

Winrm

- [Test WinRM Connectivity on both HTTP \(5985\) and HTTPS \(5986\)](#)
- [How to enable winrm in windows](#)
- [Winrm Active Directory Enumeration](#)
- [Winrm Active Directory Enumeration using HTTP \(5985\)](#)
- [Winrm Active Directory Diagnostics using HTTPS](#)
- [Enable winrm on windows server](#)

Test WinRM Connectivity on both HTTP (5985) and HTTPS (5986)

Script to test:

winrm_ad-test.py

```
#!/usr/bin/env python3
"""
WinRM Connectivity Tester and Active Directory Query Tool
=====

This script tests connectivity to a Windows Server via WinRM on both HTTP (5985)
and HTTPS (5986) ports, and runs Active Directory queries via PowerShell.

Requirements:
    pip install pywinrm[kerberos]
    # or just:
    pip install pywinrm

Optional (for HTTPS with self-signed certs, no extra deps needed - handled below).

Usage:
    python winrm_ad_test.py --host server.domain.local --user DOMAIN\admin --password
'Passw0rd!'
    python winrm_ad_test.py --host 10.0.0.5 --user admin@domain.local --password 'Passw0rd!'
--domain domain.local
"""

import argparse
import socket
import ssl
import sys
import getpass
```

```

from contextlib import closing

try:
    import winrm
    from winrm.exceptions import (
        WinRMTransportError,
        WinRMOperationTimeoutError,
        InvalidCredentialsError,
    )
except ImportError:
    print("[!] The 'pywinrm' package is required. Install it with:")
    print("    pip install pywinrm")
    sys.exit(1)

# ----- ANSI colors for nicer output -----
class C:
    OK = "\033[92m"
    WARN = "\033[93m"
    FAIL = "\033[91m"
    INFO = "\033[94m"
    BOLD = "\033[1m"
    END = "\033[0m"

def banner(text: str) -> None:
    line = "=" * 70
    print(f"\n{C.BOLD}{line}\n{text}\n{line}{C.END}")

# ----- Step 1: Raw TCP port check -----
def check_tcp_port(host: str, port: int, timeout: float = 5.0) -> bool:
    """Confirm the TCP port is reachable before attempting WinRM handshake."""
    print(f"{C.INFO}[*] Testing TCP connectivity to {host}:{port} ...{C.END}")
    try:
        with closing(socket.create_connection((host, port), timeout=timeout)):
            print(f"{C.OK}[+] TCP port {port} is OPEN on {host}{C.END}")
            return True
    except socket.timeout:
        print(f"{C.FAIL}[-] TCP port {port} timed out (firewall? service down?){C.END}")

```

```

except ConnectionRefusedError:
    print(f"{C.FAIL}[-] TCP port {port} refused the connection (service not
listening){C.END}")
except socket.gaierror:
    print(f"{C.FAIL}[-] Could not resolve hostname '{host}' (DNS issue){C.END}")
except OSError as e:
    print(f"{C.FAIL}[-] TCP error on port {port}: {e}{C.END}")
return False

# ----- Step 2: TLS certificate check on 5986 -----
def check_tls_cert(host: str, port: int = 5986, timeout: float = 5.0) -> None:
    """Optional: peek at the server's TLS cert presented on the HTTPS WinRM listener."""
    print(f"{C.INFO}[*] Inspecting TLS certificate on {host}:{port} ...{C.END}")
    ctx = ssl.create_default_context()
    # We don't want to fail on a self-signed cert here, we just want to see it.
    ctx.check_hostname = False
    ctx.verify_mode = ssl.CERT_NONE
    try:
        with socket.create_connection((host, port), timeout=timeout) as sock:
            with ctx.wrap_socket(sock, server_hostname=host) as ssock:
                cert = ssock.getpeercert(binary_form=False)
                # When verify_mode=CERT_NONE, getpeercert() can return {} - use binary form
                fallback
            if not cert:
                der = ssock.getpeercert(binary_form=True)
                print(f"{C.WARN}[!] TLS handshake OK. Server presented a certificate "
                    f"({len(der)} bytes DER), details hidden (likely self-
signed){C.END}")
            else:
                subject = dict(x[0] for x in cert.get("subject", []))
                issuer = dict(x[0] for x in cert.get("issuer", []))
                print(f"{C.OK}[+] TLS handshake OK.{C.END}")
                print(f"    Subject : {subject}")
                print(f"    Issuer  : {issuer}")
                print(f"    Valid until: {cert.get('notAfter')}")
    except Exception as e:
        print(f"{C.WARN}[!] Could not retrieve TLS info: {e}{C.END}")

```

```

# ----- Step 3: WinRM session -----
def build_session(host: str, port: int, scheme: str, user: str, password: str,
                 transport: str, verify_ssl: bool) -> winrm.Session:
    """
    Build a pywinrm Session.

    transport can be:
        - 'ntlm'      (most common for AD-joined Windows servers)
        - 'kerberos'  (requires kerberos client libs + valid ticket / krb5.conf)
        - 'basic'     (only if AllowUnencrypted=true on server; not recommended)
        - 'ssl'
        - 'credssp'   (for double-hop scenarios)
    """
    endpoint = f"{scheme}://{host}:{port}/wsman"
    print(f"{C.INFO}[*] Opening WinRM session: {endpoint} (transport={transport}){C.END}")

    session = winrm.Session(
        endpoint,
        auth=(user, password),
        transport=transport,
        server_cert_validation="validate" if verify_ssl else "ignore",
        # operation_timeout_sec=20,
        # read_timeout_sec=30,
    )
    return session

def run_winrm_test(host: str, port: int, scheme: str, user: str, password: str,
                  transport: str, verify_ssl: bool) -> winrm.Session | None:
    """Run a trivial 'whoami' / hostname command to confirm WinRM works end-to-end."""
    try:
        session = build_session(host, port, scheme, user, password, transport, verify_ssl)

        # A harmless probe command
        result = session.run_cmd("hostname")
        if result.status_code == 0:
            print(f"{C.OK}[+] WinRM authenticated successfully on
{scheme.upper()}:{port}{C.END}")
            print(f"    Remote hostname: {result.std_out.decode(errors='replace').strip()}")
            return session
    
```

```

else:
    print(f"{C.FAIL}[-] WinRM connected but 'hostname' returned exit code "
          f"{result.status_code}{C.END}")
    print(f"    stderr: {result.std_err.decode(errors='replace').strip()}")
except InvalidCredentialsError:
    print(f"{C.FAIL}[-] Invalid credentials for {user} on {scheme.upper()}:{port}{C.END}")
except WinRMTransportError as e:
    print(f"{C.FAIL}[-] WinRM transport error on {scheme.upper()}:{port}: {e}{C.END}")
except WinRMOperationTimeoutError as e:
    print(f"{C.FAIL}[-] WinRM operation timed out: {e}{C.END}")
except Exception as e:
    print(f"{C.FAIL}[-] Unexpected WinRM error on {scheme.upper()}:{port}: "
          f"{type(e).__name__}: {e}{C.END}")
return None

# ----- Step 4: Active Directory queries via PowerShell -----
AD_PROBE_PS = r"""
$ErrorActionPreference = 'Stop'
try {
    Import-Module ActiveDirectory -ErrorAction Stop
    Write-Output "MODULE_OK"
} catch {
    Write-Output "MODULE_MISSING: $($_.Exception.Message)"
    exit 2
}
"""

AD_QUERIES = {
    "Domain info": r"""
        Import-Module ActiveDirectory
        Get-ADDomain | Select-Object Forest, DNSRoot, NetBIOSName, DomainMode,
            PDCEmulator, RIDMaster, InfrastructureMaster, DistinguishedName |
            Format-List
    """,
    "Forest info & FSMO roles": r"""
        Import-Module ActiveDirectory
        Get-ADForest | Select-Object Name, ForestMode, RootDomain, SchemaMaster,
            DomainNamingMaster, GlobalCatalogs, Sites |
            Format-List
    """
}

```

```

"",
"Domain Controllers": r""
    Import-Module ActiveDirectory
    Get-ADDomainController -Filter * |
        Select-Object Name, HostName, IPv4Address, Site, OperatingSystem,
            IsGlobalCatalog, IsReadOnly |
        Format-Table -AutoSize
"",
"Domain Trusts": r""
    Import-Module ActiveDirectory
    $trusts = Get-ADTrust -Filter * -ErrorAction SilentlyContinue
    if ($trusts) {
        $trusts | Select-Object Name, Source, Target, Direction, TrustType |
            Format-Table -AutoSize
    } else {
        Write-Output "(No trusts configured)"
    }
"",

# ----- Generic AD object enumeration -----
"AD object count by class": r""
    Import-Module ActiveDirectory
    # Group every object in the directory by its objectClass to get a
    # high-level inventory of what's in AD.
    Get-ADObject -Filter * -ResultPageSize 1000 |
        Group-Object objectClass |
        Sort-Object Count -Descending |
        Select-Object @{N='ObjectClass';E={$_.Name}}, Count |
        Format-Table -AutoSize
"",
"Organizational Units (tree)": r""
    Import-Module ActiveDirectory
    Get-ADOrganizationalUnit -Filter * -Properties WhenCreated |
        Sort-Object DistinguishedName |
        Select-Object Name, DistinguishedName, WhenCreated |
        Format-Table -AutoSize -Wrap
"",
"Containers (built-in)": r""
    Import-Module ActiveDirectory
    # The non-OU top-level containers: Users, Computers, Builtin, etc.

```

```

    Get-ADObject -Filter 'ObjectClass -eq "container"' -SearchScope OneLevel |
        Select-Object Name, DistinguishedName |
        Format-Table -AutoSize -Wrap
"",
"Generic object search (first 20)": r""
    Import-Module ActiveDirectory
    # Raw Get-ADObject query – shows you exactly what's stored in AD,
    # not filtered by user/computer/group cmdlets.
    Get-ADObject -Filter * -ResultSetSize 20 -Properties whenCreated, whenChanged |
        Select-Object Name, ObjectClass, whenCreated, whenChanged, DistinguishedName |
        Format-Table -AutoSize -Wrap
"",

# ----- Users -----
"User summary (enabled vs disabled)": r""
    Import-Module ActiveDirectory
    $all      = Get-ADUser -Filter *
    $enabled  = ($all | Where-Object Enabled -eq $true).Count
    $disabled = ($all | Where-Object Enabled -eq $false).Count
    [PSCustomObject]@{
        TotalUsers = $all.Count
        Enabled    = $enabled
        Disabled   = $disabled
    } | Format-List
"",
"First 10 users": r""
    Import-Module ActiveDirectory
    Get-ADUser -Filter * -ResultSetSize 10 -Properties LastLogonDate, Enabled,
PasswordLastSet |
        Select-Object SamAccountName, Name, Enabled, LastLogonDate, PasswordLastSet |
        Format-Table -AutoSize
"",
"Stale users (no logon in 90 days)": r""
    Import-Module ActiveDirectory
    $cutoff = (Get-Date).AddDays(-90)
    Get-ADUser -Filter {LastLogonDate -lt $cutoff -and Enabled -eq $true} `
        -Properties LastLogonDate -ResultSetSize 10 |
        Select-Object SamAccountName, Name, LastLogonDate |
        Format-Table -AutoSize
"",

```

```

# ----- Computers -----
"Computer summary by OS": r""
    Import-Module ActiveDirectory
    Get-ADComputer -Filter * -Properties OperatingSystem |
        Group-Object OperatingSystem |
        Sort-Object Count -Descending |
        Select-Object @{N='OperatingSystem';E={if($_.Name){$_Name}else{'(unset)'}}}
Count |
    Format-Table -AutoSize
"",
"First 10 computers": r""
    Import-Module ActiveDirectory
    Get-ADComputer -Filter * -ResultSetSize 10 `
        -Properties OperatingSystem, LastLogonDate, IPv4Address |
        Select-Object Name, OperatingSystem, IPv4Address, LastLogonDate, Enabled |
        Format-Table -AutoSize
"",

# ----- Groups -----
"Group summary by category/scope": r""
    Import-Module ActiveDirectory
    Get-ADGroup -Filter * -Properties GroupCategory, GroupScope |
        Group-Object GroupCategory, GroupScope |
        Sort-Object Count -Descending |
        Select-Object @{N='Category/Scope';E={$_Name}}, Count |
        Format-Table -AutoSize
"",
"Privileged groups (Domain Admins)": r""
    Import-Module ActiveDirectory
    Get-ADGroupMember -Identity 'Domain Admins' -ErrorAction SilentlyContinue |
        Select-Object Name, SamAccountName, objectClass, distinguishedName |
        Format-Table -AutoSize -Wrap
"",
"Privileged groups (Enterprise Admins)": r""
    Import-Module ActiveDirectory
    Get-ADGroupMember -Identity 'Enterprise Admins' -ErrorAction SilentlyContinue |
        Select-Object Name, SamAccountName, objectClass |
        Format-Table -AutoSize
"",

```

```

# ----- Group Policy & misc -----
"Group Policy Objects": r"""
    if (Get-Module -ListAvailable -Name GroupPolicy) {
        Import-Module GroupPolicy
        Get-GPO -All |
            Select-Object DisplayName, GpoStatus, CreationTime, ModificationTime |
            Format-Table -AutoSize
    } else {
        Write-Output "(GroupPolicy module not installed on this host)"
    }
""",
"Service Accounts (gMSA)": r"""
    Import-Module ActiveDirectory
    $gmsa = Get-ADServiceAccount -Filter * -ErrorAction SilentlyContinue
    if ($gmsa) {
        $gmsa | Select-Object Name, SamAccountName, Enabled, ObjectClass |
            Format-Table -AutoSize
    } else {
        Write-Output "(No gMSA / managed service accounts found)"
    }
""",
}

```

```

def run_ps(session: winrm.Session, script: str) -> tuple[int, str, str]:
    """Run a PowerShell script and return (exit_code, stdout, stderr)."""
    r = session.run_ps(script)
    return (
        r.status_code,
        r.std_out.decode("utf-8", errors="replace"),
        r.std_err.decode("utf-8", errors="replace"),
    )

```

```

def query_active_directory(session: winrm.Session, ad_filter: str | None = None) -> None:
    banner("Active Directory queries")

    # Confirm the AD module is present (only on a DC or a box with RSAT installed)
    code, out, err = run_ps(session, AD_PROBE_PS)

```

```

if "MODULE_OK" not in out:
    print(f"{C.WARN}[!] ActiveDirectory PowerShell module is not available on this
host.{C.END}")
    print(f"    stdout: {out.strip()}")
    print(f"    stderr: {err.strip()}")
    print(f"    Tip: run this against a Domain Controller, or install RSAT-AD-
PowerShell.")
    return

print(f"{C.OK}[+] ActiveDirectory module is loaded on the remote host.{C.END}")

# Optionally narrow the query list to labels matching a substring
items = AD_QUERIES.items()
if ad_filter:
    needle = ad_filter.lower()
    items = [(k, v) for k, v in AD_QUERIES.items() if needle in k.lower()]
    if not items:
        print(f"{C.WARN}[!] No AD query labels match '{ad_filter}'. "
            f"Available labels:{C.END}")
        for k in AD_QUERIES:
            print(f"    - {k}")
        return
    print(f"{C.INFO}[*] Running {len(items)} queries matching '{ad_filter}'.{C.END}")

for label, script in items:
    print(f"\n{C.BOLD}--- {label} ---{C.END}")
    code, out, err = run_ps(session, script)
    if code == 0:
        print(out.rstrip() or "(no output)")
    else:
        print(f"{C.FAIL}[-] Query failed (exit {code}){C.END}")
        if err.strip():
            print(f"    stderr: {err.strip()}")

# ----- Orchestration -----
def test_endpoint(host: str, port: int, scheme: str, user: str, password: str,
    transport: str, verify_ssl: bool, run_ad: bool,
    ad_filter: str | None = None) -> bool:
    banner(f"Testing {scheme.upper()} on port {port}")

```

```

if not check_tcp_port(host, port):
    return False

if scheme == "https":
    check_tls_cert(host, port)

session = run_winrm_test(host, port, scheme, user, password, transport, verify_ssl)
if not session:
    return False

if run_ad:
    query_active_directory(session, ad_filter=ad_filter)

return True

def parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser(
        description="Test WinRM connectivity (HTTP 5985 / HTTPS 5986) and query Active
Directory.",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=__doc__,
    )
    p.add_argument("--host", help="Target Windows Server hostname or IP")
    p.add_argument("--user", help=r"Username (DOMAIN\user, user@domain, or local user)")
    p.add_argument("--password", help="Password (prompted if omitted)")
    p.add_argument(
        "--transport",
        default="ntlm",
        choices=["ntlm", "kerberos", "basic", "ssl", "credssp"],
        help="WinRM auth transport (default: ntlm)",
    )
    p.add_argument("--http-port", type=int, default=5985, help="HTTP WinRM port (default
5985)")
    p.add_argument("--https-port", type=int, default=5986, help="HTTPS WinRM port (default
5986)")
    p.add_argument("--only", choices=["http", "https"], help="Test only one scheme")
    p.add_argument("--verify-ssl", action="store_true",
        help="Verify the server's TLS certificate (default: ignore self-signed)")

```

```
p.add_argument("--no-ad", action="store_true", help="Skip Active Directory queries")
p.add_argument("--ad-filter", metavar="SUBSTRING",
               help="Run only AD queries whose label contains this substring "
                  "(case-insensitive). E.g. --ad-filter object")
p.add_argument("--list-ad-queries", action="store_true",
               help="List the available AD query labels and exit")
return p.parse_args()
```

```
def main() -> int:
    args = parse_args()

    if args.list_ad_queries:
        print("Available AD query labels:")
        for k in AD_QUERIES:
            print(f" - {k}")
        return 0

    if not args.host or not args.user:
        print(f"{C.FAIL}[-] --host and --user are required (use --list-ad-queries "
              f"to view queries without connecting).{C.END}")
        return 2

    password = args.password or getpass.getpass(f"Password for {args.user}: ")

    results = {}

    if args.only != "https":
        results["http"] = test_endpoint(
            host=args.host,
            port=args.http_port,
            scheme="http",
            user=args.user,
            password=password,
            transport=args.transport,
            verify_ssl=False,          # irrelevant for http
            run_ad=not args.no_ad,
            ad_filter=args.ad_filter,
        )
```

```

if args.only != "http":
    results["https"] = test_endpoint(
        host=args.host,
        port=args.https_port,
        scheme="https",
        user=args.user,
        password=password,
        transport="ssl" if args.transport == "basic" else args.transport,
        verify_ssl=args.verify_ssl,
        run_ad=not args.no_ad and not results.get("http", False), # avoid running AD
twice
        ad_filter=args.ad_filter,
    )

banner("Summary")
for scheme, ok in results.items():
    status = f"{C.OK}OK{C.END}" if ok else f"{C.FAIL}FAIL{C.END}"
    port = args.https_port if scheme == "https" else args.http_port
    print(f" {scheme.upper():5s} (port {port}): {status}")

return 0 if any(results.values()) else 1

if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n[!] Interrupted by user.")
        sys.exit(130)

```

What it tests, in order:

1. **Raw TCP check** on 5985 / 5986 — distinguishes "firewall blocking" from "service down" from "DNS broken" before any auth happens.
2. **TLS certificate inspection** on 5986 — pulls the cert (even if self-signed) so you can see subject/issuer/expiry without failing the handshake.
3. **WinRM authentication** — opens a `pywinrm` session and runs `hostname` as a sanity command.
4. **AD module probe** — checks if the `ActiveDirectory` PowerShell module is loadable on the remote host (only true on a DC or a box with RSAT installed).
5. **AD queries** — domain info, domain controllers, first 10 users, Domain Admins members, computer count.

Install the dependency:

bash

```
pip install pywinrm
```

Run it:

bash

```
# Test both 5985 and 5986
python winrm_ad_test.py --host dc01.corp.local --user 'CORP\admin'

# Only HTTPS, skip AD queries
python winrm_ad_test.py --host 10.0.0.5 --user admin@corp.local --only https --no-ad

# Use Kerberos instead of NTLM
python winrm_ad_test.py --host dc01.corp.local --user admin@CORP.LOCAL --transport kerberos
```

A few practical notes:

- WinRM over HTTP (5985) sends credentials in plaintext unless wrapped in NTLM/Kerberos message-level encryption. The script defaults to `ntlm`, which encrypts the payload even on port 5985 — so it works against the default Windows config.
- HTTPS (5986) requires a listener with a cert. On a fresh Windows Server you usually need to run `winrm quickconfig -transport:https` or create the listener manually. The script tolerates self-signed certs by default; pass `--verify-ssl` if you have a real PKI.
- The AD queries only succeed against a host where `Import-Module ActiveDirectory` works — that's normally a Domain Controller, or a member server with RSAT-AD-PowerShell installed. Against any other server the script gracefully reports the module is missing instead of failing.
- For double-hop scenarios (the remote box needs to authenticate onward to a different DC), switch `--transport credssp` and configure CredSSP on the server.

How to enable winrm in windows

Part 1 — Enable WinRM on HTTP (port 5985)

This is the easy part. Run **PowerShell as Administrator** on the Windows Server:

powershell

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

That's it for HTTP. Verify:

powershell

```
# Should show a Transport=HTTP listener on port 5985  
winrm enumerate winrm/config/Listener  
  
# Or with PowerShell cmdlets  
Get-WSManInstance -ResourceURI winrm/config/listener -Enumerate
```

If the host is on a "Public" network profile, `Enable-PSRemoting` will refuse. Either change the profile to Private/Domain, or force it:

powershell

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

Part 2 — Enable WinRM on HTTPS (port 5986)

HTTPS requires a certificate. You have three options depending on your environment.

Option A — Self-signed certificate (lab / quick test)

powershell

```
# 1) Create a self-signed cert. Use the server's real FQDN.  
$fqdn = [System.Net.Dns]::GetHostByName($env:COMPUTERNAME).HostName  
$cert = New-SelfSignedCertificate `  
    -DnsName $fqdn `  
    -CertStoreLocation Cert:\LocalMachine\My `
```

```

-KeyExportPolicy Exportable `
-KeySpec Signature `
-KeyLength 2048 `
-KeyAlgorithm RSA `
-HashAlgorithm SHA256 `
-NotAfter (Get-Date).AddYears(5)

Write-Host "Thumbprint: $($cert.Thumbprint)"
Write-Host "FQDN:      $fqdn"

# 2) Create the HTTPS listener bound to that cert
New-Item -Path WSMAN:\LocalHost\Listener `
  -Transport HTTPS `
  -Address * `
  -CertificateThumbPrint $cert.Thumbprint `
  -HostName $fqdn `
  -Force

# 3) Open the firewall for 5986
New-NetFirewallRule -DisplayName "WinRM HTTPS-In (TCP 5986)" `
  -Direction Inbound `
  -LocalPort 5986 `
  -Protocol TCP `
  -Action Allow `
  -Profile Domain,Private

```

Option B — Certificate from your enterprise CA (production)

If you have an Active Directory Certificate Services CA, request a Server Authentication cert first:

powershell

```

# Request from AD CS using a template that allows Server Authentication
# (the template name varies – common ones are "WebServer" or "Computer")
$req = Get-Certificate `
  -Template "WebServer" `
  -DnsName $fqdn `
  -SubjectName "CN=$fqdn" `
  -CertStoreLocation Cert:\LocalMachine\My

$cert = $req.Certificate

```

```
# Then create the listener the same way as Option A, step 2+3
New-Item -Path WSMAN:\LocalHost\Listener `
  -Transport HTTPS `
  -Address * `
  -CertificateThumbPrint $cert.Thumbprint `
  -HostName $fqdn `
  -Force
```

Option C — Existing certificate (you already have one)

powershell

```
# Find an existing Server Auth cert for this host
$cert = Get-ChildItem Cert:\LocalMachine\My |
  Where-Object {
    $_.Subject -match $fqdn -and
    $_.EnhancedKeyUsageList.FriendlyName -contains "Server Authentication"
  } | Select-Object -First 1

# Use its thumbprint in step 2 of Option A
```

Part 3 — Verify both listeners

powershell

```
# List all listeners; you should see two – HTTP on 5985 and HTTPS on 5986
winrm enumerate winrm/config/Listener
```

Expected output (abbreviated):

```
Listener
  Address = *
  Transport = HTTP
  Port = 5985
  Enabled = true

Listener
  Address = *
  Transport = HTTPS
  Port = 5986
  Enabled = true
```

```
CertificateThumbprint = A1B2C3D4...
```

Test from the **same server** first (loopback check):

powershell

```
# HTTP
Test-WSMan -ComputerName localhost

# HTTPS – use -UseSSL and skip CA check if self-signed
$opt = New-PSSessionOption -SkipCACheck -SkipCNCheck -SkipRevocationCheck
Test-WSMan -ComputerName $fqdn -UseSSL -SessionOption $opt
```

Then test **from another machine** on the network:

powershell

```
# From a different Windows box
Test-WSMan -ComputerName dc01.corp.local
Test-WSMan -ComputerName dc01.corp.local -UseSSL -SessionOption $opt
```

Or from your Linux box with the Python script you already have:

bash

```
python winrm_ad_test.py --host dc01.corp.local --user 'CORP\admin'
```

Part 4 — Hardening (recommended for production)

After both listeners work, lock things down:

powershell

```
# Disable unencrypted traffic (forces NTLM/Kerberos encryption or HTTPS)
Set-Item -Path WSMan:\LocalHost\Service\AllowUnencrypted -Value $false

# Disable Basic auth (use NTLM/Kerberos/CredSSP instead)
Set-Item -Path WSMan:\LocalHost\Service\Auth\Basic -Value $false

# If you want to require HTTPS only, remove the HTTP listener
Get-ChildItem WSMan:\LocalHost\Listener |
    Where-Object Keys -match "Transport=HTTP$" |
    Remove-Item -Recurse
```

```
# Optional: restrict who can connect (default is BUILTIN\Administrators)
Set-PSSessionConfiguration -Name Microsoft.PowerShell `
    -ShowSecurityDescriptorUI
```

For NTLM connections from non-domain Linux clients, you may also need to add the server to TrustedHosts on the **client**, not the server:

powershell

```
# On the Windows client (not needed for Linux pywinrm clients)
Set-Item WSMAN:\LocalHost\Client\TrustedHosts -Value "dc01.corp.local" -Force
```

Part 5 — Rolling it out at scale via GPO

For multiple servers, do it once via Group Policy instead of per-host:

Computer Configuration → Policies → Administrative Templates → Windows Components → Windows Remote Management (WinRM) → WinRM Service

- **Allow remote server management through WinRM** → Enabled, IPv4/IPv6 filter = *
- **Allow Basic authentication** → Disabled
- **Allow unencrypted traffic** → Disabled

And in Computer Configuration → Policies → Windows Settings → Security Settings → System Services, set **Windows Remote Management (WS-Management)** to Automatic.

Firewall rule via GPO: Computer Configuration → Policies → Windows Settings → Security Settings → Windows Defender Firewall with Advanced Security → Inbound Rules → New Rule → Predefined → "Windows Remote Management".

The HTTPS cert is the only piece GPO can't easily push — that needs either AD CS auto-enrollment (recommended) or a per-host script.

Common pitfalls

The most frequent issues, in order of how often I've seen them bite people: the firewall rule existing but being scoped to "Private only" while the NIC is on a "Public" profile; the cert's CN/SAN not matching the hostname clients actually use (cert says `dc01` but clients connect to `dc01.corp.local`); the HTTPS listener pointing at a thumbprint whose cert has expired or been removed; and `AllowUnencrypted=false` combined with Basic auth, which leaves no working auth path. If `Test-WSMan` fails, check those four before anything else.

Want me to put any of this into a single bootstrap PowerShell script you can drop onto a new server?

Winrm Active Directory Enumeration

winrm_ad_enum.py

```
#!/usr/bin/env python3
"""
WinRM Active Directory Object Enumerator (HTTPS-only)
=====

Connects to a Windows Domain Controller over WinRM/HTTPS (port 5986) and
retrieves ALL user and computer objects from Active Directory.

Output formats:
  - Pretty-printed table on stdout (default)
  - CSV files (one per object type)
  - JSON files (one per object type)

Requirements:
  pip install pywinrm

Usage:
  # Pull everything, print to screen
  python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin'

  # Pull everything and save to CSV in ./ad_export/
  python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
    --output-dir ./ad_export --format csv

  # Save to JSON, only users
  python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
    --object-type users --format json --output-dir ./ad_export

  # Strict TLS (your DC has a cert signed by a CA your client trusts)
  python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' --verify-ssl
```

```

# Limit search to a specific OU
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
    --search-base "OU=Employees,DC=corp,DC=local"
"""

from __future__ import annotations

import argparse
import csv
import getpass
import json
import socket
import ssl
import sys
from contextlib import closing
from datetime import datetime
from pathlib import Path

try:
    import winrm
    from winrm.exceptions import (
        InvalidCredentialsError,
        WinRMOperationTimeoutError,
        WinRMTransportError,
    )
except ImportError:
    print("[!] The 'pywinrm' package is required. Install it with:")
    print("    pip install pywinrm")
    sys.exit(1)

# -----
# Output helpers
# -----

class C:
    OK = "\033[92m"
    WARN = "\033[93m"
    FAIL = "\033[91m"
    INFO = "\033[94m"

```

```
BOLD = "\033[1m"
```

```
END = "\033[0m"
```

```
def banner(text: str) -> None:
```

```
    line = "=" * 72
```

```
    print(f"\n{C.BOLD}{line}\n{text}\n{line}{C.END}")
```

```
def info(msg: str) -> None:
```

```
    print(f"{C.INFO}[*]{C.END} {msg}")
```

```
def ok(msg: str) -> None:
```

```
    print(f"{C.OK}[+]{C.END} {msg}")
```

```
def warn(msg: str) -> None:
```

```
    print(f"{C.WARN}[!]{C.END} {msg}")
```

```
def fail(msg: str) -> None:
```

```
    print(f"{C.FAIL}[-]{C.END} {msg}")
```

```
# -----
```

```
# Pre-flight checks
```

```
# -----
```

```
def check_tcp_port(host: str, port: int, timeout: float = 5.0) -> bool:
```

```
    """Quick TCP probe before attempting the full TLS+WinRM handshake."""
```

```
    info(f"Testing TCP connectivity to {host}:{port} ...")
```

```
    try:
```

```
        with closing(socket.create_connection((host, port), timeout=timeout)):
```

```
            ok(f"TCP port {port} is OPEN on {host}")
```

```
            return True
```

```
    except socket.timeout:
```

```
        fail(f"TCP port {port} timed out (firewall? listener not configured?)")
```

```
    except ConnectionRefusedError:
```

```
        fail(f"TCP port {port} refused the connection (HTTPS listener not configured?)")
```

```

except socket.gaierror:
    fail(f"Could not resolve hostname '{host}' (DNS issue)")
except OSError as e:
    fail(f"TCP error on port {port}: {e}")
return False

def check_tls(host: str, port: int = 5986, timeout: float = 5.0) -> None:
    """Inspect the TLS certificate on the HTTPS listener (does NOT validate it)."""
    info(f"Inspecting TLS certificate on {host}:{port} ...")
    ctx = ssl.create_default_context()
    ctx.check_hostname = False
    ctx.verify_mode = ssl.CERT_NONE
    try:
        with socket.create_connection((host, port), timeout=timeout) as sock:
            with ctx.wrap_socket(sock, server_hostname=host) as ssock:
                cert = ssock.getpeercert(binary_form=False)
                if cert:
                    subject = dict(x[0] for x in cert.get("subject", []))
                    issuer = dict(x[0] for x in cert.get("issuer", []))
                    ok("TLS handshake OK.")
                    print(f"    Subject : {subject}")
                    print(f"    Issuer  : {issuer}")
                    print(f"    Valid until: {cert.get('notAfter')}")
                else:
                    der = ssock.getpeercert(binary_form=True)
                    warn(f"TLS handshake OK. Cert hidden (likely self-signed, {len(der)}B
DER).")
    except Exception as e:
        warn(f"Could not retrieve TLS info: {e}")

# -----
# WinRM/HTTPS session
# -----
def open_session(host: str, port: int, user: str, password: str,
                transport: str, verify_ssl: bool) -> winrm.Session:
    endpoint = f"https://{host}:{port}/wsman"
    info(f"Opening WinRM session over HTTPS: {endpoint} (transport={transport})")

```

```

return winrm.Session(
    endpoint,
    auth=(user, password),
    transport=transport,
    server_cert_validation="validate" if verify_ssl else "ignore",
)

def run_ps(session: winrm.Session, script: str) -> tuple[int, str, str]:
    """Run a PowerShell script remotely. Return (exit_code, stdout, stderr)."""
    r = session.run_ps(script)
    return (
        r.status_code,
        r.std_out.decode("utf-8", errors="replace"),
        r.std_err.decode("utf-8", errors="replace"),
    )

def ping_winrm(session: winrm.Session) -> bool:
    """Confirm the session works with a trivial command before the heavy lifting."""
    try:
        code, out, err = run_ps(session, "$env:COMPUTERNAME")
        if code == 0:
            ok(f"WinRM authenticated. Remote computer: {out.strip()}")
            return True
        fail(f"WinRM command returned exit code {code}: {err.strip()}")
    except InvalidCredentialsError:
        fail("Invalid credentials.")
    except WinRMTransportError as e:
        fail(f"WinRM transport error: {e}")
    except WinRMOperationTimeoutError as e:
        fail(f"WinRM operation timed out: {e}")
    except Exception as e:
        fail(f"Unexpected WinRM error: {type(e).__name__}: {e}")
    return False

# -----
# Active Directory enumeration scripts

```

```

# -----
# We ask PowerShell to emit JSON so Python can parse it reliably.
# ConvertTo-Json -Depth 4 -Compress gives us a single-line array.
# -ResultPageSize 1000 makes the AD cmdlets page through large directories
# efficiently instead of pulling everything at once.

AD_PROBE_PS = r"""
$ErrorActionPreference = 'Stop'
try {
    Import-Module ActiveDirectory -ErrorAction Stop
    Write-Output 'MODULE_OK'
} catch {
    Write-Output "MODULE_MISSING: $($_.Exception.Message)"
    exit 2
}
"""

def build_users_query(search_base: str | None) -> str:
    base_param = f"-SearchBase '{search_base}'" if search_base else ""
    return rf"""
        $ErrorActionPreference = 'Stop'
        Import-Module ActiveDirectory

        $props = @(
            'SamAccountName', 'UserPrincipalName', 'DisplayName', 'GivenName', 'Surname',
            'EmailAddress', 'Title', 'Department', 'Company', 'Manager', 'Office',
            'Enabled', 'LockedOut', 'PasswordLastSet', 'PasswordNeverExpires',
            'LastLogonDate', 'WhenCreated', 'WhenChanged', 'DistinguishedName',
            'MemberOf', 'SID', 'ObjectGUID'
        )

        $users = Get-ADUser -Filter * {base_param} `
            -ResultPageSize 1000 -Properties $props |
            ForEach-Object {{
                [PSCustomObject]@{{
                    SamAccountName      = $_.SamAccountName
                    UserPrincipalName    = $_.UserPrincipalName
                    DisplayName          = $_.DisplayName

```

```

        GivenName           = $_.GivenName
        Surname             = $_.Surname
        EmailAddress        = $_.EmailAddress
        Title               = $_.Title
        Department          = $_.Department
        Company             = $_.Company
        Office              = $_.Office
        Enabled             = $_.Enabled
        LockedOut           = $_.LockedOut
        PasswordNeverExpires = $_.PasswordNeverExpires
        PasswordLastSet     = if ($_.PasswordLastSet) {{
$_.PasswordLastSet.ToString('o') }} else {{ $null }}
        LastLogonDate      = if ($_.LastLogonDate) {{
$_.LastLogonDate.ToString('o') }} else {{ $null }}
        WhenCreated        = if ($_.WhenCreated) {{
$_.WhenCreated.ToString('o') }} else {{ $null }}
        WhenChanged        = if ($_.WhenChanged) {{
$_.WhenChanged.ToString('o') }} else {{ $null }}
        DistinguishedName  = $_.DistinguishedName
        SID                 = $_.SID.Value
        ObjectGUID          = $_.ObjectGUID.Guid
        GroupCount         = @($_.MemberOf).Count
    }}
}}

```

```

# ConvertTo-Json wraps a single object in an object (not array). Force an array.

```

```

, @($users) | ConvertTo-Json -Depth 4 -Compress

```

```

"""

```

```

def build_computers_query(search_base: str | None) -> str:

```

```

    base_param = f"-SearchBase '{search_base}'" if search_base else ""

```

```

    return rf"""

```

```

        $ErrorActionPreference = 'Stop'

```

```

        Import-Module ActiveDirectory

```

```

    $props = @(

```

```

        'Name', 'DNSHostName', 'SamAccountName', 'Enabled', 'OperatingSystem',

```

```

        'OperatingSystemVersion', 'OperatingSystemServicePack', 'IPv4Address',

```

```

        'IPv6Address', 'LastLogonDate', 'PasswordLastSet', 'WhenCreated', 'WhenChanged',
        'DistinguishedName', 'SID', 'ObjectGUID', 'Description', 'ManagedBy'
    )

    $computers = Get-ADComputer -Filter * {base_param} `
        -ResultPageSize 1000 -Properties $props |
        ForEach-Object {{
            [PSCustomObject]@{{
                Name                = $_.Name
                DNSHostName         = $_.DNSHostName
                SamAccountName      = $_.SamAccountName
                Enabled             = $_.Enabled
                OperatingSystem     = $_.OperatingSystem
                OperatingSystemVersion = $_.OperatingSystemVersion
                OperatingSystemSP   = $_.OperatingSystemServicePack
                IPv4Address         = $_.IPv4Address
                IPv6Address         = $_.IPv6Address
                Description         = $_.Description
                ManagedBy           = $_.ManagedBy
                LastLogonDate       = if ($_.LastLogonDate) {{
                    $_.LastLogonDate.ToString('o') }} else {{ $null }}
                PasswordLastSet    = if ($_.PasswordLastSet) {{
                    $_.PasswordLastSet.ToString('o') }} else {{ $null }}
                WhenCreated        = if ($_.WhenCreated) {{
                    $_.WhenCreated.ToString('o') }} else {{ $null }}
                WhenChanged        = if ($_.WhenChanged) {{
                    $_.WhenChanged.ToString('o') }} else {{ $null }}
                DistinguishedName  = $_.DistinguishedName
                SID                 = $_.SID.Value
                ObjectGUID         = $_.ObjectGUID.Guid
            }}
        }}

    ,@($computers) | ConvertTo-Json -Depth 4 -Compress
    ""

# -----
# Run queries and parse results

```

```

# -----
def parse_json_output(raw: str) -> list[dict]:
    """
    PowerShell's ConvertTo-Json sometimes emits BOM/whitespace.
    Strip and parse; always return a list of dicts (even for 0/1 items).
    """
    raw = raw.strip().rstrip("\uffff")
    if not raw:
        return []
    data = json.loads(raw)
    if isinstance(data, dict):
        return [data]
    return list(data)

def enumerate_objects(session: winrm.Session, object_type: str,
                      search_base: str | None) -> list[dict]:
    if object_type == "users":
        info("Enumerating ALL user objects from Active Directory ...")
        script = build_users_query(search_base)
    elif object_type == "computers":
        info("Enumerating ALL computer objects from Active Directory ...")
        script = build_computers_query(search_base)
    else:
        raise ValueError(f"Unknown object_type: {object_type}")

    code, out, err = run_ps(session, script)
    if code != 0:
        fail(f"PowerShell query failed (exit {code})")
        if err.strip():
            print(f"    stderr: {err.strip()}")
        return []

    try:
        objects = parse_json_output(out)
    except json.JSONDecodeError as e:
        fail(f"Could not parse JSON returned by PowerShell: {e}")
        print("    First 500 chars of output:")
        print(f"    {out[:500]}")

```

```

    return []

    ok(f"Retrieved {len(objects)} {object_type} from Active Directory.")
    return objects

# -----
# Output writers
# -----
def print_table(objects: list[dict], object_type: str, limit: int = 20) -> None:
    """Print a brief preview table to stdout (full data goes to files)."""
    if not objects:
        warn(f"No {object_type} to display.")
        return

    if object_type == "users":
        cols = ["SamAccountName", "DisplayName", "Enabled", "LastLogonDate", "Department"]
    else:
        cols = ["Name", "DNSHostName", "OperatingSystem", "Enabled", "LastLogonDate"]

    # Compute column widths
    widths = {c: max(len(c), 8) for c in cols}
    for obj in objects[:limit]:
        for c in cols:
            v = str(obj.get(c, "") or "")
            widths[c] = min(max(widths[c], len(v)), 40)

    # Header
    print()
    header = " | ".join(c.ljust(widths[c]) for c in cols)
    print(f"{C.BOLD}{header}{C.END}")
    print("-+-".join("-" * widths[c] for c in cols))

    # Rows
    for obj in objects[:limit]:
        row = " | ".join(str(obj.get(c, "") or "")[:widths[c]].ljust(widths[c]) for c in cols)
        print(row)

    if len(objects) > limit:

```

```

    print(f"... ({len(objects) - limit} more rows not shown – see exported file)")

def write_csv(objects: list[dict], path: Path) -> None:
    if not objects:
        warn(f"Nothing to write to {path}")
        return
    # Union of keys preserves all columns even if some objects have nulls
    fieldnames: list[str] = []
    seen: set[str] = set()
    for obj in objects:
        for k in obj.keys():
            if k not in seen:
                seen.add(k)
                fieldnames.append(k)

    with path.open("w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(objects)
    ok(f"Wrote {len(objects)} rows -> {path}")

def write_json(objects: list[dict], path: Path) -> None:
    with path.open("w", encoding="utf-8") as f:
        json.dump(objects, f, indent=2, ensure_ascii=False)
    ok(f"Wrote {len(objects)} objects -> {path}")

# -----
# CLI
# -----

def parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser(
        description="Enumerate all users and computers from Active Directory over
WinRM/HTTPS.",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=__doc__,
    )

```

```

p.add_argument("--host", required=True, help="Target Domain Controller hostname or IP")
p.add_argument("--user", required=True,
                help=r"Username (DOMAIN\user, user@domain.local, etc.)")
p.add_argument("--password", help="Password (prompted if omitted)")
p.add_argument("--port", type=int, default=5986, help="HTTPS WinRM port (default 5986)")
p.add_argument(
    "--transport",
    default="ntlm",
    choices=["ntlm", "kerberos", "ssl", "credssp"],
    help="WinRM auth transport (default: ntlm)",
)
p.add_argument("--verify-ssl", action="store_true",
                help="Strictly validate the server's TLS certificate "
                "(default: ignore, useful for self-signed)")
p.add_argument(
    "--object-type",
    choices=["users", "computers", "all"],
    default="all",
    help="Which object type(s) to retrieve (default: all)",
)
p.add_argument("--search-base", metavar="DN",
                help="Restrict search to an OU/DN (e.g. 'OU=Sales,DC=corp,DC=local')")
p.add_argument("--output-dir", metavar="DIR",
                help="Directory to write CSV/JSON files into (created if missing)")
p.add_argument(
    "--format",
    choices=["csv", "json", "both"],
    default="csv",
    help="Output file format when --output-dir is given (default: csv)",
)
p.add_argument("--preview-rows", type=int, default=20,
                help="How many rows to show in the on-screen preview (default 20)")
return p.parse_args()

```

```

def main() -> int:
    args = parse_args()
    password = args.password or getpass.getpass(f"Password for {args.user}: ")

```

```
banner(f"WinRM/HTTPS Active Directory enumeration - {args.host}:{args.port}")

# --- pre-flight ---
if not check_tcp_port(args.host, args.port):
    fail("Aborting: HTTPS port unreachable.")
    return 1

check_tls(args.host, args.port)

# --- session ---
try:
    session = open_session(
        host=args.host,
        port=args.port,
        user=args.user,
        password=password,
        transport=args.transport,
        verify_ssl=args.verify_ssl,
    )
except Exception as e:
    fail(f"Could not build WinRM session: {e}")
    return 1

if not ping_winrm(session):
    return 1

# --- AD module probe ---
code, out, err = run_ps(session, AD_PROBE_PS)
if "MODULE_OK" not in out:
    fail("ActiveDirectory PowerShell module is not available on the remote host.")
    print(f"    stdout: {out.strip()}")
    print(f"    stderr: {err.strip()}")
    print("    Tip: target a Domain Controller, or install RSAT-AD-PowerShell.")
    return 1
ok("ActiveDirectory module is loaded on the remote host.")

# --- output dir setup ---
out_dir: Path | None = None
if args.output_dir:
```

```

out_dir = Path(args.output_dir).expanduser().resolve()
out_dir.mkdir(parents=True, exist_ok=True)
info(f"Output directory: {out_dir}")

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
types_to_fetch = ["users", "computers"] if args.object_type == "all" else
[args.object_type]

exit_code = 0
for ot in types_to_fetch:
    banner(f"Active Directory – {ot.upper()}")
    objects = enumerate_objects(session, ot, args.search_base)
    if not objects:
        exit_code = exit_code or 2
        continue

    print_table(objects, ot, limit=args.preview_rows)

    if out_dir:
        stem = f"ad_{ot}_{timestamp}"
        if args.format in ("csv", "both"):
            write_csv(objects, out_dir / f"{stem}.csv")
        if args.format in ("json", "both"):
            write_json(objects, out_dir / f"{stem}.json")

    banner("Done")
    return exit_code

if __name__ == "__main__":
    try:
        sys.exit(main())
    except KeyboardInterrupt:
        print("\n[!] Interrupted by user.")
        sys.exit(130)

```

What this script does

HTTPS-only by design — no port 5985 fallback, no HTTP code paths. It connects to port 5986, verifies the TLS handshake, then enumerates **every** user and computer object from Active

Directory.

Pipeline

1. **TCP probe** on port 5986 — fails fast if the HTTPS listener isn't configured.
2. **TLS inspection** — pulls the cert subject/issuer/expiry so you can see what's on the wire (works even with self-signed certs).
3. **WinRM session** over HTTPS with NTLM (default) or Kerberos/CredSSP/SSL auth.
4. **AD module probe** — gracefully bails if the host isn't a DC and doesn't have RSAT installed.
5. **Bulk enumeration** — uses PowerShell's `Get-ADUser -Filter * -ResultPageSize 1000` (and the same for `Get-ADComputer`), which **pages through the entire directory** rather than hitting the LDAP page-size limit (default 1000).
6. **JSON-over-WinRM** — PowerShell emits `ConvertTo-Json -Compress`, Python parses it directly into dicts. No fragile text-table scraping.
7. **Output** — preview table on screen, optional CSV and/or JSON files in `--output-dir`.

Fields retrieved

For each user (22 fields): SamAccountName, UserPrincipalName, DisplayName, GivenName, Surname, EmailAddress, Title, Department, Company, Office, Enabled, LockedOut, PasswordLastSet, PasswordNeverExpires, LastLogonDate, WhenCreated, WhenChanged, DistinguishedName, SID, ObjectGUID, GroupCount.

For each computer (18 fields): Name, DNSHostName, SamAccountName, Enabled, OperatingSystem, OperatingSystemVersion, OperatingSystemServicePack, IPv4Address, IPv6Address, Description, ManagedBy, LastLogonDate, PasswordLastSet, WhenCreated, WhenChanged, DistinguishedName, SID, ObjectGUID.

All datetimes are converted to ISO-8601 (`.ToString('o')`) so they round-trip cleanly into CSV/JSON and can be parsed in any language.

Usage examples

bash

```
# Pull everything, just preview to screen
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin'

# Pull everything, save to ./ad_export as CSV
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
  --output-dir ./ad_export --format csv

# Both CSV and JSON, only users
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
```

```

--object-type users --format both --output-dir ./ad_export

# Limit to a single OU
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' \
  --search-base "OU=Employees,DC=corp,DC=local" \
  --output-dir ./ad_export

# Production: strict TLS verification (cert signed by a CA your client trusts)
python winrm_ad_enum.py --host dc01.corp.local --user 'CORP\admin' --verify-ssl

# Kerberos auth from a Linux box with valid ticket (kinit beforehand)
python winrm_ad_enum.py --host dc01.corp.local --user admin@CORP.LOCAL \
  --transport kerberos

```

Sample output (preview)

When you run it against a real DC, the on-screen preview looks like:

```

=====
Active Directory – USERS
=====
[*] Enumerating ALL user objects from Active Directory ...
[+] Retrieved 247 users from Active Directory.

SamAccountName | DisplayName          | Enabled | LastLogonDate          | Department
+-----+-----+-----+-----+-----+
Administrator | Administrator        | True    | 2026-05-19T08:12:03+00:00 | 
asmith         | Alice Smith          | True    | 2026-05-19T07:30:11+00:00 | Sales
bwilliams     | Bob Williams         | True    | 2026-05-15T14:10:55+00:00 | Engineering
...
... (227 more rows not shown – see exported file)

[+] Wrote 247 rows -> /home/you/ad_export/ad_users_20260519_153022.csv

```

Output files

If you pass `--output-dir`, you'll get timestamped files like:

```
ad_export/
├─ ad_users_20260519_153022.csv
├─ ad_users_20260519_153022.json
├─ ad_computers_20260519_153022.csv
└─ ad_computers_20260519_153022.json
```

The CSV uses the **union of all keys** across objects as the header row, so even sparsely-populated attributes (e.g., a user with no `Department`) keep their columns aligned. The JSON is `indent=2` and UTF-8, ready to load into pandas, jq, or any analytics tool.

A few things to know

- **Pagination is automatic.** `-ResultPageSize 1000` tells the AD cmdlets to fetch the directory in 1000-object chunks, which avoids hitting the LDAP server's `MaxPageSize` policy. For directories with tens of thousands of objects, this just works — no manual paging needed.
- **Memory.** A directory with 50,000 users will produce a JSON blob of roughly 30-50 MB transferred over WinRM. WinRM's default message size limit (`MaxEnvelopeSizeKb = 500`) should be fine since `-Compress` is used, but if you have huge directories and hit `WSManFault`, raise it on the server: `Set-Item WSMAN:\localhost\MaxEnvelopeSizeKb 8192`.
- **Permissions.** Authenticated users can read most of the directory by default, but `LastLogonDate`, `PasswordLastSet`, and `LockedOut` may show as null if the account doesn't have read access to those attributes. Run as a Domain Admin (or a delegated read account) to see everything.
- `--verify-ssl`. With self-signed certs, leave this off — the connection still uses TLS encryption, you just don't validate the cert chain. For production, install a CA-signed cert on the DC and use `--verify-ssl`.

Want me to add anything else — for example, also pulling **groups** with their member lists, or filtering by `LastLogonDate` to find stale accounts before exporting?

Winrm Active Directory Enumeration using HTTP (5985)

winrm_ad_enum-py

```
#!/usr/bin/env python3
"""
WinRM Active Directory Object Enumerator (HTTP-only, port 5985)
=====

Connects to a Windows Domain Controller over WinRM/HTTP (port 5985) and
retrieves ALL user and computer objects from Active Directory.

About transport security:
    Port 5985 is plain HTTP at the transport level, but Windows WinRM still
    wraps the WS-Management SOAP payload in NTLM or Kerberos message-level
    encryption by default. That means credentials and AD data are NOT in
    cleartext on the wire, even though the URL is http://. The exception is
    the 'basic' transport: that one DOES send credentials base64-encoded and
    should never be used over HTTP unless you've explicitly enabled
    'AllowUnencrypted=true' on the server (don't).

    Default transport here: NTLM with message-level encryption – safe for
    lab use and typical for AD-joined environments.

Requirements:
    pip install pywinrm

Usage:
    # Pull everything, preview on screen
    python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin'

    # Save to CSV in ./ad_export/
    python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin' \
        --output-dir ./ad_export --format csv
```

```

# Only computers, JSON output
python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin' \
    --object-type computers --format json --output-dir ./ad_export

# Restrict to a specific OU
python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin' \
    --search-base "OU=Employees,DC=corp,DC=local"
"""

from __future__ import annotations

import argparse
import csv
import getpass
import json
import socket
import sys

from contextlib import closing
from datetime import datetime
from pathlib import Path

try:
    import winrm
    from winrm.exceptions import (
        InvalidCredentialsError,
        WinRMOperationTimeoutError,
        WinRMTransportError,
    )
except ImportError:
    print("[!] The 'pywinrm' package is required. Install it with:")
    print("    pip install pywinrm")
    sys.exit(1)

# -----
# Output helpers
# -----

class C:
    OK = "\033[92m"

```

```
WARN = "\033[93m"
FAIL = "\033[91m"
INFO = "\033[94m"
BOLD = "\033[1m"
END = "\033[0m"
```

```
def banner(text: str) -> None:
    line = "=" * 72
    print(f"\n{C.BOLD}{line}\n{text}\n{line}{C.END}")
```

```
def info(msg: str) -> None:
    print(f"{C.INFO}[*]{C.END} {msg}")
```

```
def ok(msg: str) -> None:
    print(f"{C.OK}[+]{C.END} {msg}")
```

```
def warn(msg: str) -> None:
    print(f"{C.WARN}[!]{C.END} {msg}")
```

```
def fail(msg: str) -> None:
    print(f"{C.FAIL}[-]{C.END} {msg}")
```

```
# -----
```

```
# Pre-flight checks
```

```
# -----
```

```
def check_tcp_port(host: str, port: int, timeout: float = 5.0) -> bool:
    """Quick TCP probe before attempting the WinRM handshake."""
    info(f"Testing TCP connectivity to {host}:{port} ...")
    try:
        with closing(socket.create_connection((host, port), timeout=timeout)):
            ok(f"TCP port {port} is OPEN on {host}")
            return True
    except socket.timeout:
```

```

        fail(f"TCP port {port} timed out (firewall? listener not configured?)")
except ConnectionRefusedError:
    fail(f"TCP port {port} refused the connection (HTTP listener not configured?)")
except socket.gaierror:
    fail(f"Could not resolve hostname '{host}' (DNS issue)")
except OSError as e:
    fail(f"TCP error on port {port}: {e}")
return False

# -----
# WinRM/HTTP session
# -----
def open_session(host: str, port: int, user: str, password: str,
                transport: str) -> winrm.Session:
    endpoint = f"http://{host}:{port}/wsman"
    info(f"Opening WinRM session over HTTP: {endpoint} (transport={transport})")
    if transport == "basic":
        warn("Using 'basic' over HTTP – credentials will be sent base64-encoded "
            "in cleartext. Only do this in a trusted lab.")
    return winrm.Session(
        endpoint,
        auth=(user, password),
        transport=transport,
        # server_cert_validation is irrelevant for HTTP, but pywinrm accepts it
        server_cert_validation="ignore",
    )

def run_ps(session: winrm.Session, script: str) -> tuple[int, str, str]:
    """Run a PowerShell script remotely. Return (exit_code, stdout, stderr)."""
    r = session.run_ps(script)
    return (
        r.status_code,
        r.std_out.decode("utf-8", errors="replace"),
        r.std_err.decode("utf-8", errors="replace"),
    )

```

```

def ping_winrm(session: winrm.Session) -> bool:
    """Confirm the session works with a trivial command before the heavy lifting."""
    try:
        code, out, err = run_ps(session, "$env:COMPUTERNAME")
        if code == 0:
            ok(f"WinRM authenticated. Remote computer: {out.strip()}")
            return True
        fail(f"WinRM command returned exit code {code}: {err.strip()}")
    except InvalidCredentialsError:
        fail("Invalid credentials.")
    except WinRMTransportError as e:
        msg = str(e)
        fail(f"WinRM transport error: {msg}")
        # Common HTTP-specific hint
        if "401" in msg or "Unauthorized" in msg:
            print("    Hint: on HTTP, the server often requires NTLM/Kerberos "
                  "(not Basic). Try --transport ntlm.")
        elif "AllowUnencrypted" in msg:
            print("    Hint: server requires encryption. NTLM/Kerberos provide "
                  "message-level encryption – use --transport ntlm.")
    except WinRMOperationTimeoutError as e:
        fail(f"WinRM operation timed out: {e}")
    except Exception as e:
        fail(f"Unexpected WinRM error: {type(e).__name__}: {e}")
    return False

# -----
# Active Directory enumeration scripts
# -----
AD_PROBE_PS = r"""
$ErrorActionPreference = 'Stop'
try {
    Import-Module ActiveDirectory -ErrorAction Stop
    Write-Output 'MODULE_OK'
} catch {
    Write-Output "MODULE_MISSING: $($_.Exception.Message)"
    exit 2
}

```

```
"""
```

```
def build_users_query(search_base: str | None) -> str:
    base_param = f"-SearchBase '{search_base}'" if search_base else ""
    return rf"""
        $ErrorActionPreference = 'Stop'
        Import-Module ActiveDirectory

        $props = @(
            'SamAccountName', 'UserPrincipalName', 'DisplayName', 'GivenName', 'Surname',
            'EmailAddress', 'Title', 'Department', 'Company', 'Manager', 'Office',
            'Enabled', 'LockedOut', 'PasswordLastSet', 'PasswordNeverExpires',
            'LastLogonDate', 'WhenCreated', 'WhenChanged', 'DistinguishedName',
            'MemberOf', 'SID', 'ObjectGUID'
        )

        $users = Get-ADUser -Filter * {base_param} `
            -ResultPageSize 1000 -Properties $props |
            ForEach-Object {{
                [PSCustomObject]@{{
                    SamAccountName      = $_.SamAccountName
                    UserPrincipalName    = $_.UserPrincipalName
                    DisplayName           = $_.DisplayName
                    GivenName             = $_.GivenName
                    Surname               = $_.Surname
                    EmailAddress          = $_.EmailAddress
                    Title                 = $_.Title
                    Department             = $_.Department
                    Company               = $_.Company
                    Office                = $_.Office
                    Enabled               = $_.Enabled
                    LockedOut             = $_.LockedOut
                    PasswordNeverExpires = $_.PasswordNeverExpires
                    PasswordLastSet      = if ($_.PasswordLastSet) {{
$_.PasswordLastSet.ToString('o') }} else {{ $null }}
                    LastLogonDate       = if ($_.LastLogonDate) {{
$_.LastLogonDate.ToString('o') }} else {{ $null }}
                    WhenCreated          = if ($_.WhenCreated) {{
```

```

    $_.WhenCreated.ToString('o') }}      else {{ $null }}
        WhenChanged                      = if ($_.WhenChanged)    {{
$_.WhenChanged.ToString('o') }}      else {{ $null }}
        DistinguishedName                 = $_.DistinguishedName
        SID                               = $_.SID.Value
        ObjectGUID                        = $_.ObjectGUID.Guid
        GroupCount                        = @($_.MemberOf).Count
    }}
}}

```

ConvertTo-Json on a single object emits an object, not an array.

The leading comma forces an array context.

```
,@($users) | ConvertTo-Json -Depth 4 -Compress
```

```
""
```

```
def build_computers_query(search_base: str | None) -> str:
```

```
    base_param = f"-SearchBase '{search_base}'" if search_base else ""
```

```
    return rf""
```

```
    $ErrorActionPreference = 'Stop'
```

```
    Import-Module ActiveDirectory
```

```
    $props = @(
```

```
        'Name', 'DNSHostName', 'SamAccountName', 'Enabled', 'OperatingSystem',
```

```
        'OperatingSystemVersion', 'OperatingSystemServicePack', 'IPv4Address',
```

```
        'IPv6Address', 'LastLogonDate', 'PasswordLastSet', 'WhenCreated', 'WhenChanged',
```

```
        'DistinguishedName', 'SID', 'ObjectGUID', 'Description', 'ManagedBy'
```

```
)
```

```
$computers = Get-ADComputer -Filter * {base_param} `
```

```
-ResultPageSize 1000 -Properties $props |
```

```
ForEach-Object {{
```

```
    [PSCustomObject]@{{
```

```
        Name = $_.Name
```

```
        DNSHostName = $_.DNSHostName
```

```
        SamAccountName = $_.SamAccountName
```

```
        Enabled = $_.Enabled
```

```
        OperatingSystem = $_.OperatingSystem
```

```
        OperatingSystemVersion = $_.OperatingSystemVersion
```

```

        OperatingSystemSP      = $_.OperatingSystemServicePack
        IPv4Address            = $_.IPv4Address
        IPv6Address            = $_.IPv6Address
        Description             = $_.Description
        ManagedBy               = $_.ManagedBy
        LastLogonDate           = if ($_.LastLogonDate) {{
$_.LastLogonDate.ToString('o') }} else {{ $null }}
        PasswordLastSet        = if ($_.PasswordLastSet) {{
$_.PasswordLastSet.ToString('o') }} else {{ $null }}
        WhenCreated             = if ($_.WhenCreated) {{
$_.WhenCreated.ToString('o') }} else {{ $null }}
        WhenChanged             = if ($_.WhenChanged) {{
$_.WhenChanged.ToString('o') }} else {{ $null }}
        DistinguishedName       = $_.DistinguishedName
        SID                     = $_.SID.Value
        ObjectGUID              = $_.ObjectGUID.Guid
    }}
}}

```

```

, @($computers) | ConvertTo-Json -Depth 4 -Compress

```

```

"""

```

```

# -----
# Run queries and parse results
# -----
def parse_json_output(raw: str) -> list[dict]:
    """Strip BOM/whitespace and always return a list of dicts."""
    raw = raw.strip().rstrip("\uffff")
    if not raw:
        return []
    data = json.loads(raw)
    if isinstance(data, dict):
        return [data]
    return list(data)

def enumerate_objects(session: winrm.Session, object_type: str,
                    search_base: str | None) -> list[dict]:

```

```

if object_type == "users":
    info("Enumerating ALL user objects from Active Directory ...")
    script = build_users_query(search_base)
elif object_type == "computers":
    info("Enumerating ALL computer objects from Active Directory ...")
    script = build_computers_query(search_base)
else:
    raise ValueError(f"Unknown object_type: {object_type}")

code, out, err = run_ps(session, script)
if code != 0:
    fail(f"PowerShell query failed (exit {code})")
    if err.strip():
        print(f"    stderr: {err.strip()}")
    return []

try:
    objects = parse_json_output(out)
except json.JSONDecodeError as e:
    fail(f"Could not parse JSON returned by PowerShell: {e}")
    print("    First 500 chars of output:")
    print(f"    {out[:500]}")
    return []

ok(f"Retrieved {len(objects)} {object_type} from Active Directory.")
return objects

# -----
# Output writers
# -----
def print_table(objects: list[dict], object_type: str, limit: int = 20) -> None:
    """Print a brief preview table to stdout (full data goes to files)."""
    if not objects:
        warn(f"No {object_type} to display.")
        return

    if object_type == "users":
        cols = ["SamAccountName", "DisplayName", "Enabled", "LastLogonDate", "Department"]

```

```

else:
    cols = ["Name", "DNSHostName", "OperatingSystem", "Enabled", "LastLogonDate"]

widths = {c: max(len(c), 8) for c in cols}
for obj in objects[:limit]:
    for c in cols:
        v = str(obj.get(c, "") or "")
        widths[c] = min(max(widths[c], len(v)), 40)

print()
header = " | ".join(c.ljust(widths[c]) for c in cols)
print(f"{C.BOLD}{header}{C.END}")
print("-+-.".join("-" * widths[c] for c in cols))

for obj in objects[:limit]:
    row = " | ".join(str(obj.get(c, "") or "")[:widths[c]].ljust(widths[c]) for c in cols)
    print(row)

if len(objects) > limit:
    print(f"... ({len(objects) - limit} more rows not shown – see exported file)")

def write_csv(objects: list[dict], path: Path) -> None:
    if not objects:
        warn(f"Nothing to write to {path}")
        return
    fieldnames: list[str] = []
    seen: set[str] = set()
    for obj in objects:
        for k in obj.keys():
            if k not in seen:
                seen.add(k)
                fieldnames.append(k)

    with path.open("w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(objects)
    ok(f"Wrote {len(objects)} rows -> {path}")

```

```

def write_json(objects: list[dict], path: Path) -> None:
    with path.open("w", encoding="utf-8") as f:
        json.dump(objects, f, indent=2, ensure_ascii=False)
    ok(f"Wrote {len(objects)} objects -> {path}")

# -----
# CLI
# -----
def parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser(
        description="Enumerate all users and computers from AD over WinRM/HTTP (port 5985).",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=__doc__,
    )
    p.add_argument("--host", required=True, help="Target Domain Controller hostname or IP")
    p.add_argument("--user", required=True,
        help=r"Username (DOMAIN\user, user@domain.local, etc.)")
    p.add_argument("--password", help="Password (prompted if omitted)")
    p.add_argument("--port", type=int, default=5985, help="HTTP WinRM port (default 5985)")
    p.add_argument(
        "--transport",
        default="ntlm",
        choices=["ntlm", "kerberos", "basic", "credssp"],
        help="WinRM auth transport (default: ntlm). 'basic' is INSECURE over HTTP.",
    )
    p.add_argument(
        "--object-type",
        choices=["users", "computers", "all"],
        default="all",
        help="Which object type(s) to retrieve (default: all)",
    )
    p.add_argument("--search-base", metavar="DN",
        help="Restrict search to an OU/DN (e.g. 'OU=Sales,DC=corp,DC=local')")
    p.add_argument("--output-dir", metavar="DIR",
        help="Directory to write CSV/JSON files into (created if missing)")
    p.add_argument(

```

```

    "--format",
    choices=["csv", "json", "both"],
    default="csv",
    help="Output file format when --output-dir is given (default: csv)",
)
p.add_argument("--preview-rows", type=int, default=20,
               help="How many rows to show in the on-screen preview (default 20)")
return p.parse_args()

def main() -> int:
    args = parse_args()
    password = args.password or getpass.getpass(f"Password for {args.user}: ")

    banner(f"WinRM/HTTP Active Directory enumeration – {args.host}:{args.port}")
    warn("Using HTTP transport (port 5985). NTLM/Kerberos provide message-level "
         "encryption; Basic does not.")

    if not check_tcp_port(args.host, args.port):
        fail("Aborting: HTTP port unreachable.")
        return 1

    try:
        session = open_session(
            host=args.host,
            port=args.port,
            user=args.user,
            password=password,
            transport=args.transport,
        )
    except Exception as e:
        fail(f"Could not build WinRM session: {e}")
        return 1

    if not ping_winrm(session):
        return 1

    code, out, err = run_ps(session, AD_PROBE_PS)
    if "MODULE_OK" not in out:

```

```

fail("ActiveDirectory PowerShell module is not available on the remote host.")
print(f"    stdout: {out.strip()}")
print(f"    stderr: {err.strip()}")
print("    Tip: target a Domain Controller, or install RSAT-AD-PowerShell.")
return 1
ok("ActiveDirectory module is loaded on the remote host.")

out_dir: Path | None = None
if args.output_dir:
    out_dir = Path(args.output_dir).expanduser().resolve()
    out_dir.mkdir(parents=True, exist_ok=True)
    info(f"Output directory: {out_dir}")

timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
types_to_fetch = ["users", "computers"] if args.object_type == "all" else
[args.object_type]

exit_code = 0
for ot in types_to_fetch:
    banner(f"Active Directory – {ot.upper()}")
    objects = enumerate_objects(session, ot, args.search_base)
    if not objects:
        exit_code = exit_code or 2
        continue

    print_table(objects, ot, limit=args.preview_rows)

    if out_dir:
        stem = f"ad_{ot}_{timestamp}"
        if args.format in ("csv", "both"):
            write_csv(objects, out_dir / f"{stem}.csv")
        if args.format in ("json", "both"):
            write_json(objects, out_dir / f"{stem}.json")

    banner("Done")
    return exit_code

if __name__ == "__main__":

```

```

try:
    sys.exit(main())
except KeyboardInterrupt:
    print("\n[!] Interrupted by user.")
    sys.exit(130)

```

What changed vs the HTTPS version

The script is identical in **what** it does — enumerate every user and computer with the same fields, same CSV/JSON output, same pagination — but the transport is different:

	HTTPS version	HTTP version (this one)
URL scheme	<code>https://...</code>	<code>http://...</code>
Default port	5986	5985
TLS check	<code>check_tls()</code> inspects cert	removed (no TLS to inspect)
<code>--verify-ssl</code> flag	yes	removed (no TLS)
<code>--transport ssl</code> option	yes	removed (no TLS)
<code>--transport basic</code>	available, encrypted by TLS	available, explicit warning — credentials base64-only

About "HTTP" being a misleading label

The most common misconception about WinRM on port 5985 is that it sends data in cleartext. It usually doesn't. Here's the actual picture:

- **NTLM transport** (this script's default) wraps the SOAP payload in **NTLM message-level encryption** (SPNEGO/GSSAPI sealing). Even though the URL says `http://`, the body on the wire is encrypted with a session key derived from your credentials. Someone sniffing the network sees ciphertext.
- **Kerberos transport** does the same with Kerberos session keys.
- **CredSSP transport** also encrypts, plus enables credential delegation.
- **Basic transport** is the *only* option that sends credentials base64-encoded and the payload in cleartext. Windows refuses Basic over HTTP by default (`AllowUnencrypted=false`). That's why I added the explicit warning when `--transport basic` is selected — and you really shouldn't use it outside a closed lab.

So in practice, this HTTP script is approximately as confidential as the HTTPS one *as long as you stick with NTLM or Kerberos*. The HTTPS version still wins for two reasons: it doesn't rely on the auth-protocol's sealing being correctly negotiated, and TLS gives you server-identity verification (the cert), which NTLM alone does not.

Usage examples

bash

```
# Default: NTLM auth over HTTP/5985
python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin'

# Save everything to CSV
python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin' \
  --output-dir ./ad_export --format csv

# Only computers, JSON output, narrowed to one OU
python winrm_ad_enum_http.py --host dc01.corp.local --user 'CORP\admin' \
  --object-type computers --format json \
  --search-base "OU=Servers,DC=corp,DC=local" \
  --output-dir ./ad_export

# Kerberos auth (Linux client needs valid ticket from `kinit`)
python winrm_ad_enum_http.py --host dc01.corp.local --user admin@CORP.LOCAL \
  --transport kerberos

# Lab-only: Basic auth (server must have AllowUnencrypted=true and Basic enabled)
python winrm_ad_enum_http.py --host dc01.local --user admin --transport basic
```

If you get a 401 or "AllowUnencrypted" error

The script catches `WinRMTransportError` and prints a hint, but here's the field reference:

- **401 Unauthorized with NTLM** → The server may require Kerberos, or the user has no logon right. Try `--transport kerberos` (after `kinit`) or check the account's "Allow log on through Windows Remote Management" privilege.
- **"The WS-Management service does not support the request"** with Basic → `AllowUnencrypted` is false on the server (the secure default). Either switch to `--transport ntlm`, or on the Windows server: `Set-Item WSMAN:\LocalHost\Service\AllowUnencrypted -Value $true` and enable Basic in `WSMAN:\LocalHost\Service\Auth\Basic`. (Lab only — don't do this in production.)
- **403 / "TrustedHosts"** errors generally come from the **client** side on Windows, not Linux/pywinrm — not applicable here.

You now have three scripts

```
winrm_ad_test.py          # Both ports + connectivity tests + many AD queries
winrm_ad_enum.py         # HTTPS-only, bulk user/computer export
winrm_ad_enum_http.py    # HTTP-only, bulk user/computer export ← this one
```

Want me to merge the two enumeration scripts into a single dual-mode tool with a `--scheme` `http|https` flag so you only have to maintain one file?

Winrm Active Directory Diagnostics using HTTPS

winrm_https_diag.py

```
#!/usr/bin/env python3
"""
WinRM/HTTPS diagnostic – run this when winrm_ad_enum.py returns nothing.

Tests each layer separately and prints raw output at every step:
    1. TCP reachability on 5986
    2. TLS handshake + cert details
    3. WinRM session + trivial command ($env:COMPUTERNAME)
    4. ActiveDirectory module availability on the remote host
    5. Raw user count (single integer, easiest thing to debug)
    6. Single-user JSON sample (so we can see what PowerShell actually returns)

Usage:
    python winrm_https_diag.py --host dc01.corp.local --user 'CORP\admin'
"""
import argparse
import getpass
import socket
import ssl
import sys
from contextlib import closing

import winrm

GREEN = "\033[92m"; RED = "\033[91m"; YEL = "\033[93m"; BLUE = "\033[94m"; END = "\033[0m"

def step(n, title):
    print(f"\n{BLUE}— STEP {n}: {title} —{END}")

def ok(msg):
    print(f"{GREEN}[OK]{END} {msg}")

def bad(msg):
    print(f"{RED}[FAIL]{END} {msg}")
```

```

def warn(msg): print(f"{YEL}[WARN]{END} {msg}")

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--host", required=True)
    ap.add_argument("--user", required=True)
    ap.add_argument("--password")
    ap.add_argument("--port", type=int, default=5986)
    ap.add_argument("--transport", default="ntlm",
                    choices=["ntlm", "kerberos", "ssl", "credssp"])
    args = ap.parse_args()
    pw = args.password or getpass.getpass(f"Password for {args.user}: ")

    # ----- Step 1: TCP -----
    step(1, f"TCP connectivity to {args.host}:{args.port}")
    try:
        with closing(socket.create_connection((args.host, args.port), timeout=5)):
            ok(f"TCP {args.port} is open")
    except Exception as e:
        bad(f"TCP failed: {type(e).__name__}: {e}")
        return 1

    # ----- Step 2: TLS -----
    step(2, "TLS handshake")
    ctx = ssl.create_default_context()
    ctx.check_hostname = False
    ctx.verify_mode = ssl.CERT_NONE
    try:
        with socket.create_connection((args.host, args.port), timeout=5) as s:
            with ctx.wrap_socket(s, server_hostname=args.host) as ss:
                cipher = ss.cipher()
                cert = ss.getpeercert(binary_form=True)
                ok(f"TLS handshake succeeded. Cipher: {cipher}")
                ok(f"Server presented a cert ({len(cert)} bytes DER)")
    except Exception as e:
        bad(f"TLS handshake failed: {type(e).__name__}: {e}")
        return 1

```

```

# ----- Step 3: WinRM session + trivial command -----
step(3, f"WinRM session ({args.transport})")
endpoint = f"https://{args.host}:{args.port}/wsman"
print(f"    Endpoint: {endpoint}")
try:
    session = winrm.Session(
        endpoint,
        auth=(args.user, pw),
        transport=args.transport,
        server_cert_validation="ignore",
    )
except Exception as e:
    bad(f"Could not build session: {type(e).__name__}: {e}")
    return 1

try:
    r = session.run_ps("$env:COMPUTERNAME")
    out = r.std_out.decode(errors="replace").strip()
    err = r.std_err.decode(errors="replace").strip()
    print(f"    exit_code = {r.status_code}")
    print(f"    stdout    = {out!r}")
    print(f"    stderr    = {err!r}")
    if r.status_code == 0 and out:
        ok(f"WinRM works. Remote computer = {out}")
    else:
        bad("WinRM responded but command failed. Check stderr above.")
        return 1
except Exception as e:
    bad(f"WinRM command failed: {type(e).__name__}: {e}")
    return 1

# ----- Step 4: AD module -----
step(4, "ActiveDirectory PowerShell module")
probe = r"""
$ErrorActionPreference = 'Stop'
try {
    Import-Module ActiveDirectory -ErrorAction Stop
    Write-Output "MODULE_OK"
    Write-Output ("AD module version: " + (Get-Module ActiveDirectory).Version)
}

```

```

} catch {
    Write-Output "MODULE_MISSING: $($_.Exception.Message)"
    exit 2
}
"""

    r = session.run_ps(probe)
    out = r.std_out.decode(errors="replace").strip()
    err = r.std_err.decode(errors="replace").strip()
    print(f"        exit_code = {r.status_code}")
    print(f"        stdout    = {out!r}")
    print(f"        stderr     = {err!r}")
    if "MODULE_OK" not in out:
        bad("ActiveDirectory module is NOT available on the remote host.")
        print("        Fix: connect to a Domain Controller, or install RSAT-AD-PowerShell")
        print("        on the target box: Install-WindowsFeature RSAT-AD-PowerShell")
        return 1
    ok("AD module loaded")

# ----- Step 5: Raw counts -----
step(5, "Raw object counts (sanity check)")
count_script = r"""
$ErrorActionPreference = 'Stop'
Import-Module ActiveDirectory
$u = @(Get-ADUser -Filter *).Count
$c = @(Get-ADComputer -Filter *).Count
Write-Output "USERS=$u"
Write-Output "COMPUTERS=$c"
"""

    r = session.run_ps(count_script)
    out = r.std_out.decode(errors="replace").strip()
    err = r.std_err.decode(errors="replace").strip()
    print(f"        exit_code = {r.status_code}")
    print(f"        stdout    = {out!r}")
    print(f"        stderr     = {err!r}")
    if r.status_code != 0:
        bad("Count query failed – see stderr above.")
        return 1
    if "USERS=0" in out and "COMPUTERS=0" in out:
        warn("Both counts are 0. The account may lack read permission on AD,")

```

```

    warn("or the directory is genuinely empty (unlikely on a real DC).")
else:
    ok("Counts retrieved successfully")

# ----- Step 6: Single-user JSON sample -----
step(6, "Single-user JSON sample (this is what the main script parses)")
sample = r"""
$ErrorActionPreference = 'Stop'
Import-Module ActiveDirectory
$one = Get-ADUser -Filter * -ResultSetSize 1 -Properties DisplayName, Enabled, LastLogonDate |
    Select-Object SamAccountName, DisplayName, Enabled,
        @{N='LastLogonDate';E={if($_.LastLogonDate){$.LastLogonDate.ToString('o')}}}
,@($one) | ConvertTo-Json -Depth 4 -Compress
"""

r = session.run_ps(sample)
out = r.std_out.decode(errors="replace")
err = r.std_err.decode(errors="replace").strip()
print(f"    exit_code = {r.status_code}")
print(f"    stdout (first 500 chars):")
print(f"    {out[:500]!r}")
if err:
    print(f"    stderr    = {err!r}")
if r.status_code == 0 and out.strip():
    ok("Sample JSON received – the main script's parsing should work.")
else:
    bad("No JSON output. The ConvertTo-Json step is failing.")
    return 1

print(f"\n{GREEN}All six steps passed. The HTTPS path is fully working.{END}")
print("If the main enumeration script still returns nothing, please share")
print("its exact output – the issue is elsewhere.")
return 0

if __name__ == "__main__":
    sys.exit(main())

```

Step 1 — Tell me what you actually see

Could you tap which output pattern matches what you're seeing?

While you pick, here's a **quick diagnostic script** that isolates each layer so we can pinpoint exactly where things break. Run this before changing anything in the main script:

Created a file, read a file

Created a file, read a file

Run the diagnostic

bash

```
python winrm_https_diag.py --host YOUR_DC --user 'DOMAIN\admin'
```

It runs **six layered checks** and prints raw `exit_code`, `stdout`, and `stderr` at every step. The first one that fails tells us exactly where the problem is.

Most likely causes, ranked by what I see most often

While you run the diagnostic, here are the usual suspects for "HTTPS returns nothing":

1. HTTPS listener not configured on the server (most common) The Windows Server has WinRM enabled but only on port 5985. Verify on the server:

powershell

```
winrm enumerate winrm/config/Listener
```

You should see two entries — one with `Transport = HTTP` and one with `Transport = HTTPS`. If only HTTP shows up, follow Part 2 of my earlier guide to create the HTTPS listener.

2. Firewall blocking 5986 specifically Default Windows firewall rules cover 5985 but **not 5986**. Check on the server:

powershell

```
Get-NetFirewallRule -DisplayName "*WinRM*HTTPS*"
```

If nothing comes back, add it:

powershell

```
New-NetFirewallRule -DisplayName "WinRM HTTPS-In" -Direction Inbound `
  -LocalPort 5986 -Protocol TCP -Action Allow
```

3. PowerShell `ConvertTo-Json` silently choking On some PowerShell versions, `ConvertTo-Json -Depth 4` truncates objects with deeper nesting and emits **nothing** instead of an error. The diagnostic's Step 6 will catch this — if `stdout` is empty there, that's the issue.

4. Credentials work for auth but the account can't read AD A standard user account can authenticate to WinRM but may not have permission to read user/computer attributes. Step 5 of the diagnostic prints the raw counts — if both are 0 against a real DC, this is it.

5. The user is logged in as a local account, not a domain account `--user admin` is treated as a local account. For AD queries you need a domain account: `--user 'CORP\admin'` or `--user admin@corp.local`.

6. Network proxy or HTTPS inspection appliance A corporate proxy doing TLS inspection will break NTLM message integrity. The diagnostic's Step 3 will fail with a cryptic error if this is happening — usually `HTTPSConnectionPool` or `BadStatusLine`.

Once you've either picked an option above or run the diagnostic, paste me what you see and I'll narrow it down further.

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 <code>AllowUnencrypted=true</code> ?
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."