Enable winrm on windows server

Fix: Enable WinRM on the Windows Server

You need to run these commands **on the Windows Server itself** (`192.168.10.5`), in **PowerShell as Administrator**:

Quick path — enable HTTP listener on 5985

powershell

```
# One-shot: starts the WinRM service, sets it to auto-start,
# creates the HTTP listener on 5985, and adds the firewall rule.
Enable-PSRemoting -Force
```

If the server is on a "Public" network profile (common on standalone test VMs), that command will refuse. Use this instead:

powershell

```
Enable-PSRemoting -Force -SkipNetworkProfileCheck
```

Verify it's listening

After running the command, confirm:

powershell

```
# Should show a Transport = HTTP listener on port 5985
winrm enumerate winrm/config/Listener

# Confirm the service is running
Get-Service WinRM

# Confirm the port is bound
netstat -ano | findstr :5985
```

The `netstat` line should show something like:

```
TCP    0.0.0.0:5985    0.0.0.0:0    LISTENING    1234
```

Verify the firewall rule

powershell

```
Get-NetFirewallRule -DisplayName "*WinRM*" |
    Select-Object DisplayName, Enabled, Profile, Direction, Action |
    Format-Table -AutoSize
```

You want `Enabled = True` for the "Windows Remote Management (HTTP-In)" rule, and the `Profile` should cover the network you're connecting from (Domain, Private, or Public).

Then test from your Linux box

Once the server is configured, just retry:

bash

```
python winrm_ad_enum_http.py --host 192.168.10.5 --user 'DOMAIN\admin'
```

Why "refused" specifically (vs. "timed out")

The difference matters for diagnosis:

- **"Connection refused"** (what you got) → The TCP stack on the server responded with a RST packet. The host is reachable on the network, but nothing is listening on that port. Almost always means the service isn't running or isn't bound.
- **"Connection timed out"** → No response at all. Usually means a firewall is silently dropping packets, or the host is unreachable.

So good news: your network path to `192.168.10.5` is fine. You only need to turn WinRM on.

Optional: HTTPS at the same time

While you're on the server, you may as well enable HTTPS (5986) too so the `winrm_ad_enum.py` script also works. Here's the minimal sequence — self-signed cert is fine for a lab:

powershell

```
# Get the server's FQDN (or just use the IP / hostname clients will use)
$fqdn = [System.Net.Dns]::GetHostByName($env:COMPUTERNAME).HostName

# Self-signed cert
$cert = New-SelfSignedCertificate -DnsName $fqdn `
    -CertStoreLocation Cert:\LocalMachine\My -NotAfter (Get-Date).AddYears(5)
```

```
# HTTPS listener
New-Item -Path WSMan:\LocalHost\Listener -Transport HTTPS -Address * `
    -CertificateThumbPrint $cert.Thumbprint -HostName $fqdn -Force

# Firewall rule for 5986
New-NetFirewallRule -DisplayName "WinRM HTTPS-In (TCP 5986)" `
    -Direction Inbound -LocalPort 5986 -Protocol TCP -Action Allow
```

Run the verification again — `winrm enumerate winrm/config/Listener` should now show **both** an HTTP and an HTTPS listener.

Once that's done, give me a shout if the next run still has issues — but my bet is this fixes it cleanly.

You said: why the winrm only works with AllowUnencrypted to True for some winrm calls

why the winrm only works with AllowUnencrypted to True for some winrm calls

4:53 PM

Claude responded: Good observation — this catches a lot of people out.

Good observation — this catches a lot of people out. The short version: **WinRM doesn't decide encryption per-call based on what you're doing, it decides based on which auth transport you negotiated**. Some transports provide their own encryption layer, others don't. `AllowUnencrypted` is what the server checks when a transport can't encrypt.

The actual rule

When a WinRM request arrives over HTTP (port 5985), the server asks one question: **"Is this SOAP body wrapped in an encryption layer?"**

- **Yes** → process the request, regardless of `AllowUnencrypted`.
- **No** → check `AllowUnencrypted`. If `false`, reject with HTTP 411 / "The WinRM client cannot process the request. Unencrypted traffic is currently disabled". If `true`, process it.

The transport you authenticate with determines the answer to that question.

Transport-by-transport breakdown

Transport	Provides encryption?	Needs `AllowUnencrypted=true`?
NTLM	Yes — NTLMv2 with session security (sealing)	No
Kerberos	Yes — GSS-API confidentiality	No
CredSSP	Yes — wraps in TLS even over HTTP	No
Negotiate	Yes (negotiates to NTLM or Kerberos)	No
Basic	**No** — credentials base64, body cleartext	**Yes**
Certificate	No on its own over HTTP	**Yes**

So if some of your calls work and others don't, the most likely cause is that the failing calls are using a transport that can't encrypt — Basic or certificate auth — while the working ones are using NTLM/Kerberos.

Why this design exists

WinRM is essentially SOAP-over-HTTP. Microsoft's security model says: **the payload must be confidential, but it doesn't care how you achieve that**. There are three valid ways:

1. **Transport-level encryption** — use HTTPS (port 5986). The TLS tunnel encrypts everything, including the credentials. Any auth method is fine.
2. **Message-level encryption** — use HTTP (port 5985), but with NTLM/Kerberos/CredSSP, which encrypt the SOAP body inside the HTTP request. The HTTP envelope is plain but the contents aren't.
3. **No encryption** — only allowed if the admin explicitly opts in via `AllowUnencrypted=true`. This is for Basic auth, certificate auth, or scenarios where the admin has decided the network itself is trusted (IPsec, isolated VLAN, loopback only, etc.).

The setting exists as a **safety net** so that misconfiguring the client doesn't silently leak credentials in cleartext. If you forget to enable NTLM and accidentally fall back to Basic, the server refuses

rather than sending your password in plaintext.

Common scenarios where you'd see "some work, some don't"

Scenario A: Mixed clients

- Your Python script using `pywinrm` with default `transport=ntlm` → works.
- A `curl` test or some legacy script using Basic auth → fails until `AllowUnencrypted=true`.
- Diagnosis: check which transport each tool uses.

Scenario B: Same client, different auth

- `python winrm_ad_enum_http.py --transport ntlm` → works.
- `python winrm_ad_enum_http.py --transport basic` → fails with HTTP 411.
- Diagnosis: Basic has no encryption, so it falls afoul of the default `AllowUnencrypted=false`.

Scenario C: Kerberos vs NTLM

- NTLM works fine.
- Kerberos works *if* SPNs are correct and the client has a ticket, but the failure mode (no encryption layer because the auth handshake itself failed) can look like an encryption problem rather than an auth problem.

Scenario D: PowerShell client without `-Authentication`

- `Invoke-Command -ComputerName ...` defaults to Kerberos → encrypted → works.
- `Invoke-Command -Authentication Basic -Credential ...` → not encrypted → blocked unless `AllowUnencrypted=true`.

How to check what's actually happening

On the server, see what the current policy is:

powershell

```
Get-Item WSMan:\localhost\Service\AllowUnencrypted
Get-ChildItem WSMan:\localhost\Service\Auth
```

The `Auth` listing shows which auth methods are enabled — Basic, Kerberos, Negotiate, Certificate, CredSSP. If Basic is enabled but `AllowUnencrypted=false`, you can authenticate with Basic only over HTTPS.

On the wire, sniff a working vs failing request with Wireshark on port 5985:

- A working NTLM request: you'll see the SOAP envelope as binary garbage after the `Authorization: Negotiate ...` header — that's the encrypted payload.
- A failing Basic request: you'll see the SOAP envelope as readable XML, which is exactly why the server refuses it when `AllowUnencrypted=false`.

The right fix

Don't reach for `AllowUnencrypted=true` just to make things work. Instead, figure out *why* the failing call isn't using an encrypting transport, and fix that:

- If the client is using Basic, switch to NTLM (`--transport ntlm` in the script).
- If you can't switch (some scripts, some appliances genuinely require Basic) → enable the HTTPS listener on 5986 and point those clients at HTTPS instead. They can keep using Basic; TLS handles the encryption.
- `AllowUnencrypted=true` should be a last resort, for closed labs or networks you fully trust.

TL;DR

`AllowUnencrypted` isn't really about what command you're running — it's about whether the **auth transport** carrying that command provides encryption. NTLM/Kerberos/CredSSP do, so they bypass the check. Basic and certificate auth don't, so they trip it. "Some calls work, others don't" is almost always "some calls use NTLM, others use Basic."

Revision #1
Created 19 May 2026 16:05:55 by Admin
Updated 19 May 2026 16:06:23 by Admin