

Blue Team Tasks and Cookbook

As a member of the blue team, your primary goal is to protect your organization's systems and networks from cyber threats. However, this is no easy task. The threat landscape is constantly evolving, and you may be faced with challenges such as managing and analyzing large amounts of data, coordinating with other teams, and ensuring compliance with regulations.

In this chapter, we'll first take a closer look at the *protect, detect, and respond* approach and some of the challenges that blue teamers face. Next, we will explore an overview of some useful open source tools written in PowerShell that can help you in your daily practice as a blue teamer. Finally, we will look at the blue team cookbook, a collection of PowerShell snippets that can come in handy in your daily work as a blue team practitioner.

In this chapter, we will discuss the following topics:

- Understanding the protect, detect, and respond approach
- Common PowerShell blue team tools
- The blue 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
- Access to the GitHub repository for this chapter:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/tree/master/Chapter09>

Protect, detect, and respond

Being a blue teamer is not an easy thing to do. You need to constantly keep up with the evolving threat landscape and stay up to date. While a red teamer needs to find just one single vulnerability to be successful, a blue teamer needs to watch for everything, as one little error already means that your network could be compromised.

Blue teamers not only need to configure and manage their systems but also analyze large amounts of data and coordinate with other teams. They need to ensure compliance with regulations and standards. And while they do all that, they need to keep the right balance between secu-

rity and usability, ensuring that their users don't get overwhelmed with all the security measures and try to bypass them by themselves.

To help keep track of everything that needs to be taken into account, categorizing tasks into **protect**, **detect**, and **respond** types can help. This is an approach to secure your organization's systems, as well as its network. It is structured into three different areas – protection, detection, and response. Every pillar is of equal importance to keep your infrastructure safe.

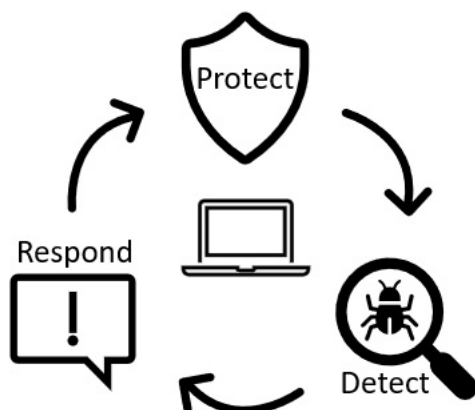


Figure 9.1 – The protect, detect, and respond approach

Many companies just focus on the protection part, although detection and response are also very important to keep adversaries out of your network.

Let's explore what each area covers in the following subsections.

Protection

The goal of **protection** measures is to mitigate security risks and implement controls to reduce and block threats *before they happen*. Protection measures could include the following:

- Regularly updating systems and monitoring them to fix vulnerabilities that could be exploited by attackers.
- Implementing user authentication and authorization to ensure that only authorized users have access to data and systems. The least privilege approach also needs to be followed.
- Encrypting sensitive data to minimize the risk of it being accessed by unauthorized users. Encrypt hard drives to avoid credential theft from a person that has physical access or even data theft if a device was stolen.
- Implementing security policies, baselines, and access control to ensure that systems are configured as safely as possible. A strong password policy also needs to be introduced.
- Deploying firewalls and **intrusion detection systems (IDSs)/intrusion prevention systems (IPSs)** to block unauthorized activities and detect suspicious activities.

Of course, protection mechanisms could also have a second purpose, such as an IDS or IPS, which not only blocks suspicious activities but also de-

fects and alerts you to them. Therefore, this solution could also be a part of the **detection** area.

Detection

In the detection phase, the goal is to identify and report potential security threats as quickly as possible. There are various things that you can do to improve your detection stance, such as the following:

- Collecting and analyzing event logs about potential security breaches, such as failed login attempts or configuration changes.
- Monitoring network activity for anomalies and suspicious behavior, such as users that are logging in to machines that they usually never log in to or attempts to access restricted resources. Another example would be if PowerShell (or other) code was executed from a workstation of a person that usually never runs code, such as employees from accounting or marketing.
- Evaluating security alerts from antivirus software and IDSs/IPSSs.
- Regularly scanning your network for vulnerabilities to identify potential weaknesses that could be abused by adversaries. Also, periodically hire external penetration testers to check your security.

Implementing good detection measures will help you raise your awareness of what happens in your network. This allows you to act on potential security threats in the response phase.

Response

If a security threat was detected, it means that you need to act on it quickly to reduce the risk and restore systems to a secure state. This can involve a variety of activities, such as the following:

- Isolating compromised systems to prevent further damage and the threat spreading within an environment.
- Gathering forensic data from affected systems and analyzing it. This helps to identify the attack source and determine the extent of the damage. It can also help to mitigate future threats.
- Restoring systems to a secure state, which may involve repairing or re-installing them in accordance with the NIST **Cybersecurity Framework** (NIST CSF) guidelines:

<https://www.nist.gov/cyberframework/framework>

- Implementing additional security controls to prevent similar threats in the future.

All three pillars combined build the protect, detect, and respond life cycle and should always be focused on with equal importance.

There are also many open source tools that can support a blue teamer to pursue the protect, detect, and respond approach. In the next section, we will explore some of them.

Common PowerShell blue team tools

As a blue teamer, you are constantly on the lookout for tools and techniques that can help you protect your organization's systems and networks from cyber threats.

In this section, we'll explore some common PowerShell open source tools that can be particularly helpful for blue teamers. These tools can assist with tasks such as analyzing system logs, gathering system information, and detecting malicious activity. Some of the tools can also help with tasks such as analyzing the attack surface of a system, identifying and decoding potentially malicious data, and searching for indicators of compromise. By leveraging these tools, you can streamline your workflows and more effectively defend your organization against cyber threats.

PSGumshoe

PSGumshoe is a powerful PowerShell module that is designed to assist with tasks such as live response, hunt, and forensics. Developed by Carlos Perez, this open source tool is designed to help blue teamers collect artifacts from a variety of sources. Whether you are investigating a security incident, conducting a hunt for indicators of compromise, or performing forensic analysis, PSGumshoe can be a valuable asset in your toolkit. It also has functions included to support retrieving data from Sysmon-generated events or to track **Windows Management Instrumentation (WMI)** activity.

You can install PSGumshoe from PowerShell Gallery using the **Install-Module PSGumshoe** command or download it from GitHub:

<https://github.com/PSGumshoe/PSGumshoe>.

PowerShellArsenal

PowerShellArsenal is a PowerShell module developed by Matt Graeber that is designed to assist reverse engineers in a variety of tasks. With its wide range of features and capabilities, this tool can help you disassemble code, perform .NET malware analysis, analyze and parse memory structures, and much more. Whether you are a seasoned reverse engineer or just starting out, PowerShellArsenal can be a valuable addition to your toolkit.

It can be downloaded and installed as a module from GitHub:

<https://github.com/mattifestation/PowerShellArsenal>.

AtomicTestHarnesses

AtomicTestHarnesses is a PowerShell module that allows you to simulate and validate the execution of attack techniques. With a PowerShell component for Windows and a Python component for macOS and Linux, this tool can be used across platforms.

Developed by Mike Haag, Jesse Brown, Matt Graeber, Jonathan Johnson, and Jared Atkinson, AtomicTestHarnesses is a valuable resource for blue teamers who are looking to test their defenses and ensure that they are prepared to respond to real-world attacks.

You can easily install AtomicTestHarnesses from the PowerShell gallery using the **Install-Module -Name AtomicTestHarnesses** command, or you can download it from GitHub at the following link:

<https://github.com/redcanaryco/AtomicTestHarnesses>.

PowerForensics

PowerForensics is a powerful framework for hard drive forensics developed by Jared Atkinson. Currently supporting NTFS (**New Technology File System**) and FAT (**File Allocation Table**) file systems, this tool is designed to assist with tasks such as analyzing Windows artifacts, the Windows registry, boot sector, and application compatibility cache, as well as creating a forensic timeline.

With its extensive range of features and capabilities, PowerForensics is an invaluable resource for blue teamers who need to conduct forensic analysis on hard drives. You can easily install PowerForensics from the PowerShell gallery using the **Install-Module PowerForensics** command, or you can download it from GitHub at the following link:

<https://github.com/Invoke-IR/PowerForensics>.

NtObjectManager

NtObjectManager is an extensive PowerShell module that allows you to access the NT Object Manager namespace. It is part of the sandbox attack surface analysis tools toolkit (which is also definitely worth a look!) that was developed by James Forshaw. The Object Manager itself is a subsystem within Windows that is responsible for managing the system's objects, which represent various system resources such as processes, threads, files, and devices.

The Object Manager is also in charge of creating and deleting objects, as well as maintaining the relationships between objects. It also handles object access requests, ensuring that only authorized entities are able to access specific objects. The Object Manager is an integral part of the operating system and is involved in many aspects of system operation, including memory management, process and thread management, and I/O operations.

The NtObjectManager module offers a wide variety of capabilities, including working with symbolic links, auditing RPC servers, manipulating the Object Manager, and generally messing around with the Windows operating system.

NtObjectManager can be easily installed using the **Install-Module -Name NtObjectManager** command, and the source code can be found on GitHub at the following link:

<https://github.com/googleprojectzero/sandbox-attacksurface-analysis-tools>.

DSInternals

DSInternals is a powerful Active Directory suite developed by Michael Grafnetter that consists of two parts – a framework that exposes various internal components of Active Directory that can be accessed from any .NET application, and a PowerShell module that provides a range of cmdlets built on top of the framework. The module offers extensive functionality, including the ability to audit Azure AD FIDO2 keys, AD passwords, and key credentials, and perform bare-metal recovery of domain controllers.

DSInternals can be easily installed using the **Install-Module DSInternals** command, or you can download it from GitHub at the following link: <https://github.com/MichaelGrafnetter/DSInternals>.

With its many features and capabilities, DSInternals is a valuable resource for blue teamers who need to manage and secure their Active Directory environment.

PSScriptAnalyzer and InjectionHunter

PSScriptAnalyzer is a tool that helps you improve the quality and security of your PowerShell scripts and modules. It checks your code against predefined rules and provides recommendations for any potential defects it finds. You can install PSScriptAnalyzer using the **Install-Module PSScriptAnalyzer** command, or you can download it from GitHub at the following link: <https://github.com/PowerShell/PSScriptAnalyzer>.

InjectionHunter is a module developed by Lee Holmes that helps you detect potential opportunities for code injection in your own PowerShell scripts. To use InjectionHunter, you need to have PSScriptAnalyzer installed, as it relies on the **ScriptAnalyzer.Generic.DiagnosticRecord** output type and uses custom detection rules. You can install InjectionHunter using the **Install-Module InjectionHunter** command, or you can find it in the PowerShell Gallery at the following link: <https://www.powershellgallery.com/packages/InjectionHunter/1.0.0>.

Also refer to the official blog post on InjectionHunter:

<https://devblogs.microsoft.com/powershell/powershell-injection-hunter-security-auditing-for-powershell-scripts/>.

Later in [Chapter 13](#), *What Else? – Further Mitigations and Resources*, we will also take a closer look at both tools and how they can be used.

Revoke-Obfuscation

Revoke-Obfuscation is a PowerShell obfuscation detection framework developed by Daniel Bohannon and Lee Holmes. Compatible with PowerShell v3 and later, this tool helps blue teamers detect obfuscated PowerShell scripts and commands at scale. Unlike other solutions that

rely on simple **indicators of compromise (IOCs)** or regular expression matching, Revoke-Obfuscation uses PowerShell's **abstract syntax tree (AST)** to extract features from a script, making it more robust in detecting even unknown obfuscation techniques.

You can easily install Revoke-Obfuscation using the **Install-Module Revoke-Obfuscation** command, or you can download it from GitHub at the following link: <https://github.com/danielbohannon/Revoke-Obfuscation>.

Posh-VirusTotal

As a defender, it's critical to regularly check files, domains, IPs, and URLs for malware. One popular service to do this is **VirusTotal** (<https://www.virustotal.com>), which allows you to quickly check whether a file hash or URL is considered malicious and whether it would be detected by one or more security vendors. However, manually uploading each file or checking URLs one by one can be time-consuming and tedious.

That's where the PowerShell module **Posh-VirusTotal** comes in. Developed by Carlos Perez, this tool enables you to automate your VirusTotal submissions and save time in your busy schedule. It's compatible with PowerShell v3 and higher and can use either the public or private version 2 API provided by VirusTotal.

You can easily install Posh-VirusTotal using the **Install-Module Posh-VirusTotal** command, or you can download it from GitHub at the following link: <https://github.com/darkoperator/Posh-VirusTotal>.

If you're using an older version of PowerShell, such as v3, you can also install Posh-VirusTotal using the **iex (New-Object Net.WebClient).DownloadString("https://gist.githubusercontent.com/darkoperator/9138373/raw/22fb97c07a:** command.

With Posh-VirusTotal, you can streamline your malware checks and stay one step ahead of threats.

EventList

EventList is a useful tool that I developed to help you improve your audit capabilities and build a more effective **security operations center (SOC)**. Developed to combine Microsoft security baselines with MITRE ATT&CK, EventList enables you to generate hunting queries for your SIEM system, regardless of the product you use.

By leveraging the power of EventList, you can take a proactive approach to detecting and responding to security threats.

It can be installed using the **Install-Module EventList** command or downloaded from GitHub: <https://github.com/miriamxyra/EventList>.

JEAnalyzer

Just Enough Administration (JEA) is a powerful tool to secure the PowerShell commands that administrators and users are allowed to use in your environment. However, configuring and auditing JEA roles can be a tedious and time-consuming task. That's where JEAnalyzer comes in.

Developed by Miriam Wiesner and Friedrich Weinmann, this tool simplifies the implementation and management of JEA, as well as providing tools to scan commands for potential danger when exposed in a JEA endpoint and creating JEA endpoints simply and conveniently.

You can easily install JEAnalyzer using the **Install-Module JEAnalyzer** command, or you can download it from GitHub at the following link: <https://github.com/PSSecTools/JEAnalyzer>.

All these PowerShell modules come in very handy for blue teamers, as they can assist in tasks such as live response, hunt, forensics, and reverse engineering. These tools can help streamline workflows and defend against cyber threats by analyzing system logs, gathering system information, detecting malicious activity, analyzing attack surfaces, identifying and decoding potentially malicious data, searching for indicators of compromise, and many more use cases.

Blue team cookbook

In the following subsections, you will find some code snippets that come in handy for your daily life as a blue team PowerShell practitioner. Blue teaming is quite extensive; therefore, you won't find use cases for every scenario but, rather, some of the basics.

Also, refer to ***Chapter 8, Red Team Tasks and Cookbook***, as you will find many red teamer code snippets and scripts there that can also sometimes be useful for a blue teamer.

Checking for installed updates

You want to find out which updates were installed on one or more remote systems.

Solution

You can use the **Get-InstalledUpdates.ps1** script to scan an IP range for installed Windows updates. You can find the script in the GitHub repository of this chapter: <https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter09/Get-InstalledUpdates.ps1>.

Use this example to scan the **172.29.0.10-20** IP range for installed updates:

```
> .\Get-InstalledUpdates.ps1 -BaseIP "172.29.0" -MinIP 10 -MaxIP 20 -Verbose
```

-MinIP represents the smallest last IP address octet, while **-MaxIP** represents the highest last IP address octet. Enabling the **-Verbose** parameter

allows the script to display a detailed output of its actions. It is also possible to use the **-MaxJobs** parameter to define how many jobs can be run in parallel to check updates.

Checking for missing updates

You want to find out which updates are missing on one or more remote host(s).

Solution

You can use the **Scan-RemoteUpdates.ps1** script to check for missing Windows updates – either on the localhost or on one or more remote host(s). You can find the script in the GitHub repository of this chapter:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter09/Scan-RemoteUpdates.ps1>.

Scanning only the localhost is done as follows:

```
> .\Scan-RemoteUpdates.ps1
```

Scanning for multiple remote hosts is done as follows:

```
> .\Scan-RemoteUpdates.ps1 -remoteHosts "PSSec-PC01", "PSSec-PC02", "PSSec-Srv01"
```

If the **-Force** parameter is specified, the **wsusscn2.cab** file will be deleted if present and a new version will be downloaded. Use the **-CabPath** parameter to specify where the **wsusscn2.cab** file should be downloaded. If nothing is specified, it will be downloaded to **\$env:temp\wsusscn2.cab**. If **-DoNotDeleteCabFile** is present, the **wsusscn2.cab** file will not be deleted after the check.

Reviewing the PowerShell history of all users

During an incident response, you want to review the PowerShell history of all users on a system.

Solution

The **Get-History** cmdlet would only get the current shell's history, which is not very helpful. To review the entire PowerShell history of each user, you can loop through the **ConsoleHost_history.txt** files on a system:

```
$UserHistory = @(Get-ChildItem "C:\Users\*\AppData\Roaming\Microsoft\Windows\PowerShell\PSReadline\Conso
```

```
$UserHistory += @(Get-ChildItem "c:\windows\system32\config\systemprofile\appdata\roaming\microsoft\wind
```

```

foreach ($Item in $UserHistory) {

    if ($Item) {

        Write-Output ""

        Write-Output "#####"

        Write-Output "PowerShell history: $item"

        Write-Output "#####"

        Get-Content $Item

    }

}

```

In this example, you would loop through all the **ConsoleHost_history.txt** files of all users, as well as through the system profile (if available).

Inspecting the event log of a remote host

You want to inspect the event log of a remote host and search for specific patterns.

Solution

You can use **Get-WinEvent** to get all events on a (remote) host and filter for specific patterns. Please note that the RemoteRegistry service needs to run on the remote host in order for the **Get-WinEvent** cmdlet to work remotely:

```

$ComputerName = "PSSec-PC01.PSSec.local"

$EventLog = "Microsoft-Windows-Powershell/Operational"

$LogEntries = Get-WinEvent -LogName $EventLog -ComputerName $ComputerName

$LogEntries | Where-Object Id -eq 4104 | Where-Object Message -like "*Mimikatz*"

```

Using this example, you would connect to the remote host, **PSSec-PC01.PSSec.local**, and retrieve all events in the **Microsoft-Windows-Powershell/Operational** event log and save them into the **\$LogEntries** variable. This allows you to quickly operate with the events by not always connecting remotely and, instead, operating the variable.

Using the **\$LogEntries** variable, you could filter for specific events or strings. In this example, we filter for events with the **4104** event ID that contain the **"Mimikatz"** string in the message body. The wildcards, *****, indicate that other characters could prefix or suffix the search term **"Mimikatz"**.

Please note that if you want to query the PowerShell Core log instead, you would need to change the **\$EventLog** variable to **"PowerShellCore/Operational"**.

PowerShell remoting versus the -ComputerName parameter

It's worth mentioning that PowerShell remoting can be used to remotely execute any cmdlet, regardless of whether the cmdlet has a **-ComputerName** parameter or not. This can be particularly useful in cases where the **-ComputerName** parameter does not work, due to closed DCOM ports or other reasons. As an example, to retrieve log entries from a remote computer, you can use the following command: – **Invoke-Command -ComputerName \$ComputerName -ScriptBlock { Get-WinEvent -LogName \$EventLog | Where-Object Id -eq 4104 | Where-Object Message -like "Mimikatz" }**.

You could also assess multiple remote hosts by looping them through using **foreach**, as shown in the following example:

```
$ComputerNames = @("DC01", "PSSec-PC01", "PSSec-PC02", "PSSec-Srv01")

$EventLog = "Microsoft-Windows-Powershell/Operational"

$LogEntries = foreach ($Computer in $ComputerNames) {

    Get-WinEvent -LogName $EventLog -ComputerName $Computer -ErrorAction SilentlyContinue

}

$LogEntries | Group-Object -Property MachineName

$LogEntries | Where-Object {($_.Id -eq 4104) -and ($_.Message -like "*Mimikatz*")} | Select-Object -Prop
```

You can assess the events collected using the **\$LogEntries** variable. To get an overview of how many events were collected from which hosts, you

can use **Group-Object** and group by **MachineName**.

Monitoring to bypass powershell.exe

You want to monitor for the execution of PowerShell without the use of the **powershell.exe** binary.

Solution

To monitor the execution of PowerShell without the use of the **powershell.exe** binary, there are two solutions. Option number one is to use the Windows PowerShell event log and look for the **400** event ID:

```
> Get-WinEvent -LogName "Windows PowerShell" | Where-Object Id -eq 400 | Where-Object Message -notmatch
```

Since there are multiple legitimate reasons to execute PowerShell without the **powershell.exe** binary, you might want to adjust this query to your environment. On a regular Windows 10 client system, on which the PowerShell ISE is also used, the following code snippet could be helpful:

```
> Get-WinEvent -LogName "Windows PowerShell" | Where-Object Id -eq 400 | Where-Object { ($_.Message -not
```

For option number two, you need to have Sysmon installed on all systems on which you want to detect the bypass of the **powershell.exe** binary. Sysmon is part of the Sysinternals suite and can be downloaded here: <https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon>.

Once Sysmon is installed and configured, you will need to look for the following DLLs using Sysmon's event ID 7, **"Image loaded"**:

- **System.Management.Automation.dll**
- **System.Management.Automation.ni.dll**

You can now search for potential bypasses of the **powershell.exe** binary, as shown in the following example:

```
$ComputerName = "PSSec-PC01.PSSec.local"
```

```
$EventLog = "Microsoft-Windows-Sysmon/Operational"
```

```
$LogEntries = Get-WinEvent -LogName $EventLog -ComputerName $ComputerName
```

```
$LogEntries | Where-Object Id -eq 7 | Where-Object { ($_.Message -like "*System.Management.Automation*")
```

If you have an EDR in place that helps you detect similar events, you don't need Sysmon to detect the PowerShell .NET assembly calls, of course.

Getting specific firewall rules

You want to filter specific firewall rules using PowerShell.

Solution

You can get all firewall rules and filter for specific ones using the **Get-NetFirewallRule** cmdlet:

```
> Get-NetFirewallRule -<parameter> <value>
```

There are many parameter filter options available using **Get-NetFirewallRule**. To get, for example, all enabled firewall rules that have the direction inbound and are allow rules, use the following command:

```
> Get-NetFirewallRule -Direction Inbound -Enabled True -Action Allow
```

You can also use the **Get-NetFirewallProfile** cmdlet, together with **Get-NetFirewallRule**, to retrieve all firewall rules that were created for a particular firewall profile. By using the following example, you would get all firewall rules that were created for the **Public** firewall profile:

```
> Get-NetFirewallProfile -Name Public | Get-NetFirewallRule
```

Allowing PowerShell communication only for private IP address ranges

You want to restrict PowerShell communication to happen only in your own network and avoid PowerShell communicating to potential C2 servers.

Solution

Create a new firewall rule using **New-NetFirewallRule** to lock down PowerShell communication to private IP address ranges only.

The following example creates a new firewall rule, with the name **Block Outbound PowerShell connections**, that restricts Windows PowerShell from establishing connections with IP addresses outside of the local network:

```
> New-NetFirewallRule -DisplayName "Block Outbound PowerShell connections" -Enabled True -Direction Outb
```

Use this example and adjust it to your needs. As most organizations still use Windows PowerShell as their default PowerShell instance, this example also refers to Windows PowerShell. If you are using PowerShell Core as your default PowerShell instance, you might want to adjust the path to the program.

Isolating a compromised system

You want to isolate a compromised system.

Solution

You can do this by using the **New-NetFirewallRule** and **Disable-NetAdapter** cmdlets. The following code snippet demonstrates how you can remotely isolate a device. First, it sends a message to all users that are currently logged on **PSSec-PC01**, then it remotely creates firewall rules to block all inbound and outbound connections, and then disables all network adapters:

```
$ComputerName = "PSSec-PC01"

msg * /server $ComputerName "Security issues were found on your computer. You are now disconnected from

$session = Invoke-Command -ComputerName $ComputerName -InDisconnectedSession -ScriptBlock {

    New-NetFirewallRule -DisplayName "Isolate from outbound traffic" -Direction Outbound -Action Block |

    New-NetFirewallRule -DisplayName "Isolate from inbound traffic" -Direction Inbound -Action Block | 0

    Get-NetAdapter|foreach { Disable-NetAdapter -Name $_.Name -Confirm:$false }

}

Remove-PSSession -Id $session.Id -ErrorAction SilentlyContinue
```

Just replace **PSSec-PC01** with the computer name of your choice, and feel free to adjust the message that will be sent to the computer users.

Checking out installed software remotely

You want to find out what software is installed on a remote PC.

Solution

You can check out what software is installed on a remote PC by using the **Get-CimInstance** cmdlet.

The following example code will let you connect to a computer named **PSSec-PC01** and find out which software it currently has installed:

```
$ComputerName = "PSSec-PC01"

Get-CimInstance -ClassName Win32_Product -ComputerName $ComputerName | Sort-Object Name
```

Starting a transcript

You want to enable an over-the-shoulder transcription to track what is happening in a PowerShell session.

Solution

Enable a transcript on the machine on which you want to track what is happening in a PowerShell session. This can be done by either enabling the transcript via Group Policy by configuring the **Turn on PowerShell Transcription** option under **Windows Components | Administrative Templates | Windows PowerShell**, or by configuring it using PowerShell to configure the registry, as shown in the blog article *PowerShell ♥ the Blue Team*: <https://devblogs.microsoft.com/powershell/powershell-the-blue-team/>

The following code snippet shows the **Enable-PSTranscription** function, which originates from this article:

```
function Enable-PSTranscription {  
  
    [CmdletBinding()]  
  
    param(  
  
        $OutputDirectory,  
  
        [Switch] $IncludeInvocationHeader  
  
    )  
  
    $basePath = "HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription"  
  
    if (-not (Test-Path $basePath)) {$null = New-Item $basePath -Force}  
  
    Set-ItemProperty $basePath -Name EnableTranscripting -Value 1  
  
    if ($PSCmdlet.MyInvocation.BoundParameters.ContainsKey("OutputDirectory")) {Set-ItemProperty $basePa  
  
    if ($IncludeInvocationHeader) {Set-ItemProperty $basePath -Name IncludeInvocationHeader -Value 1}  
  
}
```

If you used this function to enable transcription to the **C:\tmp** folder, the syntax would look like this:

```
> Enable-PSTranscription -OutputDirectory "C:\tmp\"
```

You can also use a **Universal Naming Convention (UNC)** path to save the transcript to a network folder. Make sure to secure the path so that a potential attacker cannot access and/or delete it.

To centralize PowerShell transcripts and maintain a secure audit trail, you can, for example, configure the transcript destination as a UNC path with a dynamic filename. This involves setting the transcript directory to a network share with write-only permission and using the PowerShell profile to log all activity to a file with a unique name, based on system and user variables, such as the following:

```
> Enable-PSTranscription -OutputDirectory "\\fileserver\Transcripts$\env:computername-$($env:userdomain
```

This will create a unique transcript file for each user and computer combination, with the current date and time included in the filename. By storing transcripts in a centralized location with restricted access, you can ensure that all activity is logged and available for review and analysis as needed.

This will write all transcripts to the specified file server location, which can then be accessed by authorized personnel for review and analysis.

Checking for expired certificates

You want to check for SSL certificates in your certificate store that have already expired or will expire in the next 60 days.

Solution

You can use the following script to check for SSL certificates in your certificate store that have already expired or will expire in the next 60 days:

```
$certificates = Get-ChildItem -Path "Cert:\" -Recurse | Where-Object { $_.Subject -like "*CN=*" } | Where-Object {
    ($_.ExpirationDate -lt (Get-Date).AddDays(60)) -or ($_.ExpirationDate -lt (Get-Date).AddDays(60))
}

$expiringCertificates = @()

foreach ($certificate in $certificates) {
    if (($certificate.NotAfter) -and (($certificate.NotAfter -lt (Get-Date).AddDays(60)) -or ($certificate.NotAfter -lt (Get-Date).AddDays(60)))) {
        $expiringCertificates += $certificate
    }
}
```



```
}  
  
}  
  
Write-Output "Expired or Expiring Certificates in the next 60 days:"  
  
foreach ($expiringCertificate in $expiringCertificates) {  
  
    Write-Output $expiringCertificate | Select-Object Thumbprint, FriendlyName, Subject, NotBefore, NotA  
  
}
```

You can also alter the path to **Cert:\LocalMachine\My** to only assess certificates from the personal store. For certificates from the **root** store, change the path to **Cert:\LocalMachine\Root**.

Checking the digital signature of a file or a script

You want to check the authenticity and integrity of software or a script by checking the digital signature.

Solution

You can check the status of a digital signature by using the **Get-AuthenticodeSignature** cmdlet:

```
> Get-AuthenticodeSignature "C:\Windows\notepad.exe" | Format-List
```

Using **Get-AuthenticodeSignature**, you get all sorts of useful information about the digital signature, such as the certificate chain, which is demonstrated in the following screenshot:

```
PS C:\Users\PSec-Test> Get-AuthenticodeSignature "C:\Windows\notepad.exe" | Format-List
SignerCertificate : [Subject]
                  CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
                  [Issuer]
                  CN=Microsoft Windows Production PCA 2011, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
                  [Serial Number]
                  330000033B655FAEFAD675E9D60000000033B
                  [Not Before]
                  02/09/2021 20:23:41
                  [Not After]
                  01/09/2022 20:23:41
                  [Thumbprint]
                  8BD2C438000344F4398BDFE5ABAC3223357CD67F
TimeStamperCertificate : [Subject]
                        CN=Microsoft Time-Stamp Service, OU=Thales TSS ESM:FC41-4BD4-D220, OU=Microsoft Ireland Operations Limited, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
                        [Issuer]
                        CN=Microsoft Time-Stamp PCA 2010, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
                        [Serial Number]
                        330000018E59D84600A81094CC00010000018E
                        [Not Before]
                        20/10/2021 21:27:45
                        [Not After]
                        26/01/2023 20:27:45
                        [Thumbprint]
                        3D6228EA4F4E11EA8296B9C6C3E6898AE59588C
Status : Valid
StatusMessage : Signature verified.
Path : C:\Windows\notepad.exe
SignatureType : Catalog
IsOSBinary : True
```

Figure 9.2 – Query information about the digital signature of a file

However, if you prefer to query the status only, you can also use the **(Get-AuthenticodeSignature "C:\Windows\notepad.exe").Status** command.

Checking file permissions of files and folders

You want to enumerate the access rights of files and folders.

Solution

To enumerate the access rights of files and folders, you can use the **Get-ChildItem** and **Get-Acl** cmdlets. To enumerate, for example, all files and folders in the **Windows Defender** directory recursively, you can use the following code snippet:

```
$directory = "C:\Program Files\Windows Defender"

$Acls = Get-ChildItem -Path $directory -Recurse | ForEach-Object {

    $fileName = $_.FullName

    (Get-Acl $_.FullName).Access | ForEach-Object {

        [PSCustomObject]@{

            FileName = $fileName

            FileSystemRights = $_.FileSystemRights
```

```

        AccessControlType = $_.AccessControlType

        IdentityReference = $_.IdentityReference

        IsInherited = $_.IsInherited

    }

}

}

$Acls

```

If you want to enumerate on one level only, make sure to remove the **-Recurse** parameter.

Displaying all running services

You want to display all running services and their command paths.

Solution

Although you can use the **Get-Service** cmdlet to display all running services, you can also use **Get-CimInstance** to access the WMI information of the services and get even more information, such as the command path or **ProcessId**:

```
> Get-CimInstance win32_service | Where-Object State -eq "Running" | Select-Object ProcessId, Name, Disp
```

Stopping a service

You want to stop a service from running.

Solution

To stop a service from running, you can use the **Stop-Service** cmdlet. The following example shows you how to combine **Get-Service** with **Stop-Service** to stop the **maliciousService** service:

```
> Get-Service -Name "maliciousService" | Stop-Service -Force -Confirm:$false -verbose
```

Keep in mind that if you use the **-Confirm:\$false** parameter, the confirmation prompt will be bypassed, and the command will be executed without any further confirmation. It's recommended to use this parame-

ter with caution and only in situations where you are fully aware of the potential risks and consequences. It's important to thoroughly understand the implications of using this parameter and make an informed decision based on your specific use case.

Displaying all processes

You want to display all processes, including their owners and command lines.

Solution

You can display all processes and more information about them by using **Get-WmiObject win32_process**. To display all processes, including their owners and command lines, you can use the following code snippet:

```
> Get-WmiObject win32_process | Select ProcessID,Name,@{n='Owner';e={$_.GetOwner().User}},CommandLine |
```

Stopping a process

You want to stop a process.

Solution

To stop a process, you can use the **Stop-Process** cmdlet. To stop, for example, the process with **Id 8336**, you can use the following code snippet:

```
> Get-Process -Id 8336 | Stop-Process -Force -Confirm:$false -verbose
```

It is, of course, also possible to select a process by its name with the **-Name** parameter of the **Get-Process** cmdlet to stop it. If there is more than one process with the same name, it can happen that multiple processes will be stopped.

Keep in mind that if you use the **-Confirm:\$false** parameter, the confirmation prompt will be bypassed, and the command will be executed without any further confirmation. It's recommended to use this parameter with caution and only in situations where you are fully aware of the potential risks and consequences. It's important to thoroughly understand the implications of using this parameter and make an informed decision based on your specific use case.

Disabling a local account

You want to disable a local account.

Solution

To disable a local account, you can use the **Disable-LocalUser** cmdlet.

One way to improve security in Windows is to create a new user with administrative privileges and disable the default **Administrator** account.

This helps prevent brute-force attacks that often target the default account. To achieve this, you can use the **Disable-LocalUser** cmdlet.

Here's an example that demonstrates how to disable the **Administrator** account using the **Disable-LocalUser** cmdlet:

```
> Disable-LocalUser -Name "Administrator"
```

After running the command, you can use the **Get-LocalUser** cmdlet to verify that the account has been disabled:

```
> Get-LocalUser -Name "Administrator"
```

Enabling a local account

You want to enable a local account.

Solution

To enable a local account, you can use the **Enable-LocalUser** cmdlet. Using the following example, the **Administrator** account would be enabled:

```
> Enable-LocalUser -Name "Administrator"
```

Using the **Get-LocalUser** cmdlet, you can verify that the account was enabled:

```
> Get-LocalUser -Name "Administrator"
```

Disabling a domain account

You want to disable a domain account.

Solution

To disable a domain account, you can use the **Disable-ADAccount** cmdlet, which is part of the **ActiveDirectory** module. Using the following example, the **vvega** domain account would be disabled:

```
> Import-Module ActiveDirectory
```

```
> Disable-ADAccount -Identity "vvega"
```

Using the **Get-ADUser** cmdlet, you can verify that the account was disabled:

```
> (Get-ADUser -Identity vvega).enabled
```

Enabling a domain account

You want to enable a domain account.

Solution

To enable a domain account, you can use the **Enable-ADAccount** cmdlet, which is part of the **ActiveDirectory** module. Using the following example, the **vvega** domain account would be enabled:

```
> Import-Module ActiveDirectory

> Enable-ADAccount -Identity "vvega"
```

Using the **Get-ADUser** cmdlet, you can verify that the account was disabled:

```
> (Get-ADUser -Identity vvega).enabled
```

Retrieving all recently created domain users

You want to retrieve all domain users that were recently created.

Solution

To retrieve all users that were created in the last 30 days, you can use the following code snippet:

```
Import-Module ActiveDirectory

$timestamp = ((Get-Date).AddDays(-30)).Date

Get-ADUser -Filter {whenCreated -ge $timestamp} -Properties whenCreated | Sort-Object whenCreated -desce
```

Checking whether a specific port is open

You want to check whether a specific port on a remote system is open.

Solution

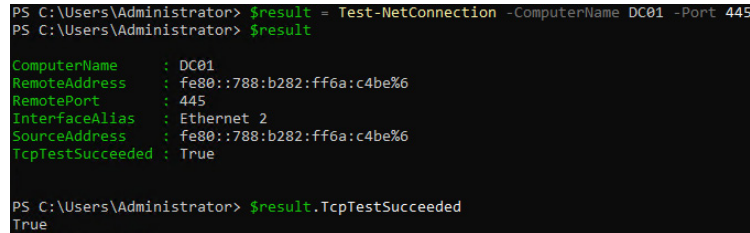
To find out whether a specific port is open, you can use the following code snippet; this example checks whether port **445** is open on the computer **DC01**:

```
$result = Test-NetConnection -ComputerName DC01 -Port 445
```

```
$result
```

```
$result.TcpTestSucceeded
```

The following screenshot shows the output of the preceding code snippet:



```
PS C:\Users\Administrator> $result = Test-NetConnection -ComputerName DC01 -Port 445
PS C:\Users\Administrator> $result
ComputerName      : DC01
RemoteAddress     : fe80::788:b282:ff6a:c4be%6
RemotePort        : 445
InterfaceAlias    : Ethernet 2
SourceAddress     : fe80::788:b282:ff6a:c4be%6
TcpTestSucceeded  : True

PS C:\Users\Administrator> $result.TcpTestSucceeded
True
```

Figure 9.3 – Checking whether port 445 is open on DC01

This method is a good way to test for a single port or for very few ports, as the **Test-NetConnection** cmdlet can be very time-consuming if used for a full port scan. Therefore, if you want to scan all ports of a remote system, you should instead use **nmap**.

Showing TCP connections and their initiating processes

You want to display all TCP connections, the initiating processes, as well as the command line that was used to open the TCP connection.

Solution

You can use **Get-NetTCPConnection** and create manual properties by using **Get-Process** and **Get-WmiObject** as **Select-Object** expressions:

```
> Get-NetTCPConnection | Select-Object LocalAddress,LocalPort,RemoteAddress,RemotePort,State,@{Label = 'ProcessName';Expression = {Get-Process ($_.RemoteAddress -ipaddress) | Select-Object Name,Path}}
```

This example shows all TCP connections, the local address and port, the remote address and port, the state of the connection, the name of the process, as well as the command line that was executed to initiate the connection.

Showing UDP connections and their initiating processes

You want to display all UDP connections, the initiating processes, as well as the command line that was used to open the UDP connection.

Solution

You can use **Get-NetUDPConnection** and create manual properties by using **Get-Process** and **Get-WmiObject** as **Select-Object** expressions:

```
> Get-NetUDPConnection | Select-Object CreationTime,LocalAddress,LocalPort,@{Label = 'ProcessName';Expression = {Get-Process ($_.RemoteAddress -ipaddress) | Select-Object Name,Path}}
```

This example shows all UDP connections, the creation time, the local address and port, the name of the process, as well as the command line that was executed to initiate the connection.

Searching for downgrade attacks using the Windows event log

You want to search for past downgrade attacks using the Windows event log.

Solution

You can search for past downgrade attacks using the Windows event log with the following code snippet, which was originally written by Lee Holmes:

```
Get-WinEvent -LogName "Windows PowerShell" | Where-Object Id -eq 400 | Foreach-Object {

    $version = [Version] ($_.Message -replace '(?s).*EngineVersion=([\d\.]+)*.*','$1')

    if($version -lt ([Version] "5.0")) { $_ }

}
```

Monitor for the **400** event ID in the Windows PowerShell event log. If **EngineVersion** is lower than **5**, you should definitely investigate further, as this could indicate a downgrade attack.

Preventing downgrade attacks

You want to prevent downgrade attacks from happening and, therefore, use **Windows Defender Application Control (WDAC)** to disable PowerShell version 2 binaries.

Solution

PowerShell version 2 cannot load if the **System.Management.Automation.dll** and **System.Management.Automation.ni.dll** assemblies are blocked, even if .NET Framework version 2 is installed and PowerShell version 2 is enabled.

Use the following code snippets to find out where those binaries are located to block them, using WDAC or another application control software of your choice:

```
> powershell -version 2 -nopprofile -command "(Get-Item ([PSObject].Assembly.Location)).VersionInfo"
```



```
> powershell -version 2 -noprofile -command "(Get-Item (Get-Process -id $pid -mo | ? { $_.FileName -matc
```

If you remove **-version 2** from the preceding code snippets, you will see that there are other binaries used for modern PowerShell versions. Therefore, you should not be afraid of breaking anything if your system relies on a modern PowerShell version and if you want to prohibit PowerShell version 2 binaries globally.

Now that you have located the PowerShell binaries, you can use WDAC to block these legacy versions. Make sure to block the native image as well as the **Microsoft intermediate language (MSIL)** assemblies.

Refer to Lee Holmes' blog post to learn more about detecting and preventing PowerShell downgrade attacks:

<https://www.leeholmes.com/detecting-and-preventing-powershell-downgrade-attacks/>.

Summary

This chapter first explored the *protect, detect, and respond* approach, emphasizing the importance of each pillar and its role in ensuring the security of an organization.

We then provided a comprehensive overview of commonly used PowerShell tools, which are essential for blue teamers to defend an organization against security threats.

Finally, the blue team cookbook, a collection of scripts and code snippets for security analysis and defense, was explored. The cookbook covers a wide range of tasks, including checking updates, monitoring bypasses, and analyzing event logs, processes, services, and network connections. The blue team cookbook serves as a valuable resource for information security practitioners, providing practical solutions to various security challenges.

Now that we've discussed daily blue team operations, let's explore further mitigation options that can help you secure your environment when using PowerShell. In the next chapter, we'll delve into language modes and **Just Enough Administration (JEA)**.

Further reading

If you want to explore some of the topics that were mentioned in this chapter, follow these resources:

- Blue Team Notes: <https://github.com/Purp1eW0lf/Blue-Team-Notes>
- Blue Team Tips: <https://sneakymonkey.net/blue-team-tips/>
- A collection of PowerShell functions and scripts a blue teamer might use: <https://github.com/tobor88/PowerShell-Blue-Team>
- Creating and Starting a Windows Service Remotely Using NtObjectManager Via Remote Procedure Calls (RPC) Over SMB:

https://blog.openthreatresearch.com/ntobjectmanager_rpc_smb_sc_m

- Detecting and Preventing PowerShell Downgrade Attacks:
<https://www.leeholmes.com/detecting-and-preventing-powershell-downgrade-attacks/>
- Directory Services Internals Blog: <https://www.dsinternals.com/en/>
- Investigating PowerShell Attacks:
<https://www.fireeye.com/content/dam/fireeye-www/global/en/solutions/pdfs/wp-lazanciyan-investigating-powershell-attacks.pdf>
- PowerForensics - PowerShell Digital Forensics:
<https://powerforensics.readthedocs.io/en/latest/>
- PowerShell ♥ the Blue Team:
<https://devblogs.microsoft.com/powershell/powershell-the-blue-team/>
- Testing adversary technique variations with AtomicTestHarnesses:
<https://redcanary.com/blog/introducing-atomictestharnesses/>
- Tracking WMI Activity with PSGumshoe:
<https://www.darkoperator.com/blog/2022/3/27/tracking-wmi-activity-with-psgumshoe>
- Windows Sandbox Attack Surface Analysis:
<https://googleprojectzero.blogspot.com/2015/11/windows-sandbox-attack-surface-analysis.html>

You can also find all links mentioned in this chapter in the GitHub repository for [Chapter 9](#) – there's no need to manually type in every link:

<https://github.com/PacktPublishing/PowerShell-Automation-and-Scripting-for-Cybersecurity/blob/master/Chapter09/Links.md>.