrl 'reports.example.com' `
    -RsInstance 'PBIRS' -Node n1,n2,n3
```

Evidence and controls *(for compliance)*

Control	How SqlCertForge enforces it
Least privilege on the private key	The grant adds one *Read* rule for the SQL service account SID — nothing broader — and records the resolved SID.
No accidental exposure	Granting key access to broad principals (Everyone, Authenticated Users, BUILTIN\Users/Guests, Anonymous) is refused unless explicitly overridden.
No silent lockout	Force Encryption (reject unencrypted clients) is opt-in and never set implicitly.
Proof of function	Reporting Services binds are gated on a real HTTPS round-trip; an expired bound certificate is reported as a failure.
Nothing leaves the network	All work is local or to your own CA/servers; the licence is verified offline against a signature — no call home.
An auditable record	Every state-changing step returns a structured result (stage, status, detail, thumbprint, SID) suitable for a change/audit log.

What an auditor can check directly: the private-key ACL grants Read to the service account SID and nothing broader; every bound thumbprint is an in-policy, unexpired certificate; where required, Force Encryption is on *and* clients still connect (no silent downgrade); and each Reporting Services endpoint returns 200/401 over TLS with a certificate whose SAN covers its host.

What SqlCertForge does not do

- It does not create or manage your certificate authority, your DNS, or your service accounts — it uses them.
- It does not restart SQL Server or Reporting Services on its own for the database-engine bind; the restart is yours to schedule (Reporting Services installs restart RS as part of the verified flow, unless you pass `-SkipRestart`).
- It does not build availability groups or Reporting Services scale-outs; it provisions certificates onto instances that already exist.

Scope — every certificate in SQL Server, and which ones this handles

Two different things share the name "certificate" in SQL Server, and they have almost nothing to do with each other. Microsoft never gave them separate names, so they get used interchangeably in the documentation, in forum answers, and in tickets. Telling which one a problem belongs to is harder than it should be, and working that out the slow way is a large part of why this tool exists.

	Windows certificate store certificate	In-database certificate
Lives in	`Cert:\LocalMachine\My` — private key in a key file on disk, protected by an ACL	Inside a SQL database, listed in `sys.certificates`
Created by	AD Certificate Services, a public CA, `certreq`, `New-SelfSignedCertificate`, a `.pfx` import	`CREATE CERTIFICATE` (T-SQL)
Backed up by	`Export-PfxCertificate` / the certificate export wizard	`BACKUP CERTIFICATE … WITH PRIVATE KEY`
Standard	Full X.509 v3 — EKU, SAN, revocation	X.509 v1 fields only — no EKU, no SAN, no revocation
Protects	The connection, in flight	The data, at rest

SqlCertForge works entirely in the left-hand column, and within it on the two TLS surfaces below. It never issues a `CREATE CERTIFICATE`, never opens a database master key, and never touches anything in `sys.certificates`.

What SqlCertForge does

Surface	Certificate lives in	If it expires	SqlCertForge
Database Engine TLS — encrypts the TDS connection between client and instance. Standalone, Failover Cluster, or Always On.	`LocalMachine\My`; thumbprint recorded under `SuperSocketNetLib`	SQL Server keeps running — it validates only at configuration time. Clients that validate per connection (Power BI is called out by name) start failing.	**Full lifecycle.** Request, submit to a CA, import, grant the private key, bind, verify — and renew on a schedule.
SSRS / Power BI Report Server HTTPS — the report server web service and the web portal.	`LocalMachine\My`; bound through the Reporting Services WMI provider and HTTP.sys	The service keeps serving; browsers and API clients reject it.	**Full lifecycle**, including the URL reservation, and a functional check that the endpoint really serves over TLS.

Renewal for both ships in v1.3.0: a job per certificate type under Windows Task Scheduler, cron, or SQL Server Agent, which replaces the live certificate only after the new one verifies.

What SqlCertForge does not do

Everything below is real, is a certificate, and is outside what this tool touches today. It is here so that when you are staring at a certificate problem you can find it on the list and know straight away whether this is the tool for it.

Also in the Windows certificate store — but not ours

Surface	What it protects	If it expires	Status
SQL Server's **self-signed fallback certificate**	Generated by the engine at startup when no suitable certificate is configured. It is why the login packet is *always* encrypted, even on a bare install. It carries no server identity, and it is usually what a vulnerability scan is reporting when a certificate finding appears on a server nobody configured.	Microsoft documents no lifetime or renewal cadence for it.	**Displaced, not managed.** Provisioning a real certificate is what replaces it — that is the point of the exercise — but it is not something you can bind, export, or renew.
Always Encrypted column master key	Column encryption keys, client-side. The CMK is a certificate in the Windows store or a key in Azure Key Vault.	Not enforced. Rotate by policy; keep the old CMK for point-in-time restores.	Not handled today.
Analysis Services HTTPS (`msmdpump` via IIS)	The HTTP pump to SSAS.	Client-side rejection, as any IIS site.	Not handled today.
SSIS package signing	Package authenticity.	Not enforced.	Not handled today.
Client-side trust — root CA distribution, `TrustServerCertificate`, `HostNameInCertificate`	Whether the client believes the certificate you just installed. Recent drivers (OLE DB 19, ODBC 18) default to `Encrypt=Yes`, which is why an install that worked for years can start failing after a driver update, with nothing on the server having changed.	—	**Verified, not distributed.** The verify stage inspects the certificate actually served on the wire and confirms its SAN covers the name clients use, so you can see when trust is the missing piece. Getting your root CA into the client trust stores stays with your PKI.

Inside the database — a different world entirely

Surface	What it protects	If it expires	Status
Transparent Data Encryption (TDE)	Data files, log files, and backups at rest, via a certificate in `master` protecting the database encryption key.	Not enforced — an expired certificate still encrypts and decrypts. The real risk is losing it.	Not handled today.
Backup encryption (`BACKUP … WITH ENCRYPTION`)	An individual backup set.	Enforced asymmetrically: new encrypted backups are blocked, restores still work. Renewing changes the thumbprint and can strand old backups.	Not handled today.

Surface	What it protects	If it expires	Status
Cell-level / column-level encryption (`EncryptByCert` , `EncryptByKey`)	Individual column values.	Not checked — the built-in encryption functions ignore certificate expiry dates.	Not handled today.
Database mirroring / Always On endpoint authentication (`CREATE ENDPOINT … AUTHENTICATION = CERTIFICATE`)	Replica-to-replica authentication where Windows auth isn't available — cross-domain and workgroup availability groups. This is a different object from the certificate that encrypts client connections, even though both sit on the same instance.	**Enforced, and it is an outage.** The endpoint connection fails. The default `EXPIRY_DATE` is one year from creation.	Not handled today.
Service Broker transport and dialog security	Broker endpoints and message encryption between instances.	Enforced — expired certificates are skipped, and the dialog fails if encryption is required.	Not handled today.
Module signing (`ADD SIGNATURE … BY CERTIFICATE`)	Grants permissions to code rather than to users.	Not checked. The practical failure is that editing the module drops the signature.	Not handled today.

Keys, not certificates — the neighbours

Surface	What it is	Status
SSRS / PBIRS encryption key (`rskeymgmt`)	A symmetric key protecting stored credentials and connection strings in the report server database. It is a different object from the SSRS TLS certificate, though both are called "the Reporting Services key" in conversation and both are reached from the same console. It does not expire; it breaks on a service-account reset, a machine or instance rename, or a migration.	Not handled today. Worth having a backup of it before you migrate.
Service Master Key / Database Master Key	The root of SQL Server's encryption hierarchy; what in-database certificates are protected by. No expiry.	Not handled today.
EKM / Extensible Key Management (HSM, Azure Key Vault)	An external key store holding the asymmetric key that protects TDE instead of a local certificate. SQL Server never rotates it for you.	Not handled today.
PolyBase internal encryption	Certificates on the internal channel between SQL Server and the PolyBase engine. On Linux the external-services certificate is self-managed on a 365-day cycle; on Windows the mechanism is not documented in detail.	Not handled today.
Azure SQL Database / Managed Instance	Microsoft manages the service TLS certificates and the built-in TDE certificate, rotating them on their own schedule. There is nothing on the box for you to bind.	Not applicable — this tool provisions certificates onto instances you run.

> **The short version.** If the certificate lives in the Windows certificate store and its job is to encrypt a connection to the database engine or to a report server, SqlCertForge handles it end to end. If it lives inside a database and its job is to encrypt data at rest, it does not.

Where this goes next

The list above is a map, not a boundary. The two connection surfaces came first for the reasons set out at the start of this guide, and they were finished completely before anything else was started — that order was deliberate.

Which part comes next is decided by what customers actually need rather than by what looks good on a roadmap. So if one of the rows above is the one costing you time, tell us which. That is how it gets prioritised.

Function Reference

Every command SqlCertForge exposes, in full

How to read this reference

What follows is every command SqlCertForge exposes — nineteen of them — and each one is written six ways. You are not expected to read all six. Find the one that is yours and skip the rest.

Three of the six are about **who is reading**:

- **Manager** — what the command is for and what it changes on a server, without the syntax.
- **Practitioner** — the working detail: parameters, real examples, and what to watch for.
- **Learner** — the same ground for someone who hasn't done this before, with the background filled in rather than assumed.

Three are about **what has to be checked** before it runs anywhere real:

- **Software Approval** — the questions a review board asks: what it touches, what it needs, and what it sends anywhere (nothing).
- **Dependencies** — what has to be present for the command to work, and whether each item is required or optional.
- **Compliance** — what the command enforces, and what evidence it leaves behind afterwards.

The same reference reads better online

On paper all six views have to be printed one after another, so every command runs long even when only one of the six is the one you came for. That is the honest cost of a document you can hand to someone, print, or attach to a change record.

The web version doesn't have that problem. It is the same nineteen commands with the audience and the lens as tabs, so you pick your two and the rest gets out of the way:

detentpoint.com/sqlcertforge/wiki

Same words and the same examples — both this reference and those pages are generated from one set of source files, so neither can drift from the other. If you are reading at a desk rather than from a binder, that is the better way to use it. The pages here are for when the desk isn't an option.

SQL Database Engine

Connection certificates for the database engine

Install-SqlConnectionCertificate

> End-to-end orchestrator: takes a SQL Server from no TLS to a verified, encrypted endpoint — request, issue, install, permission, bind, and confirm, in the right order, on one server or many. **Domain:** SQL Database Engine · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7 (module or MCP tool)

Manager

Putting a valid TLS certificate on a SQL Server is a six-step chore, and every step has a way to fail quietly. This command does all six in the correct order and reports, at each step, whether it worked.

The payoff: encrypting a SQL Server stops being a half-day ticket a senior DBA has to babysit and becomes one reviewable command that runs the same way on one server or fifty. It removes the two mistakes that cause outages — binding a certificate the SQL service account can't actually read, and turning on "force encryption" before confirming clients can still connect.

Three things to know before you approve its use:

1. **It can use your existing certificate authority, or none at all.** If you run AD Certificate Services, it requests and installs the cert for you. If you buy certs elsewhere, hand it the `.pfx` and it does the rest. No new certificate vendor to sign off.
2. **Nothing about your certificates or servers leaves your network.** The whole process runs inside your perimeter. There is no call home.
3. **"Force encryption" is off unless you ask for it.** That switch makes the server reject unencrypted connections — powerful, and occasionally an outage. The command never flips it silently.

What success looks like — one server, end to end, using your own CA:

```
Install-SqlConnectionCertificate -CommonName sql01.corp.com `
    -CaConfig 'CA01\Corp Issuing CA' -CertificateTemplate WebServer `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER'
```

Outcome — a certificate requested, issued, installed, permissioned, bound, and verified serving over TLS. One command, one audit line per step.

The bottom line for a budget owner: read-only checks in this toolkit are free and unlicensed. This command is one of the paid actions. What you buy is the removal of a recurring, error-prone manual task and the outages it causes.

Practitioner

The end-to-end orchestrator. It sequences the six atomic functions — `New-SqlCertRequest` → `Submit-SqlCertRequest` → `Import-SqlCert` → `Add-SqlCertPrivateKeyAccess` → `Set-SqlCertBinding` → `Test-SqlCertBinding` — threading the issued thumbprint forward so Grant and Bind operate on the exact cert. Every stage returns a `SqlCert.Result`; a Failed or Pending stage stops the chain. It never throws.

What you actually need to remember:

1. `-Stage` **runs any subset.** Default is all six. Skip Request/Submit when you already have a `.pfx`.
2. **Pending is not failure.** A CA that queues the request returns Pending — resume later with the same command and it picks up where it stopped.
3. **Remote nodes need** `-Credential`**.** Import/Grant/Bind/Verify run over WinRM; Kerberos pass-through covers domain-joined, a credential covers workgroup/cross-domain.
4. `-ForceEncryption` **is a separate, deliberate switch.** It rejects every unencrypted connection. Verify first.

Recipe 01 — Bring your own PFX (no CA in the path)

```
Install-SqlConnectionCertificate -Stage Import,Grant,Bind,Verify `
    -PfxPath C:\certs\sql01.pfx -PfxPassword $pw `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER'
```

Imports the pfx, grants the service account read on the private key, binds, and confirms it serves. Four results, four green stages.

Recipe 02 — A remote server in a workgroup

```
$cred = Get-Credential
Install-SqlConnectionCertificate -Node sql02 -Credential $cred `
    -PfxPath \\share\sql02.pfx -PfxPassword $pw -Stage Import,Grant,Bind,Verify `
    -SqlInstance MSSQLSERVER -ServiceAccount 'CORP\svc_sql'
```

Every stage runs on sql02 over WinRM under the supplied credential.

Recipe 03 — Confirm only (did last month's bind survive?)

```
Install-SqlConnectionCertificate -Stage Verify -SqlInstance MSSQLSERVER
```

Reads the binding from the registry and reports the bound thumbprint and expiry. No change made.

Recipe 04 — Turn on Force Encryption, deliberately

```
Install-SqlConnectionCertificate -Stage Bind,Verify -SqlInstance MSSQLSERVER -ForceEncryption
```

Sets ForceEncryption=1 and confirms the bind. Restart the SQL service to apply; the instance then rejects unencrypted clients.

Recipe 05 — Resume a CA that queues requests for approval

```
# First call returns Pending on Submit; after the CA operator approves, re-run:
Install-SqlConnectionCertificate -CommonName sql01.corp.com `
    -CaConfig 'CA01\Corp Issuing CA' -CertificateTemplate WebServer `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER'
```

Pending stops the chain without error; the same command resumes from where it stopped.

Watchpoint — the bind needs a restart. Writing the thumbprint to `SuperSocketNetLib` is instant; SQL loads it on the next service start. Verify confirms the registry value; the encrypted connection is live after the restart.

Learner

Start here if certificates are new to you. A TLS certificate is how a SQL Server proves it is really itself and encrypts the traffic to it. Getting one onto a server takes six small steps, and this command walks all six.

Step	What happens
Request	Make a certificate signing request — a file that says "please issue a cert for this server name."
Submit	Hand that request to your certificate authority and get the issued certificate back.
Import	Install the certificate into the server's certificate store.
Grant	Give the SQL Server service account permission to read the certificate's private key. Skip this and SQL silently can't use the cert.
Bind	Tell SQL Server "use this certificate" by writing its fingerprint into the right registry key.
Verify	Check the binding really took.

The whole thing, one line, with your own CA:

```
Install-SqlConnectionCertificate -CommonName sql01.corp.com `
    -CaConfig 'CA01\Corp Issuing CA' -CertificateTemplate WebServer `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER'
```

You get six results back, one per step. Green all the way down means the server is encrypted.

The one that bites everyone is the **Grant** step. If the SQL service account can't read the private key, SQL starts, ignores the cert, and connections stay unencrypted with no obvious error. This command never skips Grant.

Software Approval

Question	Answer
Outbound network calls at run time?	**None to the vendor.** The command talks only to your own CA (if used) and the target SQL servers, inside your network.
What is installed?	A signed PowerShell module, or the same code as a .NET global tool: `dotnet tool install --global DetentPoint.SqlCertForge.Mcp`.

Question	Answer
Runtime	Windows, with Windows PowerShell 5.1 *or* PowerShell 7 — either is sufficient.
Privileges needed	Local admin on the target SQL host (writes to the machine cert store and HKLM). For remote targets, WinRM access under a credential you supply.
Certificate authority	Optional. Uses AD Certificate Services if you have it; otherwise you supply a `.pfx` from any source.
Data handled	Certificates and their private keys, on your own servers. Nothing about them is transmitted anywhere.
Licence	Read-only audit commands are free. State-changing commands (this one) are licensed; the licence is verified offline against a signature — no activation call.

The one line that matters for a security review: nothing leaves your network. The licence check is an offline signature verification, not a call home.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Reads the machine cert store, writes the SQL registry binding.	Yes
Local administrator on the SQL host	The cert store and `SuperSocketNetLib` live under HKLM.	Yes
The SQL Server service account name	Passed to Grant so that account can read the private key.	Yes (for Grant/Bind)
AD Certificate Services CA	Only for the Request/Submit stages. Skip both with a supplied `.pfx`.	No
WinRM to the target	Only when acting on a remote `-Node`. Local runs don't need it.	No (local)
A SQL service restart	The bound cert loads on the next start. Yours to schedule.	To go live

No hidden dependencies: no agent, no service, no database of its own, no internet. If the six things above are in place, it runs.

Compliance

Controls built in:

- **Least privilege on the private key.** Grant gives the SQL service account *Read* only — not full control — on the key file.
- **Broad principals are refused.** Granting key access to Everyone, Authenticated Users, BUILTIN\Users, BUILTIN\Guests, or Anonymous is blocked unless `-AllowBroadPrincipal` is explicitly passed. The refusal is the default.
- **Force Encryption is opt-in.** The switch that rejects unencrypted connections is never set implicitly.

- **Every stage is evidence.** Each step returns a structured `SqlCert.Result` — stage, status, detail, thumbprint, resolved SID — suitable for an audit log.

A real result an auditor can keep (from a live run):

```
Grant  Success  Granted Read to SQLSPNLAB\svc_sql_ao on: CLVLAB-NODE1
Bind   Success  Bound BD7B1179…9582C to: AOAG on CLVLAB-NODE1
       The certificate [Cert Hash(sha1) "BD7B1179…9582C"] was
       successfully loaded for encryption.
```

The thumbprint, the account, and the target are on the record. The SID the account resolved to is captured for the grant.

What to check in an audit:

1. The private-key ACL grants *Read* to the SQL service account SID and nothing broader.
2. The bound thumbprint matches an in-policy, unexpired certificate.
3. Where required, Force Encryption is on and clients still connect (no silent downgrade).

New-SqlCertRequest

> Generates a SAN certificate signing request (CSR) for SQL connection encryption — the first step, done to spec so the issued cert actually works with SQL Server. **Domain:** SQL Database Engine · **Risk:** Changes state (writes a request + private key) · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

Before a certificate authority can issue a certificate, it needs a request that states the server name and the technical properties SQL Server requires. Getting those properties wrong is the usual reason a freshly issued cert refuses to bind. This command writes a correct request every time, so the cert you get back is one SQL Server can use.

It changes nothing on SQL Server itself — it produces a request file (and the matching private key) on the machine. It is the opening move of provisioning, not the risky part.

Practitioner

Writes an INF (Server Authentication EKU, Digital Signature + Key Encipherment, RSA-2048/SHA256, CNG "Microsoft Software Key Storage Provider", exportable) and runs `certreq -new`. Returns a `SqlCert.Result` with the `.inf` and `.req` paths. Under `-WhatIf` nothing is written and Status is Skipped. Never throws.

Modern clients require a SAN — always supply `-DnsName`, or it defaults to the CN.

Recipe 01 — Request for one FQDN plus its short name

```
New-SqlCertRequest -CommonName 'sql01.example.com' -DnsName 'sql01.example.com','sql01'
```

Writes request.inf / request.req; hand the `.req` to `Submit-SqlCertRequest`.

Recipe 02 — Put the artifacts in a specific scratch folder

```
New-SqlCertRequest -CommonName 'sql01.example.com' -WorkPath 'D:\certreq\sql01'
```

Keeps the `.inf` / `.req` where your change record expects them.

Recipe 03 — Preview without writing anything

```
New-SqlCertRequest -CommonName 'sql01.example.com' -WhatIf
```

Status = Skipped; nothing is written — useful in a dry run.

Learner

A "certificate signing request" (CSR) is a small file that says: *here is the server name I want a certificate for, and here are its technical settings.* You send it to a certificate authority, which signs it and hands back the real certificate.

The technical settings matter. SQL Server insists a connection certificate be marked for "Server Authentication" and use a key it can read. This command fills all of that in for you, so you don't have to know the settings by heart.

Software Approval

Question	Answer
Outbound calls?	None. It writes local files and generates a keypair on the machine.
What it produces	An `.inf`, a `.req` (the request), and a private key in the machine key store.
Privileges	Local admin (writes to the machine key store).
Licence	Part of the provisioning (paid) path. Read-only audit commands remain free.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Runs `certreq` and writes the request.	Yes
`certreq.exe` (in-box on Windows)	Generates the CSR and key.	Yes (present by default)
A writable scratch path	Holds the `.inf` / `.req`.	Yes

Compliance

- The request pins **RSA-2048 / SHA-256** and the **Server Authentication** purpose — no weak-key or wrong-purpose certs enter the flow.
- The key is generated in the machine key store, not exported to disk in the clear.
- Under `-WhatIf` the command is a no-op, so a change-controlled environment can preview it safely.

Submit-SqlCertRequest

> Submits a CSR to an AD CS certificate authority and retrieves the issued certificate — handling the common case where the CA holds the request for approval. **Domain:** SQL Database Engine · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the step that turns a request into an actual certificate by asking your certificate authority to sign it. Two outcomes are normal: the CA issues immediately, or it queues the request for a human to approve. Both are handled — a queued request is reported as *Pending*, not an error, and can be picked up later. Nobody has to sit and watch it.

Practitioner

Runs `certreq -submit`. If the CA auto-issues, the cert is accepted and Status is Success with a node-independent thumbprint. If the CA holds the request, Status is Pending and the result carries the RequestId plus the exact retrieve command. Never throws.

`-CaConfig` is the CA **configuration string** (`CAHost\CA Common Name`), not the `/certsrv` URL — `certutil -dump` shows it.

Recipe 01 — Submit a request to an issuing CA

```
Submit-SqlCertRequest -CaConfig 'CA01\Example Issuing CA' `
    -RequestPath C:\certreq\sql01\request.req -CertificateTemplate WebServer
```

Success returns the issued thumbprint; Pending returns the RequestId and the retrieve command.

Recipe 02 — Choose where the issued .cer lands

```
Submit-SqlCertRequest -CaConfig 'CA01\Example Issuing CA' `
    -RequestPath C:\certreq\sql01\request.req -CertificateTemplate WebServer `
    -OutPath D:\certs\sql01.cer
```

Writes the issued certificate to a known path for your records.

Learner

A certificate authority (CA) is the trusted party that signs certificates. Submitting is handing your request to it and getting the signed certificate back.

Sometimes the CA issues right away. Sometimes an administrator has to approve it first — in that case you get a **Pending** result with a request number. That's not a failure; it means "waiting for approval." Once approved, the same flow retrieves the finished certificate.

Software Approval

Question	Answer
Outbound calls?	To **your own** AD CS certificate authority, inside your network. Never to the vendor.
What it produces	The issued `.cer`, accepted into the machine store.
Privileges	Enrolment rights on the chosen CA template; local admin on the machine.
Licence	Part of the provisioning (paid) path.

Dependencies

Dependency	Why	Required?
An AD CS certificate authority	Signs the request.	Yes
A `.req` from `New-SqlCertRequest`	The thing being submitted.	Yes
Enrolment rights on the template	The CA rejects submissions without them.	Yes

Compliance

- The certificate is obtained from **your** enterprise CA under a template **you** control — issuance policy stays in your hands.
- A **Pending** result is surfaced honestly rather than being retried into a silent failure; the approval step remains a real, auditable gate.
- The accepted thumbprint is returned for the record and threaded forward so later steps bind the exact certificate that was issued.

Import-SqlCert

> Imports a `.pfx` into the machine certificate store on one or more nodes — with every node agreeing on the same thumbprint by construction. **Domain:** SQL Database Engine · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

When you buy a certificate from an external provider (or export one you already hold), it arrives as a password-protected `.pfx` file. This command installs it onto the servers that need it. For a cluster, it installs the *same* certificate on every node, so a failover never lands on a server missing the cert.

Practitioner

Uses native `Import-PfxCertificate` on the local node, or ships the PFX bytes over a PSSession and imports on the remote node. The thumbprint is read from the PFX file itself, so every node agrees by construction. Never throws.

Recipe 01 — Import to the local machine

```
Import-SqlCert -PfxPath C:\certs\sql01.pfx -PfxPassword $pw
```

Recipe 02 — Import the same cert to every cluster node at once

```
Import-SqlCert -PfxPath C:\certs\sql.pfx -PfxPassword $pw -Node SQLNODE01,SQLNODE02
```

Both nodes end up with the identical thumbprint — the point of reading it from the PFX.

Recipe 03 — A workgroup or cross-domain node

```
$cred = Get-Credential
Import-SqlCert -PfxPath \\share\sql.pfx -PfxPassword $pw -Node SQL-DMZ01 -Credential $cred
```

Kerberos pass-through doesn't apply off-domain, so supply a credential.

Learner

A `.pfx` file is a certificate together with its private key, wrapped in a password. Importing it puts the certificate into the server's store where SQL Server can find it.

The useful detail: this command reads the certificate's fingerprint (its "thumbprint") straight out of the file. That means when you import to several servers, they all reference the exact same certificate — no drift, no "which one did node 2 get?"

Software Approval

Question	Answer
Outbound calls?	None. Local import, or a direct PSSession to the servers you name.
What it produces	The certificate installed in `LocalMachine\My` on each node.
Privileges	Local admin on each target; WinRM for remote nodes.
Handling of the PFX password	Taken as a `SecureString`; not written to disk or logged.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Runs the import.	Yes
The `.pfx` and its password	The certificate being installed.	Yes
WinRM to remote nodes	Only when importing to a remote `-Node`.	No (local)

Compliance

- The PFX password is handled as a `SecureString` end to end.
- The installed thumbprint is returned, giving a record of exactly which certificate landed on which node.

- Multi-node imports are provably consistent — the thumbprint comes from the file, not from re-reading each node, so an auditor can confirm every node holds the same certificate.

Add-SqlCertPrivateKeyAccess

> Grants the SQL Server service account Read on the certificate's private key — the single step whose omission silently leaves a server unencrypted. **Domain:** SQL Database Engine · **Risk:** Changes state (an ACL grant) · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the least visible and most important step. A certificate is installed, it looks bound, everything appears fine — but if the account SQL Server runs as can't read the certificate's private key, SQL quietly ignores it and connections stay unencrypted. This command grants exactly the access needed, and nothing more, and refuses to grant it to a dangerously broad group.

Practitioner

Finds the target certificate (by exact thumbprint, or an anchored `CN=` match as a fallback), locates its private-key file — probing both the CNG key store and the legacy CAPI machine-key store — and adds a single **Read** rule for the named account. Existing permissions are left intact. Never throws.

> The dual-store probe matters: the "Microsoft RSA SChannel Cryptographic Provider" that SQL Server's TLS certs commonly use is surfaced through the CNG API yet writes its key file to `RSA\MachineKeys`. Both locations are checked, so the grant doesn't fail on a valid cert.

Recipe 01 — Grant the default instance's service SID

```
Add-SqlCertPrivateKeyAccess -ServiceAccount 'NT SERVICE\MSSQLSERVER' -Thumbprint $tp
```

Recipe 02 — A domain service account

```
Add-SqlCertPrivateKeyAccess -ServiceAccount 'CORP\svc_sql' -Thumbprint $tp
```

Recipe 03 — Select the cert by common name instead of thumbprint

```
Add-SqlCertPrivateKeyAccess -ServiceAccount 'NT SERVICE\MSSQLSERVER' -CommonName 'sql01.corp.com'
```

Recipe 04 — A remote node

```
Add-SqlCertPrivateKeyAccess -ServiceAccount 'CORP\svc_sql' -Thumbprint $tp -Node SQL02 -Credential $c
```

Guardrail — broad principals are refused. Passing a group like Everyone or Authenticated Users returns Failed with a hint; add `-AllowBroadPrincipal` only if that exposure is genuinely intended.

Learner

Every certificate has a private key — a secret file that proves the certificate really belongs to this server. Windows protects that file with permissions. SQL Server runs under a service account, and *that* account must be allowed to read the key, or SQL can't use the certificate.

This command adds exactly one permission: the service account gets **Read** on the key. Not full control, not "everyone" — just the one account, just read. That's the least-privilege way to do it.

Software Approval

Question	Answer
Outbound calls?	None. It edits a file permission on the local (or named) machine.
What it changes	Adds one Read ACE on the private-key file for the SQL service account.
Privileges	Local admin (to edit the key file's ACL).
Safety rail	Refuses broad principals (Everyone, Authenticated Users, BUILTIN\Users/Guests, Anonymous) unless explicitly overridden.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Reads the key store and edits the ACL.	Yes
The certificate present with its private key	There must be a key to grant access to.	Yes
The SQL service account name	The principal being granted Read.	Yes

Compliance

- **Least privilege by construction:** exactly one *Read* rule, for one resolved account SID, added to the existing ACL — nothing is widened.
- **Broad-principal refusal is the default**, so a careless grant to Everyone or Authenticated Users can't happen without an explicit, auditable override.
- The result records the resolved **SID** the grant was applied for — an auditor can confirm it matches the SQL service account and nothing broader.

Set-SqlCertBinding

> Binds a certificate to one or more SQL instances — the step that tells SQL Server which certificate to use. Works identically on standalone, Failover Cluster, and Always On instances. **Domain:** SQL Database Engine · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the step that actually points SQL Server at the certificate. It's a small, precise change — writing the certificate's fingerprint into the setting SQL Server reads at startup. It behaves the same whether the target is a plain server, a clustered instance, or an Always On availability group, so your team learns it once.

Practitioner

Writes the thumbprint into `SuperSocketNetLib` in the registry (StrictMode-safe instance-id resolution). Restarting the SQL service to apply is the caller's responsibility. Never throws. The write is per-instance and identical across topologies — an Always On replica is bound exactly like a standalone instance.

Recipe 01 — Bind the default instance

```
Set-SqlCertBinding -SqlInstance 'MSSQLSERVER' -Thumbprint $tp
```

Restart the SQL service to apply.

Recipe 02 — Bind a named instance and force encryption

```
Set-SqlCertBinding -SqlInstance 'HOST\SQL2019' -Thumbprint $tp -ForceEncryption
```

Force Encryption makes the instance reject unencrypted clients — validate in a test environment first.

Recipe 03 — Bind several instances in one call

```
Set-SqlCertBinding -SqlInstance 'MSSQLSERVER','HOST\SQL2019' -Thumbprint $tp
```

Recipe 04 — Bind on a remote node

```
Set-SqlCertBinding -SqlInstance 'MSSQLSERVER' -Thumbprint $tp -Node SQL02 -Credential $cred
```

Watchpoint — a restart is required. The registry write is instant; SQL loads the cert on the next service start. Confirm with `Test-SqlCertBinding`, and confirm the *running* instance loaded it via the SQL error log line "successfully loaded for encryption."

Learner

SQL Server decides which certificate to use by reading one registry value at startup. This command writes the certificate's fingerprint into that value. After the next service restart, SQL uses that certificate for encrypted connections.

Two things to remember: the change takes effect on **restart**, not instantly; and there's a separate switch, `-ForceEncryption`, that tells SQL to *require* encryption. Leave that off until you've confirmed clients can still connect.

Software Approval

Question	Answer
Outbound calls?	None. It writes one registry value on the local (or named) machine.
What it changes	The `Certificate` value under the instance's `SuperSocketNetLib` key; optionally `ForceEncryption`.
Privileges	Local admin (the key is under HKLM).
Reversibility	Fully reversible — clear the value and restart.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Writes the registry binding.	Yes
The certificate installed, with key access granted	SQL must be able to read the key it's pointed at.	Yes
A SQL service restart	The binding loads on the next start.	To go live

Compliance

- **Force Encryption is opt-in.** The switch that rejects unencrypted connections is never set implicitly, so an operator can't accidentally cut off clients.
- The bound thumbprint is returned for the record, and `Test-SqlCertBinding` reads it back independently for verification.
- Identical behaviour across standalone / FCI / Always On means one documented, reviewable procedure covers every SQL topology — no special-case scripts to audit.

Test-SqlCertBinding

> Reports the certificate binding SQL Server will use — bound thumbprint, Force Encryption state, and expiry — across one or many nodes. Read-only, and free. **Domain:** SQL Database Engine · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the check that answers "is this server actually set up to encrypt, and with a certificate that hasn't expired?" — without changing anything. It's part of the free, read-only tier, so a team can inventory every SQL Server's TLS state at no cost before deciding what to fix.

Practitioner

Reads the `SuperSocketNetLib` registry key for each instance and returns the bound thumbprint, the `ForceEncryption` flag, and certificate expiry. This reflects what SQL Server will use on the **next restart**. Returns Failed if the bound certificate has expired (SQL rejects an expired cert on restart). Never throws.

> For proof the *running* instance loaded the cert, check the SQL error log for "successfully loaded for encryption" — the registry value is intent; the error log is fact.

Recipe 01 — Check the default instance

```
Test-SqlCertBinding -SqlInstance 'MSSQLSERVER'
```

Recipe 02 — Check the same instance across cluster nodes

```
Test-SqlCertBinding -SqlInstance 'MSSQLSERVER' -Node 'NODE1','NODE2'
```

One call returns a row per node — useful for confirming a cluster is consistent.

Recipe 03 — A remote node with explicit credentials

```
Test-SqlCertBinding -SqlInstance 'MSSQLSERVER' -Node 'SQL-DMZ01' -Credential $cred
```

Learner

This command *looks* but never *touches*. It reads the setting that tells you which certificate SQL Server is configured to use, whether it's set to require encryption, and when that certificate expires.

It's the safe first thing to run: on a server you're not sure about, this tells you where you stand before you change anything. If it reports an expired certificate, that's a Failed result — a heads-up that a restart would break encryption.

Software Approval

Question	Answer
Outbound calls?	None. Registry reads on the local (or named) machines.
What it changes	Nothing — read-only.
Privileges	Read access to the instance registry; WinRM for remote nodes.
Licence	**Free.** Read-only audit commands need no licence.

Dependencies

Dependency	Why	Required?
Windows + PowerShell 5.1 or 7	Reads the registry and cert store.	Yes
WinRM to remote nodes	Only for remote `-Node` reads.	No (local)

Compliance

- A **safe, non-mutating** way to inventory TLS posture across an estate — ideal for a compliance sweep.
- Flags an **expired** bound certificate as a Failed result, surfacing a latent outage before it happens.
- Returns per-node/per-instance rows (InstanceName, Thumbprint, ForceEncryption, ExpiresOn, IsExpired, Node) — structured evidence for an audit record, no licence required to produce it.

Reporting Services & Power BI Report Server

SSRS and PBIRS HTTPS certificates

Install-RsConnectionCertificate

> One command for the common Reporting Services / Power BI Report Server case: reserve the vanity HTTPS URL, bind the certificate, restart, and confirm the site actually serves — on every node. **Domain:** SSRS / PBIRS · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

Getting HTTPS working on Reporting Services (or Power BI Report Server) by hand is fiddly and easy to leave half-done — a certificate can be bound while the site still returns an error, and nobody notices until a user complains. This command does the whole job on each server and then *checks the website actually loads over HTTPS with the right certificate*. A node is only "done" when that check passes.

It handles the multi-server case directly: point it at every node in a scale-out or cluster and each one is configured and verified independently.

Practitioner

For each node it: reserves the vanity HTTPS URL (RS applies the service-account ACL automatically — no netsh, no SID), binds the certificate (replacing a conflicting binding with `-Force`), restarts the RS service, then runs the functional check (`Test-RsHttpsEndpoint`). The final Verify result per node is authoritative — a node passes only when its site actually serves. Never throws.

The certificate must already be in `LocalMachine\My` on each node and valid for the vanity name (use `Import-SqlCert` to place it).

Recipe 01 — A three-node PBIRS scale-out behind one vanity name

```
Install-RsConnectionCertificate -Thumbprint $tp -VanityUrl 'reports.example.com' `
    -RsInstance 'PBIRS' -Node n1,n2,n3
```

Reserves the vanity URL, binds the SAN cert, restarts RS, and confirms the site serves on every node.

Recipe 02 — A single SSRS 2019 server, default port

```
Install-RsConnectionCertificate -Thumbprint $tp -VanityUrl 'reports.example.com' -RsInstance 'SSRS'
```

Recipe 03 — Configure only the web portal, on a non-default port

```
Install-RsConnectionCertificate -Thumbprint $tp -VanityUrl 'reports.example.com' `
    -RsInstance 'SSRS' -Application ReportServerWebApp -Port 8443
```

Recipe 04 — Bind now, restart in your own maintenance window

```
Install-RsConnectionCertificate -Thumbprint $tp -VanityUrl 'reports.example.com' `
    -RsInstance 'PBIRS' -SkipRestart
```

Everything is configured; you own the restart. The functional check will only pass after RS restarts.

Learner

Reporting Services and Power BI Report Server are websites that serve your reports. To make them use HTTPS you have to do three things on each server: reserve the web address, attach the certificate, and restart the service. Then — the step people forget — actually open the site and confirm it works.

This command does all four, in order, and treats the last one as the real test. If the website doesn't load over HTTPS with the right certificate, the node is reported as *not done*, even if the earlier steps "succeeded."

Software Approval

Question	Answer
Outbound calls?	None to the vendor. It configures the RS servers you name and makes a local HTTPS request to confirm each serves.
What it changes	A URL reservation, an SSL binding, and (unless skipped) an RS service restart, per node.
Privileges	RS admin + local admin on each node; WinRM for remote nodes.
Manual-netsh avoided	Uses the RS-native configuration API, so HTTP.sys and RS config stay in sync — no `netsh` drift.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or Power BI Report Server	The service being configured.	Yes
The certificate in `LocalMachine\My` on each node, valid for the vanity name	The cert to bind.	Yes
WinRM to remote nodes	For multi-node fan-out.	No (local only)
An RS service restart	The binding takes effect on restart.	Yes (unless `-SkipRestart`)

Compliance

- The **functional check is the gate** — a node passes only when the site genuinely serves over HTTPS with the expected certificate, so "configured but broken" can't be recorded as success.
- Uses the **RS-native configuration API**, keeping HTTP.sys and `rsreportserver.config` consistent — no out-of-band `netsh` changes for an auditor to reconcile.
- Emits one `SqlCert.Result` per node-stage, with the authoritative per-node Verify result — a clean, per-server evidence trail for a scale-out deployment.

Set-RsUrlReservation

> Reserves a URL for a Reporting Services endpoint the RS-native way — RS applies the correct service-account permission automatically, so you never compute an SID or SDDL. **Domain:** SSRS / PBIRS · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

Before a report server can answer at a web address, that address has to be "reserved" for it. Done by hand with `netsh`, this is the single most error-prone step — you have to hand-compute a permission string, and a wrong one means the site won't start. This command lets Reporting Services do it, so the permission is always correct.

Practitioner

Calls `ReserveURL` on the RS WMI configuration class. RS applies the correct ACL for its own service account automatically — you never supply an SDDL/SID. Use the strong wildcard `https://+:443` (default) to serve every host name that resolves to the box from one reservation; use a host-specific string only to deliberately restrict access. Never throws.

Recipe 01 — Reserve a vanity name for PBIRS

```
Set-RsUrlReservation -RsInstance 'PBIRS' -UrlString 'https://reports.example.com:443'
```

RS applies the service-account ACL automatically.

Recipe 02 — Strong wildcard for the SSRS web portal (covers every name that resolves)

```
Set-RsUrlReservation -RsInstance 'SSRS' -Application ReportServerWebApp
```

Defaults to `https://+:443`, covering machine name, vanity, cluster listener, and IP in one reservation.

Recipe 03 — Remote node with credentials

```
Set-RsUrlReservation -RsInstance 'SSRS' -UrlString 'https://reports.example.com:443' `
    -ComputerName RS02 -Credential $cred
```

Learner

A "URL reservation" tells Windows: *traffic for this web address belongs to this service.* Reporting Services needs one before it can serve HTTPS.

The hard part, historically, is a permission string tied to the service account — get it wrong and the site fails to start. This command sidesteps all of that by asking Reporting Services to make the reservation itself; RS knows its own account and sets the permission correctly.

Software Approval

Question	Answer
Outbound calls?	None. A WMI call to the local (or named) RS server.
What it changes	Adds a URL reservation in HTTP.sys via the RS configuration API.
Privileges	RS admin + local admin; WinRM for remote.
Manual-netsh avoided	Yes — no hand-computed SID/SDDL.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being configured.	Yes
The RS virtual directory	`ReserveURL` requires one; the command ensures it (idempotent).	Handled
WinRM to remote nodes	Only for remote targets.	No (local)

Compliance

- The **service-account ACL is applied by RS itself** — removing the classic manual-`netsh` mistake of an over-broad or incorrect URL permission.
- A **host-specific** reservation can deliberately restrict which names the server answers, when that's a requirement.
- Keeps HTTP.sys and RS configuration in sync, so what's reserved matches what RS believes is reserved — nothing hidden from a review.

Set-RsCertBinding

> Binds a TLS certificate to a Reporting Services HTTPS endpoint the RS-native way — updating RS config and HTTP.sys together, with a `-Force` path that replaces a conflicting binding without dropping to `netsh`.
> **Domain:** SSRS / PBIRS · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This attaches the certificate to the report server's web endpoint. It uses Reporting Services' own configuration interface, so the two places a binding lives — RS configuration and the Windows HTTP layer — stay in agreement. When an old, conflicting certificate is already in place, `-Force` replaces it cleanly.

Practitioner

Calls `CreateSSLCertificateBinding` on the RS WMI configuration class — updating `rsreportserver.config` and the HTTP.sys binding atomically. The RS service must be restarted to apply (caller's responsibility). Never throws; the RS instance is located automatically (no hard-coded version integer).

Recipe 01 — Bind on a SSRS 2017+ instance

```
Set-RsCertBinding -Thumbprint 'A1B2C3…A1B2' -RsInstance 'SSRS'
```

Binds to the web service endpoint on port 443.

Recipe 02 — The web portal on PBIRS, on a remote server

```
Set-RsCertBinding -Thumbprint 'A1B2C3…A1B2' -RsInstance 'PBIRS' `
    -Application ReportServerWebApp -Port 443 -ComputerName RSSERVER01 -Credential (Get-Credential)
```

Recipe 03 — Replace a conflicting binding

```
Set-RsCertBinding -Thumbprint $newTp -RsInstance 'SSRS' -Force
```

Without `-Force`, a different cert already bound to that IP:port returns Failed with a hint. With it, the old binding is removed via the RS API and the new cert bound.

Watchpoint — the SSL binding at an IP:port is shared. Every RS application bound there uses the same certificate, so `-Force` re-binds the new cert for **all** applications on that IP:port, not just the one named.

Learner

"Binding" a certificate means telling the web endpoint: *use this certificate to prove who you are and to encrypt traffic.* Reporting Services stores that in two places, and they must match. This command writes both through the official RS interface, so they can't drift apart.

If a different certificate is already bound to the same address and port, the command stops and tells you — unless you pass `-Force`, which replaces it. After binding, restart the RS service for it to take effect.

Software Approval

Question	Answer
Outbound calls?	None. A WMI call to the local (or named) RS server.
What it changes	The SSL certificate binding for the endpoint (RS config + HTTP.sys).
Privileges	RS admin + local admin; WinRM for remote.

Question	Answer
Manual-netsh avoided	Yes, including the conflict-replace path.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being configured.	Yes
The certificate in `LocalMachine\My`, not expired	The cert being bound.	Yes
An RS service restart	The binding takes effect on restart.	Yes

Compliance

- Writes through the **official RS configuration API**, keeping config and HTTP.sys atomically consistent — no `netsh` side-channel to reconcile.
- The **shared-IP:port** behaviour is documented and surfaced, so a `-Force` replacement's full effect is understood rather than a surprise.
- Refuses to silently overwrite a conflicting binding without `-Force`, keeping certificate replacement an explicit, auditable action.

Remove-RsUrlReservation

> Removes a Reporting Services URL reservation the RS-native way — keeping HTTP.sys and RS configuration in sync, unlike `netsh http delete urlacl`. **Domain:** SSRS / PBIRS · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

When a report server's web address changes — a retired vanity name, a decommissioned endpoint — the old reservation should be removed cleanly. Doing it with `netsh` removes it from Windows but leaves Reporting Services believing it still exists. This command removes it through Reporting Services, so both agree.

Practitioner

Calls `RemoveURL` on the RS WMI configuration class, keeping HTTP.sys and `rsreportserver.config` in sync — the supported alternative to `netsh http delete urlacl`, which touches only HTTP.sys. Never throws; failures return Failed. Use `Get-RsHttpConfig` first to see exactly what is reserved.

Recipe 01 — Remove a stale vanity reservation from the PBIRS portal

```
Remove-RsUrlReservation -RsInstance 'PBIRS' -Application ReportServerWebApp `
    -UrlString 'https://old-name.example.com:443'
```

Recipe 02 — Preview before removing (dry run)

```
Remove-RsUrlReservation -RsInstance 'SSRS' -Application ReportServerWebService `
    -UrlString 'https://+:443' -WhatIf
```

Status = Skipped; nothing is removed.

Recipe 03 — Remote server

```
Remove-RsUrlReservation -RsInstance 'SSRS' -Application ReportServerWebService `
    -UrlString 'https://retired.example.com:443' -ComputerName RS02 -Credential $cred
```

Learner

A URL reservation is the record that says a web address belongs to Reporting Services. When that address is no longer used, you remove the reservation.

The catch with the old `netsh` way is that it only cleans up Windows' copy — Reporting Services still thinks the address is reserved, and the two disagree. This command removes it through Reporting Services, so nothing is left dangling. Run `Get-RsHttpConfig` first to see the exact string to remove.

Software Approval

Question	Answer
Outbound calls?	None. A WMI call to the local (or named) RS server.
What it changes	Removes one URL reservation via the RS configuration API.
Privileges	RS admin + local admin; WinRM for remote.
Safety	Supports `-WhatIf` for a no-op preview.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being configured.	Yes
The exact reservation string	What's being removed (see `Get-RsHttpConfig`).	Yes
WinRM to remote nodes	Only for remote targets.	No (local)

Compliance

- Removes the reservation through the **RS API**, so config and HTTP.sys stay consistent — no orphaned state left behind for an audit to flag.
- `-WhatIf` allows a change-controlled preview before any removal.
- Pairs with `Get-RsHttpConfig` so the operator removes an exact, verified reservation rather than guessing.

Get-RsHttpConfig

> Inventories everything Reporting Services owns in the Windows HTTP layer — reserved URLs, SSL certificate bindings, and the real browseable URLs — in one read. Read-only, and free. **Domain:** SSRS / PBIRS · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This answers "what does this report server currently expose, and with which certificate?" in a single command, without changing anything. It replaces squinting at raw `netsh` output. It's in the free, read-only tier, so it's the natural starting point for auditing an estate before any work is planned.

Practitioner

Returns, in one read: reserved URLs (`ListReservedUrls` , including strong-wildcard `+` forms), SSL certificate bindings (`ListSSLCertificateBindings` — which thumbprint is bound to which application/IP/port), and registered URLs (`GetReportServerUrls` , with `+` resolved to real host names). Replaces eyeballing `netsh http show urlacl` / `show sslcert` . Never throws.

Recipe 01 — Inventory the local PBIRS instance

```
Get-RsHttpConfig -RsInstance 'PBIRS'
```

Lists every reserved URL, SSL binding, and registered URL.

Recipe 02 — Inventory a remote SSRS server

```
Get-RsHttpConfig -RsInstance 'SSRS' -ComputerName RS01 -Credential $cred
```

Recipe 03 — Pull just the SSL bindings for a report

```
(Get-RsHttpConfig -RsInstance 'SSRS').Data.SslBindings |
    Format-Table Application, CertificateHash, IPAddress, Port
```

Learner

Reporting Services keeps three related lists in Windows: the addresses it has reserved, which certificate is attached to each, and the actual URLs a browser would use. Reading these by hand means parsing cryptic `netsh` output.

This command gathers all three into one tidy result, so you can see at a glance what the report server exposes and which certificate secures it — a safe look with no changes.

Software Approval

Question	Answer
Outbound calls?	None. WMI reads on the local (or named) RS server.
What it changes	Nothing — read-only.
Privileges	RS admin (read); WinRM for remote.
Licence	**Free.** Read-only inventory needs no licence.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being inspected.	Yes
WinRM to remote nodes	Only for remote reads.	No (local)

Compliance

- A **non-mutating inventory** of the report server's HTTP surface — reserved URLs, bound certificate thumbprints, and live URLs — ideal for a compliance sweep.
- Surfaces exactly **which certificate** secures each endpoint, so an auditor can confirm it against policy.
- Free to run across an estate, so coverage isn't gated by licensing.

Test-RsCertBinding

> Reports the certificate bound to a Reporting Services endpoint — thumbprint and expiry — and flags HTTP-only mode or an expired cert. Read-only, and free. **Domain:** SSRS / PBIRS · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This tells you which certificate a report server endpoint is using and whether it's still valid — without changing anything. It also distinguishes "no certificate bound" (the server is running plain HTTP) from "bound but expired," which is a latent outage. Part of the free tier, so it costs nothing to sweep an estate.

Practitioner

Resolves the RS WMI object, reads the bound thumbprint via `ListSSLCertificateBindings`, looks the cert up in `LocalMachine\My`, and checks expiry. Returns Success with `Thumbprint=null` and an informational Detail when RS has no cert bound (HTTP-only). Returns Failed if RS can't be found, the bound cert has expired, or a WMI call fails. Never throws.

Recipe 01 — Check the default endpoint locally

```
Test-RsCertBinding
```

Default SSRS instance (MSSQLSERVER), ReportServerWebService, local machine.

Recipe 02 — Check the PBIRS web portal

```
Test-RsCertBinding -RsInstance 'PBIRS' -Application ReportServerWebApp
```

Recipe 03 — A remote SSRS 2017+ node with credentials

```
Test-RsCertBinding -RsInstance 'SSRS' -ComputerName $node -Credential $cred
```

Learner

This reads which certificate a report server endpoint is set to use, and when that certificate expires. It never changes anything.

Two useful signals: if nothing is bound, it tells you the endpoint is HTTP-only (not encrypted) — that's a Success with an "informational" note, not an error. If the bound certificate has already expired, that's a Failed result — a warning that the site will have certificate problems.

Software Approval

Question	Answer
Outbound calls?	None. WMI reads on the local (or named) RS server.
What it changes	Nothing — read-only.
Privileges	RS admin (read); WinRM/ `-Credential` for remote.
Licence	**Free.** Read-only audit needs no licence.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being inspected.	Yes
WinRM to remote nodes	Only for remote reads.	No (local)

Compliance

- Cleanly separates **HTTP-only** (nothing bound) from **bound-but-expired**, so a report of "not encrypted" is precise rather than ambiguous.
- Flags an **expired** endpoint certificate as Failed — surfacing a problem before users hit it.
- Returns structured data (RsInstance, Application, Thumbprint, ExpiresOn, IsExpired) — audit-ready, no licence required.

Test-RsHttpsEndpoint

> The functional round-trip check WMI can't give you: it discovers the real URL, makes an HTTPS request, and inspects the certificate actually served on the wire. Read-only, and free. **Domain:** SSRS / PBIRS · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

A configuration can look correct and the website still be broken — a certificate bound to the wrong name, a service that wasn't restarted, a stale reservation. This command does what a user would do: it opens the site over HTTPS and confirms it genuinely works, with the right certificate. It's the difference between "we configured it" and "it works." Free to run.

Practitioner

Discovers the real URL (`GetReportServerUrls`), issues an HTTPS GET, and inspects the served certificate. Healthy = TLS handshake succeeds **and** HTTP status is 200 or 401 (401 is the normal `RSWindowsNegotiate` challenge — the site is alive). Confirms the served cert's SAN covers the host and, with `-ExpectedThumbprint`, that the served cert is the one you bound. Never throws.

Recipe 01 — Confirm PBIRS serves at its vanity name

```
Test-RsHttpsEndpoint -RsInstance 'PBIRS' -VanityHost 'reports.example.com'
```

GETs the discovered path under the vanity host and confirms 200/401 with a SAN that covers it.

Recipe 02 — Assert the served cert is exactly the one you bound

```
Test-RsHttpsEndpoint -RsInstance 'SSRS' -Url 'https://reports.example.com/Reports' -ExpectedThumbprir
```

Fails if the wire certificate's thumbprint doesn't match `$tp`.

Recipe 03 — Test a remote node

```
Test-RsHttpsEndpoint -RsInstance 'SSRS' -VanityHost 'reports.example.com' `
    -ComputerName RS02 -Credential $cred
```

Note — `-VanityHost` is required with a wildcard reservation. A strong-wildcard (`+`) reservation can't tell you the vanity name, and testing `localhost` wouldn't validate the certificate's SAN — so name the host you want proven.

Learner

Everything up to now was configuration. This is the *test drive*: the command actually visits the report server's web address over HTTPS and checks it responds, with the correct certificate.

A friendly detail: report servers answer a valid request with a "401 — who are you?" challenge before you sign in. This command treats that 401 as "the site is alive and encrypted," because it is — a dead site

wouldn't get that far. It can also confirm the certificate on the wire is exactly the one you meant to bind.

Software Approval

Question	Answer
Outbound calls?	An HTTPS request to **your own** report server, to confirm it serves. Nothing to the vendor.
What it changes	Nothing — read-only.
Privileges	RS admin for URL discovery; WinRM/`-Credential` for remote.
Licence	**Free.** The functional check needs no licence.

Dependencies

Dependency	Why	Required?
SSRS 2016+ or PBIRS	The service being tested.	Yes
Network path to the endpoint	It makes a real HTTPS request.	Yes
WinRM to remote nodes	Only for remote URL discovery.	No (local)

Compliance

- **Proves encryption end to end** — a real TLS handshake and HTTP response, not just a configuration value — which is the evidence a control like "reporting is served over TLS" actually needs.
- Confirms the **served certificate's SAN covers the host**, catching the common "bound the wrong-name cert" mistake.
- With `-ExpectedThumbprint`, asserts the **exact** certificate on the wire — a precise, reproducible check for an audit.

Scheduled Renewal

Automatic renewal jobs across Task Scheduler, cron, and SQL Agent

Register-SqlCertRenewalJob

> Sets up automatic certificate renewal: saves a job spec and creates a schedule entry that renews a certificate before it expires — under Windows Task Scheduler, cron, or SQL Server Agent. **Domain:** Scheduled renewal (SQL / SSRS / PBIRS) · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

An expired TLS certificate is an outage that arrives on a schedule you didn't choose. This command removes that risk: it sets up a job that watches a certificate and renews it before it lapses, without anyone remembering to. You set it up once per certificate; it runs itself thereafter.

Three things to know before you approve its use:

1. **It fits your existing scheduler.** Whether your team lives in Windows Task Scheduler, cron, or SQL Server Agent, the job runs there — no new scheduling system to adopt.
2. **It runs under an identity you control.** The recommended setup is a managed service account (gMSA) that already holds the needed rights, so no password is stored anywhere.
3. **A failed renewal never causes an outage.** The job proves the new certificate works before it replaces the old one; if anything fails, the working certificate stays in place.

Practitioner

Writes a spec (`%ProgramData%\SqlCertForge\renewals\<JobId>.json` — no plaintext secret; a PFX password is a reference) and creates a schedule entry in the chosen backend. Two certificate sources — a CA (Request/Submit) or a supplied PFX — selected by parameter set. The scheduled runner (`Invoke-SqlCertRenewalJob`) does the work; the entry just triggers it.

Recipe 01 — SQL engine, CA-issued, Task Scheduler, under a gMSA

```
Register-SqlCertRenewalJob -JobId sql01-engine -CertType SqlConnection `
    -CaConfig 'CA01\Corp Issuing CA' -CertificateTemplate WebServer -CommonName sql01.corp.com `
    -SqlInstance MSSQLSERVER -ServiceAccount 'NT SERVICE\MSSQLSERVER' -RunAsUser 'CORP\gmsa_scf$'
```

Renews within 30 days of expiry (default), no stored secret.

Recipe 02 — PBIRS scale-out, PFX source, on SQL Server Agent

```
Register-SqlCertRenewalJob -JobId reports-pbirs -CertType Pbirs `
    -PfxPath \\share\reports.pfx -PfxPasswordRef ReportsPfxPwd `
    -VanityUrl reports.example.com -Node n1,n2 -Scheduler SqlAgent -SqlInstance RPT01
```

`-PfxPasswordRef` names a SecretManagement secret — the password itself is never in the spec.

Recipe 03 — Tighter threshold, SSRS, on cron

```
Register-SqlCertRenewalJob -JobId ssrs-portal -CertType Ssrs -ThresholdDays 45 `
    -PfxPath /certs/ssrs.pfx -PfxPasswordRef SsrsPfxPwd `
    -VanityUrl reports.example.com -RsInstance SSRS -Scheduler Cron
```

Watchpoint — the run-as identity needs the rights. The scheduled runner acts with no human present, so the account it runs as (`-RunAsUser`, ideally a gMSA) must hold CA-enrolment / local-admin / the service-account knowledge. If a backend's tooling isn't on the host, the command returns the schedule entry text to apply by hand rather than failing silently.

Learner

A "renewal job" is a saved note that says: *this certificate, on these servers, renew it when it's within N days of expiring, from this source.* This command writes that note and tells your scheduler to check it on a cadence.

You don't have to know the internals of Task Scheduler, cron, or SQL Agent — you pick one with `-Scheduler`, and the command sets up the entry. From then on, the certificate renews itself before it expires.

Software Approval

Question	Answer
Outbound calls?	None to the vendor. The job later talks only to your own CA (if used) and your servers.
What it writes	A spec file (JSON, **no plaintext secret**) and a schedule entry in your chosen backend.
Privileges	Local admin to create the schedule; the run-as identity holds the renewal rights.
Secret handling	A PFX password is a SecretManagement **reference**, resolved at run time — never stored in the spec.

Dependencies

Dependency	Why	Required?
A scheduler backend (Task Scheduler / cron / SQL Agent)	Triggers the renewal on a cadence.	Yes (one of)
A run-as identity (gMSA recommended)	The unattended runner acts as it.	Yes
A cert source (CA config, or a PFX + password reference)	What the renewal issues/imports.	Yes

Compliance

- **No plaintext secret is persisted** — the spec is safe to read; a PFX password is a vault reference.
- The renewal **never replaces a working certificate until the new one verifies** (safe-swap), so automation can't cause a silent outage.
- Every scheduled run appends a structured outcome to the run log — an audit trail of what renewed, when, and whether it succeeded.

Invoke-SqlCertRenewalJob

> Runs one renewal job — the entry point every scheduler calls. Renews only when the certificate is due (or `-Force`), and never takes a working server offline. **Domain:** Scheduled renewal (SQL / SSRS / PBIRS) · **Risk:** Changes state (only when a renewal is due) · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the worker your scheduler triggers. Each time it runs, it checks whether the certificate is close to expiring. If it isn't, it does nothing and exits — so running it daily is safe and cheap. When the certificate is due, it renews it, proves the new one works, and only then swaps it in. You rarely call this by hand; your scheduler does. But `-Force` gives you a "renew now" button when you need one.

Practitioner

Reads the spec, reads the live certificate's expiry (via the free `Test-*` commands), and decides: **NotDue** → no-op (Success, idempotent); **Unknown** (a failed read) → Skipped, because a failed read is not a licence to act; **DueNow** (or `-Force`) → renew via the validated orchestrators, whose own Verify stage is the safety gate. Appends the outcome to the run log. Never throws.

Recipe 01 — What the scheduler runs (no arguments beyond the job)

```
Invoke-SqlCertRenewalJob -JobId sql01-engine
```

A no-op on most days; renews on the day the certificate enters the threshold window.

Recipe 02 — Renew right now, regardless of expiry

```
Invoke-SqlCertRenewalJob -JobId reports-pbirs -Force
```

Skips the due-date check and renews immediately — a manual "renew now".

Recipe 03 — Preview what it would do (no change)

```
Invoke-SqlCertRenewalJob -JobId sql01-engine -WhatIf
```

Watchpoint — Unknown means skip, not fail. If the expiry can't be read (a transient outage on the target), the run reports Skipped and changes nothing, leaving the next run to retry — deliberately, so a blind read never triggers a needless re-issue.

Learner

This is the part that actually does the renewal, and it's smart about *when*. Most of the time it looks at the certificate, sees it's not close to expiring, and stops — that's why a daily schedule is fine. Only when the certificate is genuinely near its end does it renew.

And it's careful: it builds the new certificate and checks it works *before* replacing the old one. If the new one fails, the old — still-working — certificate stays. Your server never goes dark because a renewal went wrong.

Software Approval

Question	Answer
Outbound calls?	None to the vendor. On a renewal, only your own CA (if used) and your servers.
What it changes	Nothing on a no-op day; on a due day, it renews and re-binds a certificate.

Question	Answer
Privileges	Runs as the job's scheduled identity (a gMSA is recommended).
Auditability	Appends every run's outcome to the job's run log.

Dependencies

Dependency	Why	Required?
A registered renewal job (spec)	Tells it what to renew.	Yes
Read access to the target's current cert	To decide whether it's due.	Yes
The run-as identity's renewal rights	To act when due.	On a due run

Compliance

- Acts **only when a certificate is genuinely near expiry** (or on an explicit `-Force`) — no unnecessary re-issues.
- A failed expiry read yields **Skipped**, never a blind renewal.
- The **safe-swap invariant** holds: the live certificate is replaced only after the new one verifies, so scheduled automation cannot cause a silent outage. Each run is logged for audit.

Test-SqlCertRenewalJob

> Previews a renewal job — would it renew now, and why? — without changing anything. Read-only, and free.
> **Domain:** Scheduled renewal (SQL / SSRS / PBIRS) · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This answers "is this certificate about to be renewed, and when?" without touching anything. It's the safe way to confirm a renewal job is set up correctly and to see how many days remain before the next renewal. Part of the free tier, so it costs nothing to check across an estate.

Practitioner

Reads the spec, reads the live certificate's expiry (via the free `Test-*` commands), and runs the due-date gate — returning the decision (**DueNow / NotDue / Unknown**), days to expiry, and the reason. Makes no change; safe to run any time.

Recipe 01 — Would this job renew now?

```
Test-SqlCertRenewalJob -JobId sql01-engine
```

Returns e.g. Decision=NotDue, DaysToExpiry=180 — set up correctly, nothing due yet.

Recipe 02 — Confirm a new job reads the right target

```
(Test-SqlCertRenewalJob -JobId reports-pbirs).Data | Format-List Decision, DaysToExpiry, ExpiresOn
```

Learner

This *looks* but never *touches*. It reads how close the certificate is to expiring and tells you whether a run would renew it right now — DueNow (yes), NotDue (not yet), or Unknown (it couldn't read the expiry). Run it after setting up a job to confirm it points at the right certificate and sees a sensible number of days left.

Software Approval

Question	Answer
Outbound calls?	None to the vendor. It reads your own certificate's state.
What it changes	Nothing — read-only.
Licence	**Free.** Read-only preview needs no licence.

Dependencies

Dependency	Why	Required?
A registered renewal job	The spec it previews.	Yes
Read access to the target's cert	To report the decision.	Yes

Compliance

- A **non-mutating** way to confirm renewal coverage and see days-to-expiry — ideal for a compliance check that certificates are on an active renewal schedule.
- Surfaces the same decision the runner would make, so "will this renew in time?" is answerable without waiting for the scheduled run.

Get-SqlCertRenewalJob

> Lists the registered renewal jobs and each one's last recorded run. Read-only and offline, and free.
> **Domain:** Scheduled renewal (SQL / SSRS / PBIRS) · **Risk:** Read-only · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

This is the inventory: which certificates are on an automatic renewal schedule, on which scheduler, and how the last run went. It's the answer to "are all our certificates covered by a renewal job?" — a one-command roll call, no changes, no cost.

Practitioner

Reads the spec store. With `-JobId`, returns that one; otherwise all. For each job it returns the key spec fields (CertType, ThresholdDays, Scheduler, CreatedUtc) plus the most recent run-log line. **Offline and read-only** — it does not contact any server; use `Test-SqlCertRenewalJob` for a live "would it renew now" check.

Recipe 01 — All renewal jobs and their last run

```
Get-SqlCertRenewalJob | Format-Table JobId, CertType, Scheduler, @{n='LastRun';e={$_.LastRun.Status}}
```

Recipe 02 — One job in detail

```
Get-SqlCertRenewalJob -JobId sql01-engine
```

Learner

This lists the renewal jobs you've set up — like a table of contents for automatic renewals. For each, you see what it renews, which scheduler runs it, and how the last run turned out. It only reads local files, so it's instant and safe.

Software Approval

Question	Answer
Outbound calls?	None — it reads local spec and log files only.
What it changes	Nothing — read-only.
Licence	**Free.**

Dependencies

Dependency	Why	Required?
The local renewal spec store	The jobs it lists.	Yes

Compliance

- A **coverage report** for certificate renewal — an auditor can confirm every certificate that should auto-renew has a job, and see each job's last outcome.
- Offline and read-only, so it's safe to run anywhere and reflects exactly what's registered locally.

Unregister-SqlCertRenewalJob

> Removes a renewal job — its schedule entry, and (unless you keep it) its spec. The run log stays as history.
> **Domain:** Scheduled renewal (SQL / SSRS / PBIRS) · **Risk:** Changes state · **Runs on:** Windows, PowerShell 5.1 or 7

Manager

When a server is decommissioned or a certificate is handled some other way, this cleanly tears down its renewal job — removing the scheduled task/cron/Agent entry so nothing keeps trying to renew a certificate that no longer matters. It leaves the run log behind as a record of what the job did.

Practitioner

Loads the spec to learn its scheduler backend, removes the schedule entry, and deletes the spec file unless `-KeepSpec` is given. Best-effort and never throws; returns a descriptor of what was removed.

Recipe 01 — Remove a job entirely

```
Unregister-SqlCertRenewalJob -JobId sql01-engine
```

Removes the schedule entry and the spec; the run log is kept.

Recipe 02 — Drop the schedule but keep the spec (to re-schedule later)

```
Unregister-SqlCertRenewalJob -JobId reports-pbirs -KeepSpec
```

Recipe 03 — A SQL Agent-scheduled job

```
Unregister-SqlCertRenewalJob -JobId ssrs-portal -SqlInstance RPT01
```

For a SqlAgent job, name the instance hosting the Agent job so the Agent job is removed too.

Learner

This undoes a renewal job. It reads the job to find out which scheduler it lives in, removes the scheduled entry there, and deletes the saved job note — unless you say `-KeepSpec`, which removes only the schedule and leaves the note so you can re-schedule it later. The history of past runs is left in place.

Software Approval

Question	Answer
Outbound calls?	None to the vendor. It removes a local schedule entry (and a SQL Agent job, if that backend).
What it changes	Removes the schedule entry; deletes the spec unless `-KeepSpec`. Leaves the run log.
Privileges	Local admin (or Agent rights) to remove the schedule entry.

Dependencies

Dependency	Why	Required?
The job's spec	To learn which backend to clean up.	Yes
The scheduler's removal tooling	To delete the entry (returns a note if absent).	Yes

Compliance

- Removes the automation cleanly, so a decommissioned certificate isn't quietly re-issued forever.
- **Keeps the run log** as history — the record of what the job did survives the job's removal, for audit.
- `-KeepSpec` separates "stop scheduling" from "forget entirely", so a pause is distinguishable from a deletion.