Network Defender Archeology

An NSM Case Study in Lateral Movement With DCOM

By: Alex Sirr, Justin Warner

Black Hat Europe 2018

I. Introduction

The Windows operating system is built upon layers of dormant yet powerful technology that has remained consistently enabled since early Windows NT 4.0.[1] The Component Object Model (COM) subsystem is one of those dormant technologies that exposes interfaces and functionality within software defined objects and has the ability to share this functionality over the network via the Distributed COM (DCOM) protocol.

Administrators and programmers have utilized the COM and DCOM subsystems for a myriad of tasks including but not limited to integration and automation. These subsystems present the same benefits to administrators as to adversaries who look to control systems in the network. In fact, adversaries have a lot to gain by leveraging legitimate functionality for malicious purposes, and often disguise their activity under the guise of "normal administrator behavior". Over the last several years, there has been a significant surge in the malicious use of Component Object Model (COM) objects in lateral movement using an approach colloquially known as "living off the land".

With over twenty years in existence and more than a year of relative popularity among adversaries, one would imagine that network analysis and detection of DCOM attacks is old news. On the contrary, very few people understand the techniques and tools fail to properly parse the network protocol, allowing adversaries to continue to successfully leverage it to further the compromise of networks. Needless to say, it is difficult to defend against techniques that the defenders don't understand or are unfamiliar with. This whitepaper is a natural byproduct of performing detection engineering work on DCOM. It will introduce the reader to the COM/DCOM systems from a network standpoint, explain how the systems are being abused by adversaries for lateral movement, and break down different techniques for detecting and analyzing behavior from a network security monitoring (NSM) standpoint.

[1] https://web.archive.org/web/20160304022635/http://www.microsoft.com/msj/0596/activex0596.aspx

II. COM/DCOM Overview

This section will present a high-level overview of both COM and DCOM. This is not meant to be comprehensive, but rather it aims to provide readers with background information and key concepts referenced in later parts of this paper.

A. COM

The Component Object Model (COM) is a language independent model that allows applications to expose objects with functionality for use in other applications at runtime. Through COM, callers only need to know about the functions and calling conventions of a particular object and have no need for implementation details. This makes COM components incredibly versatile and reusable as they reduce the burden on clients. There are three key concepts that make up a COM component:

- **COM Interface—**Defines the expected behavior of a component through a set of member functions. Interfaces are identified through a unique IID (Interface Identifier).

- **COM Class—**Implements one or more interfaces and contains the functionality of a component. Classes exist within DLLs or EXEs and are identified through a unique CLSID (Class Identifier).

- **COM Object—**An instantiation of a COM Class by a client.

Additionally, COM components are typically registered within the Windows Registry. The word typically is used, because it is possible to use components without the Windows Registry, but for the purposes of this paper, assume that components are registered normally.
All COM components are required to inherit the IUnknown interface which contains functions to manage an object's lifetime including reference counting and interface querying. Components can also inherit the IDispatch interface, which allows clients to query object information at runtime. An example of the behavior exposed by IDispatch can be seen while trying to tab-complete a COM object's function in Windows PowerShell or querying a function's documentation string. Finally, the IClassFactory interface is also notable as it handles the instantiation of a COM class and is required for components registered in the Registry.

B. Distributed COM (DCOM)

DCOM is an extension of COM that allows COM components to be created, managed, and called over a network in a distributed manner, thus Distributed COM. DCOM leverages DCE/RPC as a transportation protocol which uses a client/server model, whereby clients can request that objects be instantiated and executed on a remote server. Clients can then interact with objects over the network in the same way that they could interact with local COM components. Like any other client/server model, all client requests are executed on the server and the output is returned to the client—this fact will be very important in an adversary's use of DCOM. DCE/RPC's 'Bind' and 'Alter_Context' commands are used to attach to specific COM interfaces using an Interface Identifier (IID). After an interface is bound, messages will be passed using ORPC THIS/THAT structures as seen in the figure below. The next section will break down DCOM's network traffic in more detail.

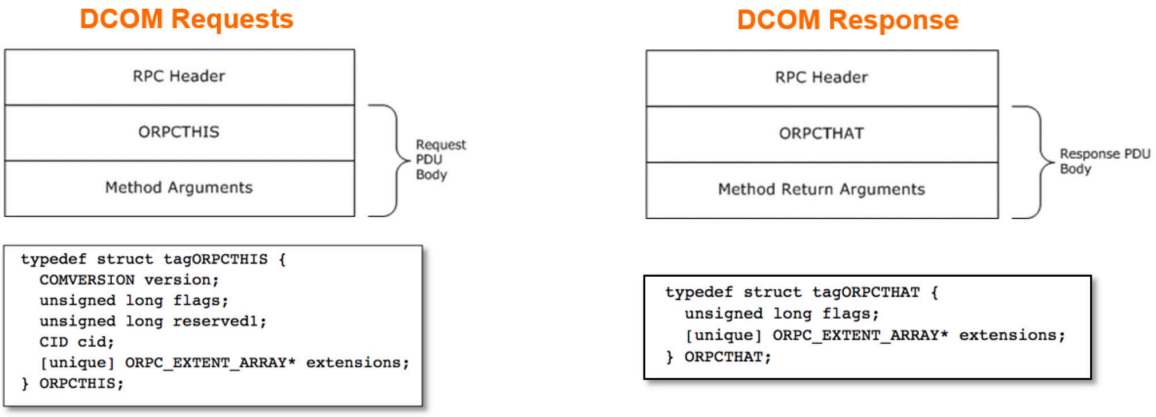


Figure 1: ORPC THIS/THAT structures.

III. Intelligence Analysis - Adversary Use of DCOM

Intelligence analysis is the process in which detection engineers consider how a given technology is abused by threat actors and what that means in context of threat detection. Performing this process enables a detection engineer to get a basic understanding of what they aim to detect, how to conduct threat simulation, and the impact of environmental context on the technique. This section identifies how DCOM is abused by threat actors and documents the rising popularity of the technique.

A. DCOM For Lateral Movement

MITRE maintains a framework known as ATT&CK, a "globally-accessible knowledge base of adversary tactics and techniques based on real-world observations".[2] It allows threat researchers to map generic techniques to an adversary attack chain. DCOM has been identified in ATT&CK as technique #T1175 and allows "adversaries operating in the context of an appropriately privileged user [to] remotely obtain arbitrary and even direct shellcode execution through Office applications as well as other Windows objects that contain insecure methods".[3] In short, DCOM enables an attacker to perform lateral movement, the process of progressively gaining access to additional hosts in a network with the goal of "moving" toward an objective. Although some tools and techniques leverage software vulnerabilities to achieve lateral movement, the techniques described throughout this paper are examples of how by-design functionality can be used in ways that were not intended.

B. Origins and Rise of DCOM Lateral Movement

In 2017, a security researcher, Matt Nelson, released a blog publicly demonstrating how an attacker could leverage a native COM object inside of Microsoft Windows ('MMC20.Application') to their advantage, essentially spawning arbitrary commands.[4,5] The release sparked a wave of public research in a similar fashion. Throughout 2017, researchers continued to advance the technique introducing additional COM objects including many from Microsoft Office that allowed not only shell command execution, but also enabled a threat actor to execute arbitrary VBScript or JScript.[6] The research was not only detailed and explained, but a weaponized POC capability was released in the form of a plugin for a popular penetration testing toolkit[7].

[2] https://attack.mitre.org/
[3] https://attack.mitre.org/techniques/T1175/
[4] https://twitter.com/enigma0x3
[5] https://enigma0x3.net/2017/01/05/lateral-movement-using-the-mmc20-application-com-object/
[6] https://enigma0x3.net/2017/09/11/lateral-movement-using-excel-application-and-dcom/
[7] https://blog.cobaltstrike.com/2017/01/24/scripting-matt-nelsons-mmc20-application-lateral-movement-technique/

DCOM is attractive to an adversary because it provides a native capability to execute code on remote hosts in a variety of adaptable manners. Additionally, DCOM is a decades old Windows subsystem that few people have recent in-depth knowledge of leading to a general lack of public awareness.

While there is little public threat reporting showing the use of DCOM as a lateral movement technique, it is well known that several notable threat groups follow and utilize techniques released by the offensive security research community.[8]

IV. Network Behavior Analysis

Behavior analysis is the process by which a detection engineer reviews data captured during threat simulation with the purpose of fully identifying patterns in underlying behavior and associated artifacts applicable as potential detection opportunities. This section will share the methodology of the threat simulation testing and detail the results of the behavior analysis.

A. Threat Simulation

This section includes a detailed walkthrough of the threat simulation testing conducted for detection purposes. Tests were performed across three different COM objects, three different weaponization vectors, and multiple versions of the Windows operating system.

Testing was performed in a controlled lab environment composed of three workstations and a domain controller.

[8] https://www.fireeye.com/blog/threat-research/2018/03/iranian-threat-group-updates-ttps-in-spear-phishing-campaign.html

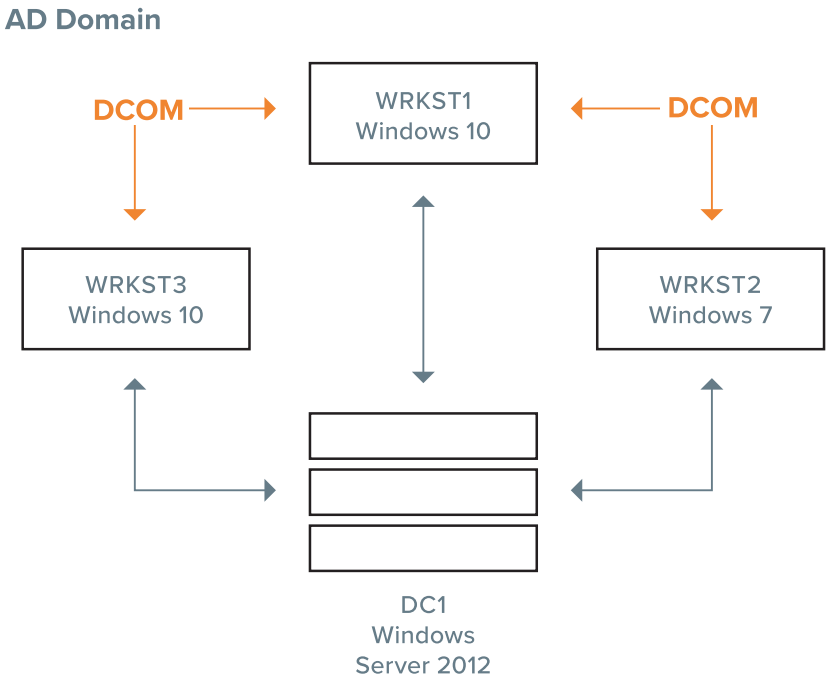


Figure 2: Diagram of the environment.

Each controlled scenario began by simulating that the threat actor had already escalated privileges and isolated the simulation to focus on lateral movement via DCOM. In each scenario, a weaponization vector (PowerShell, Python, Native Code) was utilized to gain control over a remote host using a specific COM object variation. After gaining remote control, testing would then terminate. PCAP was collected from the network during testing to allow for behavior analysis.

B. Analysis Results

For the purpose of this behavior analysis and examples, a specific testing scenario was used to generate screenshots and detail the results. There were slight variations in testing scenarios, but for the purpose of the generic behavior walkthrough, the following scenario was used:

Operating System	Windows 10 Pro - 1709
PowerShell Version	5.1
CLR Version	4.0
Object Details	ProgID: MMC20.Application CLSID: 49b2791a-b1ae-4c90-9b8e-e860ba07f889
Commands Run	> $com = [Type]::GetTypeFromProgID("MMC20.Application", $hostname) > $obj = [Activator]::CreateInstance($com) > $obj.Document.ActiveView.ExecuteShellCommand("C:\Windows\System32\calc.exe", $null, $null, 7)

At a high-level, there are three stages to DCOMs network behavior, which are as follows:

1. **Activation—**The requested COM object is instantiated on the remote host.

2. **Optional "Type" Operations—**Object information is resolved such as function names, parameter details, and documentation strings.

3. **Execution—**Selected function is executed.

Below, each stage is broken down and the relevant packet level content is provided.

ACTIVATION.

DCOM's activation phase prepares and instantiates the requested COM object on the remote server. To accomplish this, the client must bind with the remote RPC server through OXID Operations.[9] After successfully binding, the server will then authenticate and check the client's authorization. Once the client has been properly authenticated and authorized, the process of instantiating the remote object can begin. During this research, two methods of instantiating an object were identified. The first method is to gain a reference to the object's class factory via ISystemActivator's 'RemoteGetClassObject' method. After

[9] https://msdn.microsoft.com/en-us/library/cc226901.aspx

obtaining the reference, IClassFactory's 'CreateInstance' method will instantiate the object. The second observed method is to skip using a class factory and use ISystemActivator's 'RemoteCreateInstance' to instantiate the object directly. Both methods use the CLSID to identify which object the server should be creating. Figures 3 and 4 show the CLSID of 'MMC20.Application' as seen on the wire using both previously stated methods of object instantiation.

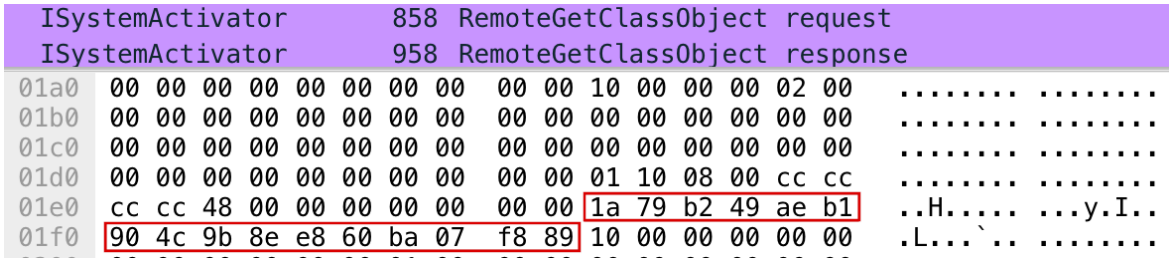


Figure 3: The 'MMC20.Application' CLSID within 'RemoteGetClassObject'

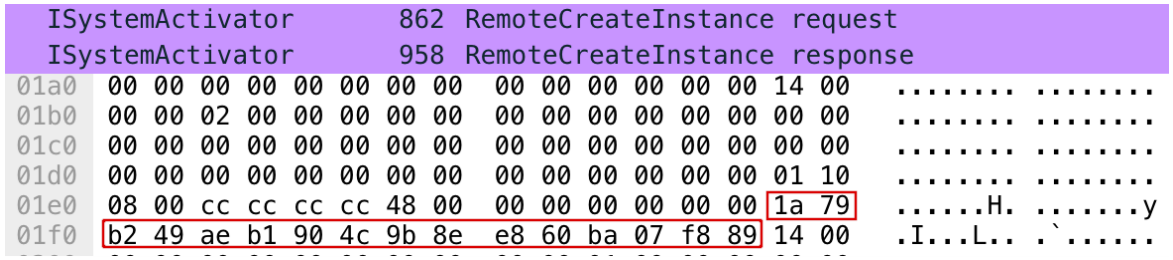


Figure 4: The 'MMC20.Application' CLSID within 'RemoteCreateInstance'

OPTIONAL "TYPE" OPERATIONS.

Following the object instantiation phase, the DCOM client may attempt to resolve information about the object via the ITypeInfo interface. This phase will only take place if the instantiated object inherits the IDispatch interface. As mentioned in the overview of COM, the IDispatch interface exposes functionality that allows clients to determine object information at runtime. Without this interface, clients such as PowerShell will be unable to determine what functions an object exposes and how to call them.

The first ITypeInfo operation is 'GetFuncDesc' (Opnum 5). This function is used to get information about a member function exposed by the requested object. This information includes details such as how many parameters the function accepts, the type of each parameter, and the return type of the function. When calling 'GetFuncDesc', a client will provide a function index / ordinal to indicate on which function it would

like information. Along with the function details, the server will also provide the client with the requested function's MemberID, which the client will use for future requests. Figures 5 and 6 show the request and response of 'GetFuncDesc' on the wire respectively.

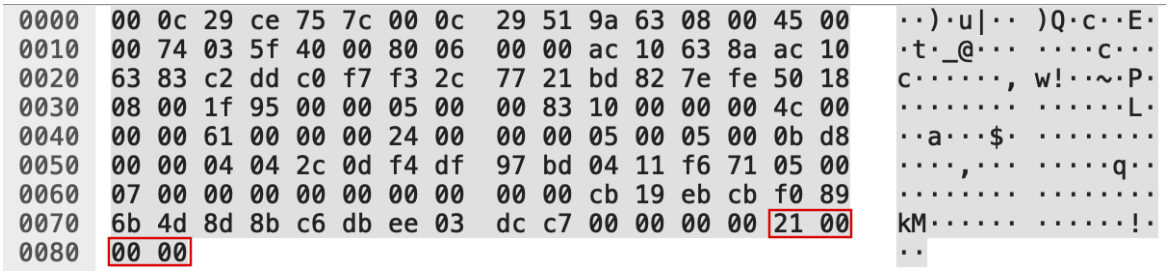


Figure 5: 'GetFuncDesc' request with function index highlighted.

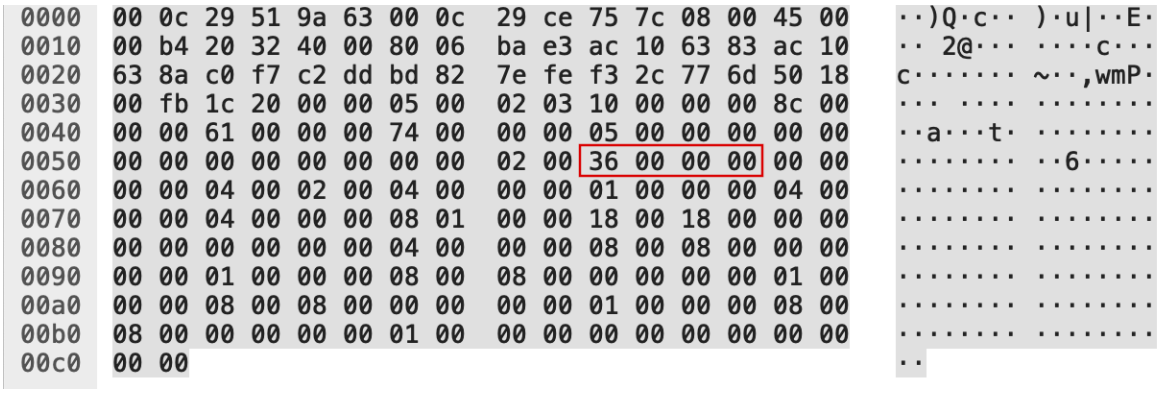


Figure 6: 'GetFuncDesc' response with MemberID highlighted.

The next operation is 'GetNames' (Opnum 7), which retrieves the human readable function name and parameters. The client must provide the MemberID obtained from the 'GetFuncDesc' operation to identify the function it would like to resolve. Figures 7 and 8 show the request and response of 'GetNames' on the wire respectively.

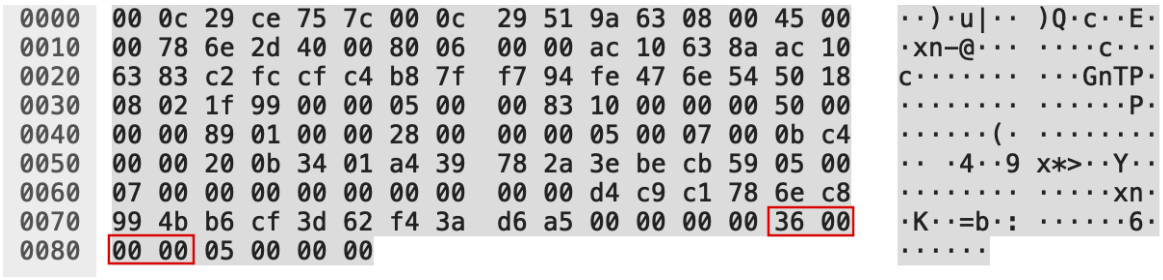


Figure 7: 'GetNames' request with function index highlighted.

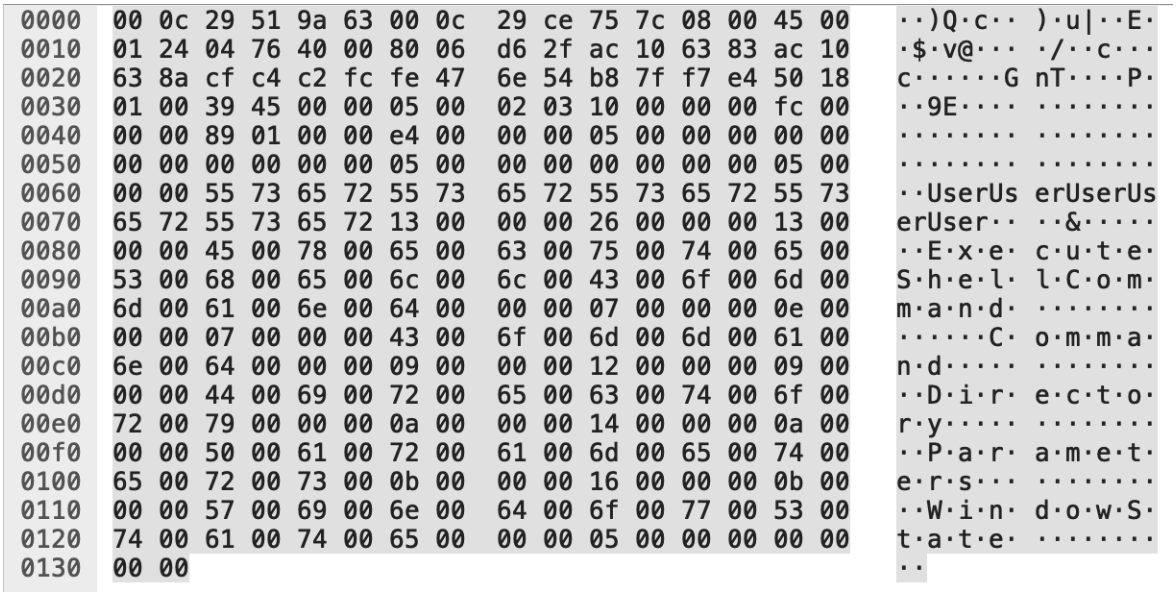


Figure 8: 'GetNames' response containing the function name and parameters.

The final ITypeInfo operation to discuss is 'GetDocumentation' (Opnum 12). This operation returns any documentation strings associated with the requested function. Once again, the client must provide a MemberID to the server. Figures 9 and 10 show the request and response of 'GetDocumentation' on the wire respectively.

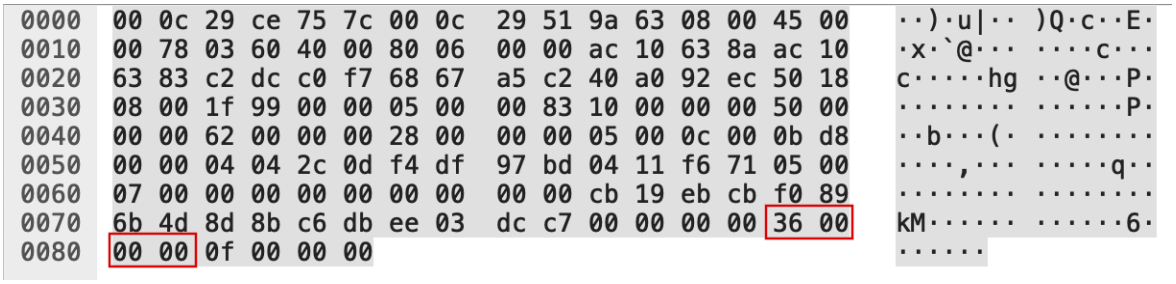


Figure 9: 'GetDocumentation' request with MemberID highlighted.

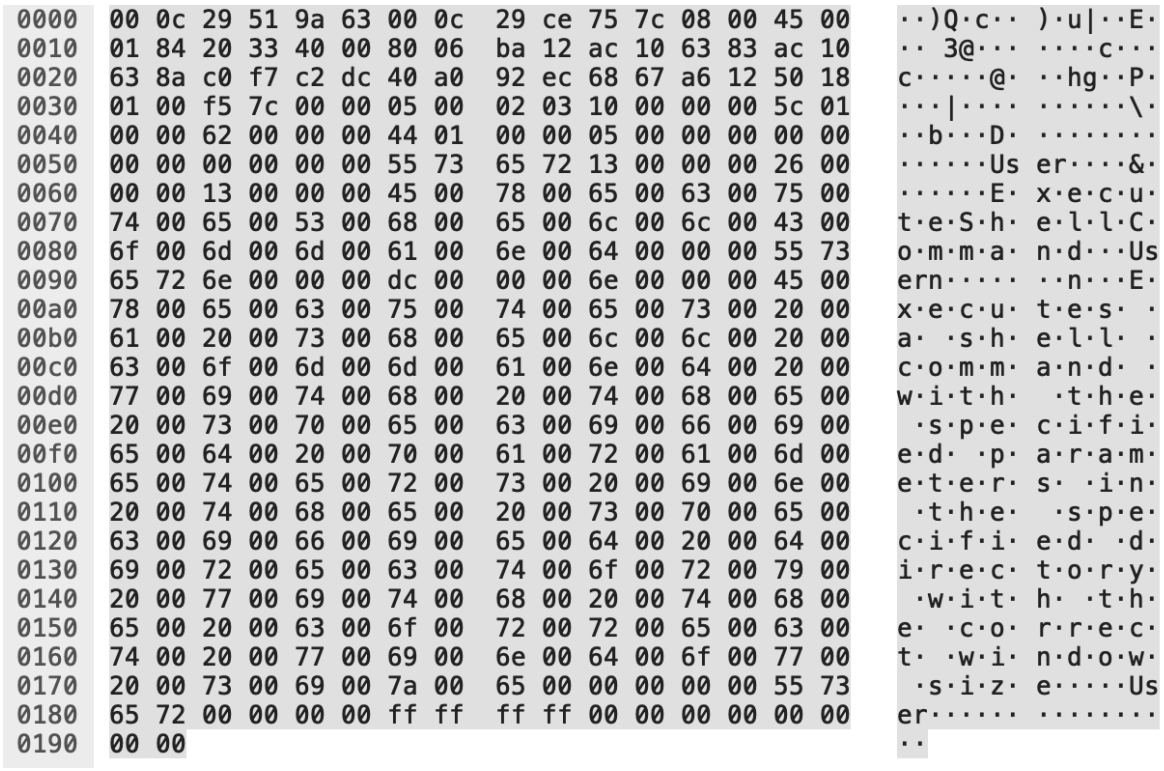


Figure 10: 'GetDocumentation' response containing documentation strings.

EXECUTION.

The final phase is the execution of a selected function by the client. In this case, the 'ExecuteShellCommand' function exposed by 'MMC20.Application' was executed. To perform this, the client uses IDispatch's 'Invoke' function. The client provides the MemberID of the function to execute along with any parameters

the function may require. The function is then executed on the remote server and any output is returned to the client. Figure 11 shows the execution of 'ExecuteShellCommand' on the wire.

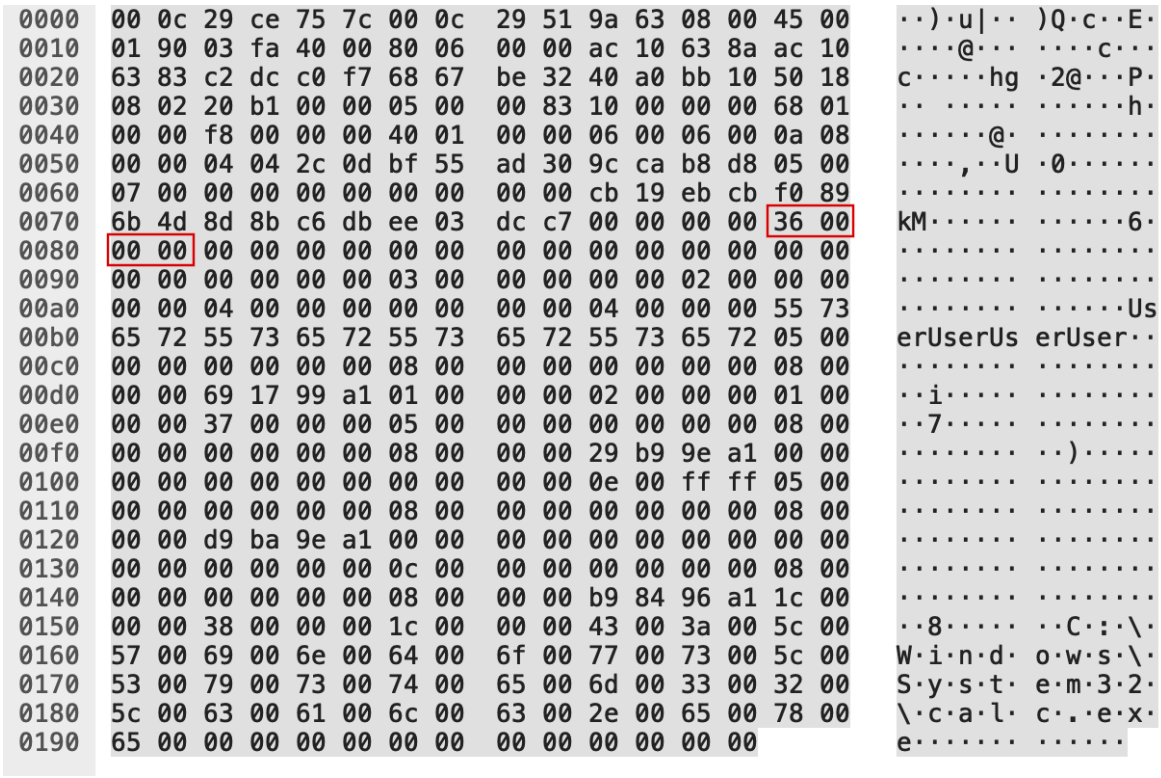


Figure 11: 'Invoke' request with MemberID highlighted.

It should be noted that certain aspects of this breakdown may vary slightly depending on certain execution factors, but the overall flow should remain roughly the same.

C. Weaponization Vectors

Like many other lateral movement techniques, DCOM can be weaponized in several different ways, which may result in slightly different behaviors. The initial phase of research found that a majority of DCOM lateral movement examples leveraged Windows PowerShell. It is for this reason that the previous section of this paper focused so heavily on the artifacts generated when DCOM is weaponized using PowerShell. However, it was hypothesized that other implementations using DCOM would exhibit different network behavior.

In order to test this, a weaponization vector that does not rely on .NET needed to be found, as .NET is what PowerShell is built atop of. The Python package titled "pypiwin32" quickly surfaced.[10] This package uses native C bindings to expose Windows functionality to Python. The process for implementing a comparable DCOM call was simple and can be seen below where the 'MMC20.Application' COM object is once again weaponized.

```
> import win32com.client
> obj = win32com.client.DispatchEx("MMC20.Application", HOSTNAME)
> obj.Document.ActiveView.ExecuteShellCommand("C:\Windows\System32\calc.exe", None, None, None)
```

After executing lateral movement using both PowerShell and Python, the collected PCAPs were compared and two differences were identified within their respective PCAPs. The first difference was that Python instantiated the object using ISystemActivator's 'RemoteCreateInstance' function while PowerShell used IClassFactory's 'CreateInstance' function. The second difference between the two behaviors was the number of ITypeInfo operations. Python only retrieved information about the specific methods and properties of the object that was being executed. PowerShell on the other hand enumerated and retrieved information about every method and property that the 'MMC20.Application' object exports.

Although the differences between the two weaponization vectors may appear slight, it is important that detection engineers understand that different techniques may produce slightly different indicators. By understanding such differences, defensive tools can be better configured to detect lateral movement regardless of the weaponization vector.

[10] https://pypi.org/project/pypiwin32/

D. Abuseable Characteristics of a COM Object

In addition to different weaponization vectors, there are many different DCOM objects that can be used to perform lateral movement. In general, an object must have the following properties to be useable for DCOM lateral movement:

1. **Means of Command Execution—**In order to move laterally, an adversary must achieve some form of code execution on a remote system, which means candidate DCOM objects must expose some command execution functionality. Depending on the needs of the adversary, this functionality can be in the form of shell command execution, loading malicious DLLs, or arbitrary script execution.

2. **Exploitable Permissions—**An adversary must also find objects whose permissions will allow them to be executed by a remote user whose account the adversary has access to. The simplest way to accomplish this is to look for objects with blank 'LaunchPermissions'. By default, this will allow administrators to instantiate the DCOM object.

3. **Registered on Target System—**Finally, the specific object must be registered on the remote system to be executed. Many applications may expose objects that fulfil the previous two criteria, however, the objects are useless if their parent applications are not installed on the target system. For this reason, objects that are registered by default Windows applications make good candidates as attackers do not need to worry about third party software being installed. However, if an adversary is aware that certain applications are prevalent in an environment, then objects exposed by those applications may be targeted.

With the above in mind, it is expected to find additional abuseable objects. Detection logic that attempts to be a "generalized" detection for the entire technique class will need to consider the diversity of the objects that can be abused.

V. Criteria Analysis – Understanding DCOM Artifacts

Criteria analysis is the process by which artifacts or components of an attack chain are classified to aid in robust detection authorship. By performing this analysis, engineers are able to craft robust detection logic as well as ensure appropriate detection coverage considering variations and removing known false-positive cases. The classification outcomes of this analysis are:

1. **Strict—**Components that are required to be present for the attack chain to exist.

2. **Loose—**Components that will commonly be present in the attack chain. Generally, at least one instance of these will be present upon execution of a technique. Also includes attacker behavior choices.

3. **Exclusion—**Components that may appear to be part of the attack chain but are benign and thus should be excluded from consideration. Identifying these is complemented if you have a large data corpus to test detection logic against.

In the specific case of DCOM as a lateral movement technique, the following criteria were identified:

STRICT CRITERIA

- The object activation phase must include a CLSID.

- There must be a method to execute code.

LOOSE CRITERIA

- The object (CLSID) being used can vary.

- How the object is instantiated can vary (‘IClassFactory::CreateInstance’ or ‘ISystemActivator:: RemoteGetClassObject’).

- ITypeInfo operations may exist.

- There might be an ‘IDispatch::Invoke’ with the Dispatch ID / MEMBERID of relevant function.

VI. Network Detection & Analysis of DCOM Lateral Movement

The goal for this research project was to determine how this particular threat could be detected and then mitigated within an environment. This section will begin by describing three different technologies commonly used in threat detection and explain the strengths and weaknesses of each. It will then explain how each of the described solutions was used to detect the presence of DCOM lateral movement.

A. Technologies Overview

When it comes to network security monitoring (NSM), there are many different techniques and technologies that can be used. A brief overview will be given of three different collection techniques, followed by how each one can be used to detect signs of DCOM lateral movement.

RULE BASED IDS

An Intrusion Detection System (IDS) is a system that monitors network traffic and creates alerts when suspicious traffic is observed. Most IDS systems are rule based, meaning users provide a defined set of rules to the system and when observed traffic matches a rule, an alert is created. Rule based IDS solutions are a good first step as they can provide information on possible malicious activity using the full packet contents, however they lack the context that other types of NSM techniques provide. Rule based systems also suffer from the issue that they only detect “known bad”. That is, a rule must be created for a specific threat before an IDS will know to alert on the traffic. This means that new threats often are not caught by rule-based IDS solutions until a rule is created to address the new threat.

METADATA EXTRACTION

Metadata extraction is a technique that seeks to reduce network traffic into a defined set of metadata. This metadata typically includes details such as important protocol headers and net flow information. Metadata can also be enriched with external sources to provide more context around events such as using GeoIP databases to tag external IP addresses. One downside to an exclusively metadata focused solution is that full packet details are not ingested. Despite the lack of full packet details, a properly engineered metadata extraction solution can be effective. Although the lack of packet details may seem like a critical flaw, by ingesting key protocol metadata into a system that can be easily queried, security engineers are able to hunt and detect threats just as effectively. Furthermore, metadata focused approaches require far less technical investment as the amount of data being stored and searched is a fraction of what full content solutions provide.

FULL CONTENT AND PCAP

The third technique is full content and PCAP extraction which when compared to the previous two techniques, provides the most context surrounding a network event. However, full PCAP solutions are notoriously hard to scale both in terms of data storage and data searching. Most solutions use tools that "index" PCAP metadata to allow for faster searching, but these tools come with their own challenges that an organization must be ready to face. For small scale PCAP analysis, Wireshark is the "go-to" tool for most people. Wireshark excels at analyzing traffic at a short snapshot at time, perhaps where an incident took place, but it is impractical for hunting or looking at traffic over a longer time period.

B. Detecting DCOM Lateral Movement with IDS

In order to help detect DCOM lateral movement, several rules for the Suricata IDS were written to trigger an alert when objects with known lateral movement capabilities are created via DCOM. These signatures will be broken down in this section (see Appendix for the rest of the signatures).

Before delving into DCOM specific IDS rules, a general process on writing IDS rules for a given threat will be provided. The first step is to simply ask the question, "what am I trying to detect"? Answers to this question can range from general to specific. In some cases, the goal of a rule may be to detect the presence of "Tool X", while other times the goal is to detect a very specific command or technique. This question is important as it will help to guide the search for possible identifiers later in the process. After creating a scope, one must gather specific data on the tool or technique that the rule will attempt to detect. Data collection can be as simple as compiling information from blog posts that analyze a specific threat. However, it may be more effective when a detection author can interact with a tool or technique in a sandbox environment. This method requires more time and resources but will allow an author to gather data over the course of multiple executions and within a variety of sandbox environments. Furthermore, by tweaking the execution each time, either by supplying different arguments or inputs, an author will have an easier time finding constant identifiers within the network traffic generated by a tool or technique. Once the data collection phase is complete, it is time to use the collected data to search for common indicators that a rule will match on. Examples of such indicators are unique byte patterns, user-agent strings, or command syntax. Once commonalities have been identified, an author now must incorporate them into a rule using the syntax of their preferred IDS. Once a rule is written, it is critical that it is tested not only to confirm that it can detect what it was written to detect, but also that it does not false positive on unrelated traffic. After all steps have been completed, the rule can be released into the desired environment, but should be reviewed periodically to confirm that the original detection goals are still being met.

Now that a process for creating IDS signatures has been described, the rule shown below, which will detect when an 'MMC20.Application' object is created using DCOM, will be explained in detail.

```
alert tcp any any -> any 135 (msg:"MMC DCOM Object Created with RemoteGetClassObject"; flow:to_server, established; content:"|03|"; offset:22; depth:1; content:"MEOW"; offset:68; depth:4; content:"|1a 79 b2 49 ae b1 90 4c 9b 8e e8 60 ba 07 f8 89|"; offset:436; depth: 16; classtype: misc-attack; sid: XXXXXX; rev: XXXXXX;)
```

The signature begins by looking for DCE / RPC opnum 3. Tracking RPC context switching is rather awkward in Suricata rules, so checking for opnum 3 will match any DCE / RPC call with an opnum of 3. However, this is not too much of an issue as the rest of the signature should throw out any packets that are not ISystemActivator 'RemoteGetClassObject' (opnum 3) calls. The string "MEOW" which is the signature for a marshalled COM interface, is then searched for.[11] Finally, the CLSID of 'MMC20.Application' is searched for. The other signatures written follow a similar pattern; first look for an opnum that indicates an object is being created, then check for the "MEOW" signature, finally look for the known CLSID of the object.

C. Detecting DCOM Lateral Movement with Zeek (Formerly Bro)

The Bro Network Security Monitor, recently renamed to Zeek, is an open-source project which can ingest live network traffic and output metadata for specific protocols or general NetFlow.[12] Zeek has three key components:

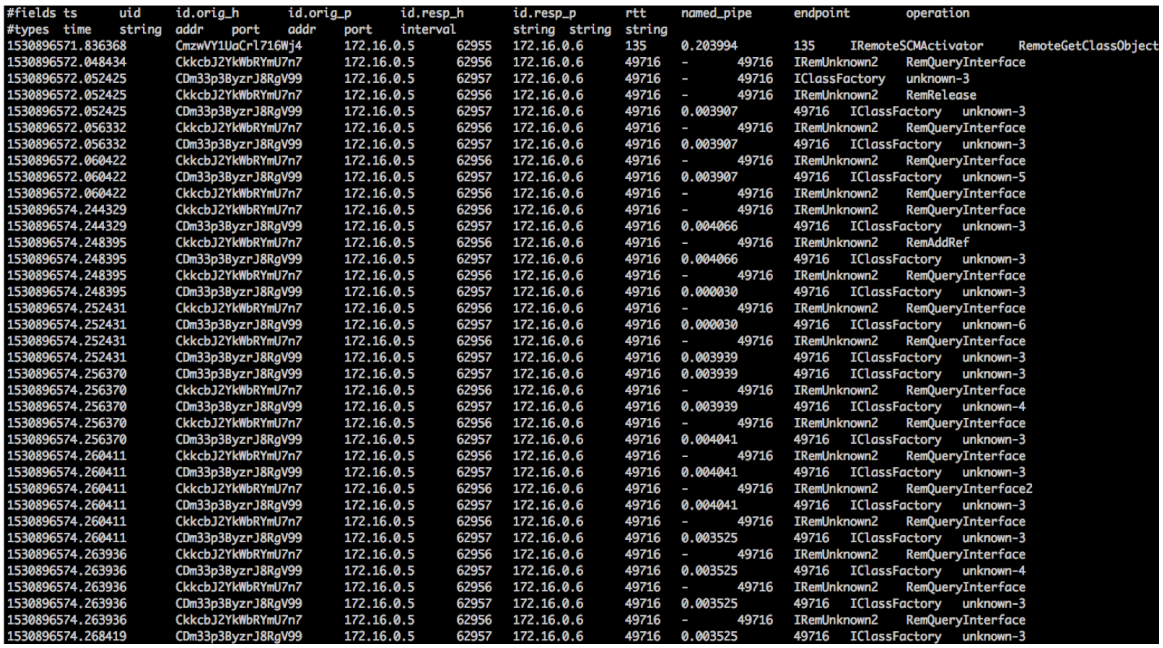
1. **BinPAC—**High level language for protocol parsers.

2. **Events—**Reduces traffic into "high level events". Generated by parsers. Passes variables defined in BinPAC to event handlers.

3. **Scripts—**Execute and handle event handlers to consume parser events.

During the initial phase of research, it was found that Zeek failed to properly parse DCOM's RPC traffic. When reviewing Zeek logs of DCOM lateral movement, the object being called was not shown in the parsed metadata as seen below.

[11] https://msdn.microsoft.com/en-us/library/cc226828.aspx
[12] https://www.bro.org/zeek.html



```
#fields ts      uid     id.orig_h       id.orig_p       id.resp_h       id.resp_p       rtt     named_pipe      endpoint        operation
#types  time    string  addr    port    addr    port    interval        string  string  string
1530896571.836368       CmzwVY1UaCrl716Wj4      172.16.0.5      62955   172.16.0.6      135     0.203994        135     IRemoteSCMActivator     RemoteGetClassObject
1530896572.048434       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896572.052425       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   -       49716   IClassFactory   unknown-3
1530896572.052425       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemRelease
1530896572.052425       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003907        49716   IClassFactory   unknown-3
1530896572.056332       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896572.056332       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003907        49716   IClassFactory   unknown-3
1530896572.060422       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896572.060422       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003907        49716   IClassFactory   unknown-5
1530896572.060422       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.244329       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.244329       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.004066        49716   IClassFactory   unknown-3
1530896574.248395       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemAddRef
1530896574.248395       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.004066        49716   IClassFactory   unknown-3
1530896574.248395       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.248395       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.000030        49716   IClassFactory   unknown-3
1530896574.252431       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.252431       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.000030        49716   IClassFactory   unknown-6
1530896574.252431       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.252431       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003939        49716   IClassFactory   unknown-3
1530896574.256370       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003939        49716   IClassFactory   unknown-3
1530896574.256370       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.256370       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003939        49716   IClassFactory   unknown-4
1530896574.256370       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.256370       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.004041        49716   IClassFactory   unknown-3
1530896574.260411       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.260411       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.004041        49716   IClassFactory   unknown-3
1530896574.260411       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface2
1530896574.260411       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.004041        49716   IClassFactory   unknown-3
1530896574.260411       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.260411       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003525        49716   IClassFactory   unknown-3
1530896574.263936       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.263936       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003525        49716   IClassFactory   unknown-4
1530896574.263936       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.263936       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003525        49716   IClassFactory   unknown-3
1530896574.263936       CkkcbJ2YkWbRYmU7n7      172.16.0.5      62956   172.16.0.6      49716   -       49716   IRemUnknown2    RemQueryInterface
1530896574.268419       CDm33p3ByzrJ8RgV99      172.16.0.5      62957   172.16.0.6      49716   0.003525        49716   IClassFactory   unknown-3
```

Figure 12: Logs showing how Zeek parses DCOM traffic.

Upon further inspection, it was found that Zeek's RPC parser does not support context switching, which is used to specify the interface to execute functions against. When Zeek parses RPC traffic that contains DCOM, the lack of context switching causes Zeek to return the same endpoint until a new interface is explicitly bound. To be clear, that means that any operations after an 'alter_context' command is issued will be associated with the wrong endpoint and therefore be erroneous logs. This issue highlights a larger problem that metadata focused approaches face; protocols that use other protocols as means of transportation will not be properly parsed unless they are explicitly parsed or handled by additional code. In this case, Zeek's RPC parser is only extracting RPC specific metadata, but fails to parse data important in the context of DCOM.

In an attempt to improve Zeek's handling of DCOM, test code was written to modify Zeek's RPC parser. Specifically, the goal of the modifications was to have Zeek parse CLSIDs when an object is instantiated. To do this, the RPC parser was modified to pass the full "stub" during a 'dce_rpc_request' to the event handler. A Zeek script was then created that parses and returns the CLSID of an object during 'RemoteGetClassObject' or 'RemoteCreateInstance' operations. These changes appeared to be effective as now the Zeek logs revealed which CLSIDs were being instantiated. The image below shows the 'MMC20.Application' CLSID being parsed by the newly modified version of Zeek.

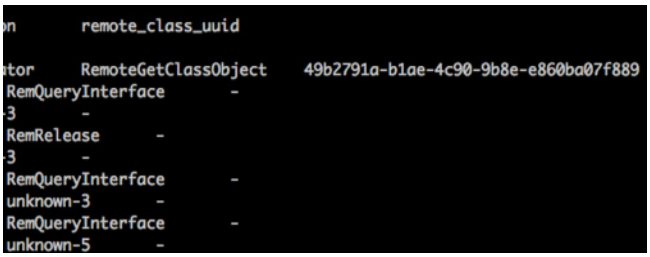


Figure 13: Log showing how the modified Zeek now parses CLSIDs.

It is important to note that these modifications are simply a proof of concept and requires further testing and review before being integrated into the Zeek project. There are potential performance implications when passing larger volumes of data to handle at later stages of the Zeek process.

D. Wireshark Open Source Contribution

As mentioned before, Wireshark is the most popular tool for packet analysis, but the current version of the tool only dissects certain DCOM operations. The ITypeInfo operations discussed earlier were among the operations not implemented. This led to large portions of the PCAP being incomprehensible as seen in the image below.

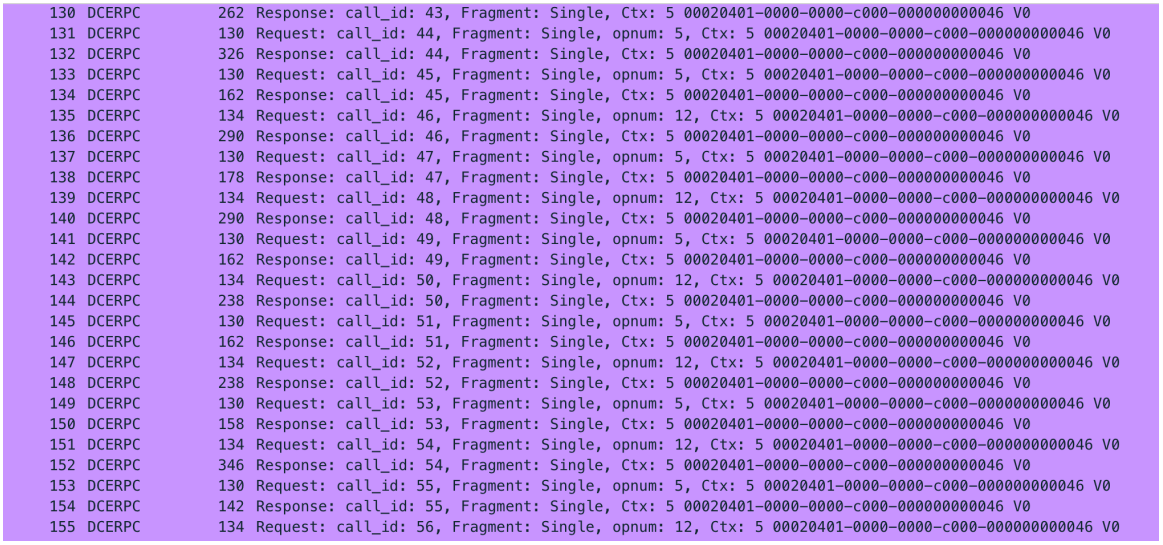


Figure 14: Screenshot showing how the current Wireshark version parses DCOM.

To combat this issue, research was done to see if it would be possible to modify Wireshark to dissect additional DCOM operations, most notably the ones discussed in Section 4. Before attempting any modifications, time was first spent learning about how Wireshark parses protocols. Within Wireshark, there are two primary features that can perform protocol dissection:

1. **Dissectors—**These iteratively analyze and parse protocols, usually subsequently handing off to sub-dissectors.

2. **Plugins—**External components that extend the functionality of Wireshark through various methods, to include protocol dissection. Written in Lua.

In general, a protocol parser can be written as either a dissector or as a plugin without much difference in end functionality. However, the original DCOM functionality exists as a dissector, so it was decided that any new DCOM functionality should be implemented as a dissector as well. This paper will not get into specific implementation details, but Wireshark's developers guide was invaluable in describing how Wireshark works under the hood and how to properly implement protocol dissection.[13]

Prior to writing the new dissector, a custom DCOM object with very basic functionality was first implemented. This object would prove to be immensely useful as it could be modified at any time to change return types, parameter types, number of parameters, etc. This provided the ability to get multiple PCAPs of different executions of this object. From there, Microsoft's documentation was used to provide insight on what structures were being sent over the wire.[14] After comparing the documentation to the gathered PCAPs, inconsistencies began surfacing, such as how certain structure properties were appearing in unexpected locations within the PCAPs. This was puzzling for quite some time, until the Network Data Representation (NDR) syntax was discovered.[15] NDR syntax is used when DCOM is executed over RPC and it dictates how data structures should be represented in octet streams. After reading about the specifics of NDR, all the "misplaced" properties in the test PCAPs begin to make sense. The biggest lesson throughout this process: always check to see if a protocol uses a specific transfer syntax before attempting to map documentation structures to PCAP data. Had this process begun by investigating how DCOM data is represented on the wire, many hours of confusion and headache would have been saved. The final part of this process was to implement the dissector itself, recompile Wireshark, and verify that the new dissector was outputting as expected.

These modifications greatly increased the readability of captured DCOM PCAPs and allowed for better understanding of the communication occurring between the client and server. The figure below shows an example of how the modifications resulted in Wireshark correctly parsing ITypeInfo operations.

---

[13] https://www.wireshark.org/docs/wsdg_html_chunked/
[14] https://msdn.microsoft.com/en-us/library/cc237759.aspx
[15] http://pubs.opengroup.org/onlinepubs/9629399/chap14.htm

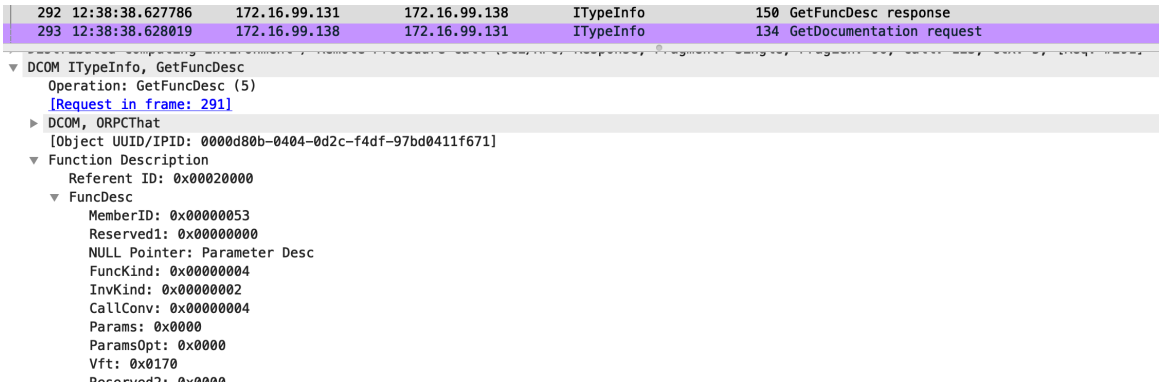


Figure 15: Screenshot showing how Wireshark now dissects DCOM traffic after the contribution.

VII. Evasion / Durability Testing—Cat and Mouse Game

So long as defensive capabilities exist, attackers are going to try to evade detection. During the detection engineering process, it is crucial that there is a contrarian involved in the exercise to serve as an adaptable adversary and attempt to evade newly minted capabilities. By implementing this as a standard part of one's detection engineering process, you ensure as robust detections as possible from the start with documented known evasion cases.

In regard to DCOM lateral movement, the research team observed public discussion over one such interesting evasion technique: RPC Encryption. According to Microsoft's documentation, RPC can be configured to use a variety of "Authentication Levels", most of which modify how much of an RPC packet is verified for integrity.[16] However, the 'RPC_C_AUTHN_LEVEL_PKT_PRIVACY' level not only verifies data integrity, but also encrypts the RPC arguments. When this is the case, all the aforementioned detection techniques will fail as they require that the RPC arguments are plaintext. With that being said, encryption can also serve as an indicator in some cases. For example, if an organization does not have RPC encryption enabled by default, encrypted calls may stand out as suspicious and may be indicative of an attacker attempting to evade defensive systems using encryption. For this reason, it is important that security professionals understand the environment they are monitoring so that anomalous behavior can be properly investigated.

[16] https://docs.microsoft.com/en-us/windows/desktop/rpc/authentication-level-constants

VIII. Conclusion

Threat actors thrive off native Windows functionality that fits in their natural workflow. It enables them to lurk in the shadows and hide under the guise of "legitimate" administrative activity. Even though DCOM is a decades old Windows subsystem, its abuse by adversaries in 2017 was innovative. As attackers innovate, defenders must stay abreast of new tactics and techniques and aim to detect them. A structured detection engineering process enabled engineers to research an adversary technique, understand the associated artifacts, and improve multiple technologies to allow for detection. It is the hope of the research team that by being transparent in the methodology used and freely sharing example work, others will advance the state of the research and continue to close the gap on threats facing enterprises.

APPENDIX

alert tcp any any -> any 135 (msg:"MMC DCOM Object Created with RemoteGetClassObject"; flow:to_server, established; content:"|03|"; offset:22; depth:1; content:"MEOW"; offset:68; depth:4; content:"|1a 79 b2 49 ae b1 90 4c 9b 8e e8 60 ba 07 f8 89|"; classtype: misc-attack; sid: XXXXXX; rev: XXXXXX;)

alert tcp any any -> any 135 (msg:"Excel.Application DCOM Object Created with RemoteGetClassObject"; flow:to_server, established; content:"|03|"; offset:22; depth:1; content:"MEOW"; offset:68; depth:4; content:"|00 45 02 00 00 00 00 00 C0 00 00 00 00 00 00 46|"; classtype: misc-attack; sid: XXXXXX; rev: XXXXXX;)

alert tcp any any -> any 135 (msg:"FileExplorer DCOM Object Created with RemoteGetClassObject"; flow:to_server, established; content:"|03|"; offset:22; depth:1; content:"MEOW"; offset:68; depth:4; content:"|72 59 a0 9b a8 f6 cf 11 a4 42 00 a0 c9 0a 8f 39|"; classtype: misc-attack; sid: XXXXXX; rev: XXXXXX;)

alert tcp any any -> any 135 (msg:"htafile DCOM Object Created with RemoteCreateInstance"; flow:to_server, established; content:"|04|"; offset:22; depth:1; content:"MEOW"; offset:72; depth:4; content:"|d8 f4 50 30 b5 98 cf 11 bb 82 00 aa 00 bd ce 0b|"; classtype: misc-attack;sid: XXXXXX; rev: XXXXXX;)

alert tcp any any -> any 135 (msg:"Outlook.Application DCOM Object Created with RemoteGetClassObject"; flow:to_server, established; content:"|03|"; offset:22; depth:1; content:"MEOW"; offset:68; depth:4; content:"|3a f0 06 00 00 00 00 00 c0 00 00 00 00 00 00 46|"; classtype: misc-attack; sid: XXXXXX; rev: XXXXXX;)