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.

Revision #1
Created 19 May 2026 15:55:54 by Admin
Updated 19 May 2026 15:59:19 by Admin