Global Information Assurance Certification Paper

Identifying Vulnerable Network Protocols with PowerShell

GIAC (GCIA) Gold Certification

Author: David R Fletcher Jr, 6fletch9@gmail.com
Advisor: Manuel Humberto Santander Pelaez
Accepted: February 20th 2017
Template Version September 2014

Abstract

Microsoft Windows PowerShell has led to several exploit frameworks such as PowerSploit, PowerView,and PowerShell Empire. However, few of these frameworks investigate network traffic for exploitative potential. Analyzing a small amount of network traffic can lead to the discovery of possible network-based attack vectors such as Virtual Router Redundancy Protocol (VRRP), Dynamic Trunking Protocol (DTP), Link Local Multicast Name Resolution (LL-MNR) and PXE boot attacks, to name a few. How does one gather and analyze this traffic when Windows does not include an integrated packet analysis tool? Microsoft Windows PowerShell includes several network analysis and network traffic related capabilities. This paper will explore the use of these capabilities with the goal of building a PowerShell reconnaissance module which will capture, analyze, and identify commonly misconfigured protocols without the need to install a third-party tool within a Microsoft Windows environment.

1. Introduction

During a typical penetration test a great deal of focus is placed on vulnerabilities found in operating systems and software applications. However, an often-overlooked area of vulnerability analysis deals with network configuration errors. Many computers and network devices are deployed with default or improper configurations that expose them to various attacks.

In some cases, the simple observation of a given protocol may indicate vulnerability. Protocols such as Virtual Local Area Network (VLAN) trunking, network routing, and network redundancy protocols typically should not be propagated to the client. This is because an attacker with access to these protocols may be able to manipulate the flow of traffic across the network, expand access to other subnets, or cause denial of service.

In other cases, investigation into a protocol's configuration may lead to second order effects. In the case of Dynamic Host Configuration Protocol (DHCP), certain options present may give an attacker the opportunity to analyze a boot image for credentials or other sensitive information. As an alternative, the attacker could attempt to force a user to boot a malicious image in order to expand their foothold.

Many protocol analysis tools already exist. Tools such as windump, tcpdump, Wireshark, and Microsoft Message Analyzer allow a network analyst to troubleshoot issues within their respective network. However, if the penetration testing rules of engagement do not accommodate installation of software, an attacker must improvise.

This paper will investigate current protocols of interest which represent potential exploitable vulnerabilities within an environment. After cataloging the protocols, methods for identifying them from the perspective of a standard Microsoft Windows client computer will be explored. These methods will then be used to generate a script modeled after the PowerShell Empire PowerUp script to provide easy identification of the targeted protocols without the need to install third-party tools. The resulting script will allow both attackers and defenders to quickly evaluate an environment for common vulnerabilities.

David R Fletcher Jr., 6fletch9@gmail.com

This focus of the resulting script is on identification of vulnerable protocols only. This script currently supports IPv4 and may work with IPv6. The IPv6 header is currently processed. However, only the first "next header" field is currently evaluated. Exhaustive testing of each of the protocol parsers could not be accomplished in the time allotted. Future enhancements will include full stability testing, full support for IPv6 processing, and may include attack capabilities.

2. Background

2.1. Protocols of Interest

The following protocols are covered due to the presence of current tools to take advantage of vulnerable configurations. This list can be expanded upon based on future toolset expansion.

Name Resolution Protocols:

Name resolution protocols provide an opportunity for an attacker to execute several different attacks. By manipulating the hostname to IP address relationship, an attacker can send malicious responses to a user's requests or to become a Man-in-the-Middle (MitM) in the network conversation. By doing so, the attacker can observe all traffic passing between the two communicating parties. As a result, the attacker can gather sensitive information such as authentication credentials or manipulate information transmitted to either party.

NetBIOS Name Service (NBT-NS) - RFC 1001 and 1002 define the components of the NetBIOS protocol suite. One of the elements of this protocol is the NetBIOS Name Service. This service is used to perform name resolution within a Windows environment. NBT-NS communication can be identified on the network by listening for packets on TCP and UDP port 137. NBT-NS is a broadcast protocol; therefore, the destination address of these packets will be the subnet broadcast address (IETF, 1987).

Link Local Multicast Name Resolution (LLMNR) and Multicast DNS (mDNS) - According to RFC 4795, this protocol is meant to enable name resolution when conventional DNS is unavailable (Aboba, Thaler, & Esibov, 2007). In recent versions of

Microsoft Windows operating systems, LLMNR is included as a successor and serves as a successor to the NBT-NS protocol.

LLMNR communication can be identified on the network by listening for packets on TCP and UDP port 5355. The IPv4 address for LLMNR is 224.0.0.252 using MAC address 01-00-5E-00-00-FC. The IPv6 address for LLMNR is FF02::1:3 using MAC address 33-33-00-01-00-03 (Aboba, Thaler, & Esibov, 2007). This information is summarized in the table below.

Ethernet	IPv4	IPv6
01-00-5e-00-00-fc 33-33-00-00-01-03	224.0.0.252	ff02::1:3

Figure 1: LLMNR Multicast Addresses

The protocols mentioned above allow computers within the same broadcast domain to assist one another in the face of a DNS failure. If enabled, both may allow an attacker with access to a vulnerable network to spoof responses to observed queries. When a Windows host receives the spoofed response, then that host will attempt to communicate with the attacker's target using the client's desired protocol (Sternstein).

Typical LLMNR queries observed are for protocols such as SMB, WPAD, and others which require authentication. As a consequence, the client automatically attempts to complete challenge-response authentication with the attacker's service. This results in the attacker capturing the user's LM or NT hash for use in pass-the-hash attacks or password cracking (Gaffie, 2013). Credentials captured and cracked can be used for direct access to resources within the Active Directory domain. With authenticated access, an attacker can quickly escalate privilege and completely compromise the Active Directory environment.

Routing and Redundancy Protocols:

Routing protocol traffic should not be propagated to access ports. This routing information can be valuable for simple network reconnaissance. In addition, the protocol and its configuration could expose the network to route manipulation attacks. If routing traffic is present on an access port, an attacker can parse this information to determine whether authentication is being used to capture credentials. Without authentication, the

attacker may be able to inject routing information that causes traffic to pass through a computer that the attacker controls.

Hot Standby Routing Protocol (HSRP) - RFC 2281 describes the Cisco proprietary Hot Standby Router Protocol. This protocol provides default gateway redundancy using multicast communication. The active router is used as the default gateway until it becomes inaccessible. Once this happens, the standby router with the next highest assigned priority will assume the IP and MAC address of the active router's interface resulting in failover without any service interruption (Li, Cole, Morton & Li, 1998).

HSRP can be identified by its multicast addresses, which are 224.0.0.2 using UDP 1985 (v1), 224.0.0.102 (v2) using UDP 1985, and ff02::66 using UDP 2029 (Li, Cole, Morton & Li, 1998). These details are summarized in the table below.

Ethernet	IPv4	IPv6
01-00-5e-00-00-02	224.0.0.2 224.0.0.102	ff02::66

Figure 2: HSRP Multicast Addresses

Virtual Router Redundancy Protocol (VRRP) - VRRP is described by RFC 5798 as an election protocol used by routers sharing an IPv4 or IPv6 address which provides routing redundancy and dynamic failover for a network. Multiple routers are used to provide this redundancy. The master router is used for forwarding of traffic on the segment. Once the master router becomes unavailable, one of the secondary routers takes over forwarding after being elected as the new master (Nadas & Ericsson, 2010).

VRRP can be identified by its multicast address, which is IPv4 224.0.0.18 and IPv6 ff02::12 using IP protocol number 112 (Nadas & Ericsson, 2010).

If either of these protocols is not sufficiently protected and propagated to an access port on an Ethernet switch, an attacker may be able to attempt to elect himself as the master or active router. Once this occurs, the attacker could manipulate the flow of network traffic to collect sensitive information or MitM sessions propagating along the route (Wright, 2015).

Open Shortest Path First (OSPF) - RFC 2328 describes this interior network routing protocol. It is one of several interior routing protocols that allow network infrastructure devices to determine routes to other interior layer 3 networks and that may include a default route to the larger internet (Moy, 1998). Typically, interior routing protocols differ in the method by which they determine the most desirable route and in which they are either open source or proprietary.

Whether proprietary or open source, all these protocols perform the same basic function, automated aggregation of routing information based on router to router relationships. Some of the protocols identified above support authentication based on the design specifications in the applicable RFC. If an attacker can attain membership in the interior routing hierarchy, then that attacker can influence the routing of packets across the network. As a result, the attacker can become MitM and manipulate or eavesdrop on legitimate traffic searching for sensitive information such as session cookies or network credentials (Wright, 2015).

OSPF traffic on the network can be identified by its multicast Ethernet and IP addresses seen in the table below. In addition, OSPF packets use IP protocol number 89 (Moy, 1998).

Ethernet	IPv4	IPv6
01-00-5e-00-00-05		
01-00-5e-00-00-06	224.0.0.5	ff0::5
33-33-00-00-00-05	224.0.0.6	ff02::6
33-33-00-00-00-06		

Figure 3: OSPF Multicast Addresses

Link-Layer Protocols:

Spanning Tree Protocol (STP) - STP is a layer 2 protocol defined by IEEE 802.1D. This protocol is used to prevent loops within a layer 2 mesh network. This is accomplished through an election process whereby only one connected uplink is permitted to forward Ethernet frames (IEEE, 2004). Since this information is primarily valuable to layer 2 switching devices, it should not be propagated to access ports. An attacker who can observe and manipulate STP traffic can become Man-in-the-Middle (MitM) by electing himself as the root bridge within the STP domain (Barroso & Andres).

The various STP versions (STP, RSTP, and MST) can be identified by the presence of the destination multicast Ethernet address 01:80:C2:00:00:00 within frames (IEEE, 2004).

Cisco Discovery Protocol (CDP) and Logical Link Discovery Protocol (LLDP) - CDP and LLDP are proprietary and open source information sharing protocols that may provide valuable information to an attacker. While the CDP standard is defined by Cisco Systems, Inc, LLDP is defined in IEEE 802.1AB. Both protocols expose the following types of information which may be a valuable element of reconnaissance in staging follow-on attacks (IEEE, 2009):

- Service Discovery Information
- Device Hardware Revision
- Device Software Revision
- Serial and Service Tag Numbers

Service discovery data can be used to locate Voice Over IP (VoIP) services on the network. Hardware and software revision information can be useful in performing vulnerability research in order to select a suitable exploit. Some human machine interfaces such as Integrated Lights Out (ILO) and other web interfaces use device serial numbers as an authentication mechanism.

CDP and LLDP traffic can be identified by the presence of any of the following multicast addresses: 01-80-c2-00-00-00, 01-80-c2-00-00-03, 01-80-c2-00-00-0e or 01:00:0c:cc:cc:cc.

Dynamic Trunking Protocol (DTP) and VLAN Trunking Protocol (VTP) - DTP and VTP are two other Cisco proprietary protocols for managing VLAN configuration on a network. Both protocols are meant to reduce management overhead. However, both also carry the potential to introduce vulnerabilities into an environment.

If an attacker observes VTP in use on an access port, then the attacker may have the ability to cause a denial of service by clearing the VTP server configuration. More interestingly, DTP is responsible for negotiating Virtual LAN (VLAN) trunk configuration between two switches that support the protocol. An attacker who observes

this protocol on an access port can masquerade as a participating switch to alter the VLAN configuration of the attached port. This can grant the attacker the ability to hop VLANs and consequently access resources that the administrator did not intend (Wright, 2015) (Rouiller).

DTP and VTP can be identified by their multicast MAC address. These protocols use the same multicast Ethernet address as CDP traffic (01:00:0c:cc:cc:cc).

Boot Protocols:

Boot protocols allow hosts on the network to obtain configuration information necessary for proper operation on the network. Some boot protocol implementations provide limited configuration options focusing on network parameters only. Others allow a host to find and boot its full operating system from the network.

Protocols and options that distribute sensitive information such as boot images with integrated credentials should be propagated to the smallest audience possible. As an example, an attacker may download and inspect a boot image to discover credentials which may be useful in attacks.

Dynamic Host Configuration Protocol (DHCP) and BOOTP - DHCP and BOOTP are typically used to configure TCP/IP parameters on hosts automatically within a network environment. Among the configuration options normally observed are IP address, default gateways, DNS servers, network time servers, and domain suffixes. These configuration parameters can be observed on the end host and provide little value to an attacker.

However, DHCP can be used to supply client boot configuration information within options 66, 67, 128, and 150 (Microsoft, 2008) for standard network boot and options 208-210 for PXELINUX requests (IANA, 2016). This information should be limited to the DHCP scope which employs network boot technologies. However, administrators may configure these parameters and advertise them to an entire network without understanding the resulting security implications. DHCP is typically used in this fashion for thin client and operating system deployment solutions such as Windows Deployment Services (WDS). When the client receives a DHCP response with these

options configured, the client can attempt to perform a network boot (or a local user can typically press F12 to cause the computer to network boot). The boot image identified in the appropriate option is then retrieved from the boot server using TFTP (Microsoft, 2014).

Boot images may include default credentials or configuration information that would be valuable to an attacker. An attacker observing these options in use could (Wright, 2015):

- Configure the computer he/she is using to perform a network boot to inspect the loaded boot image.
- Download the boot image from the TFTP server using the DHCP option values and inspect the boot image at rest.
- Attempt to spoof DHCP responses with an alternate boot server and image using a malicious image that would be valuable to the attacker.

DHCP can be identified by observing broadcast traffic on UDP ports 67 and 68.

2.1.1. Protocol Detection

Each of the protocols identified above can be detected using either their layer 2 (MAC addresses), 3 (IP addresses), or 4 (protocol and port) characteristics. These details have been assembled from the previous sections of this paper and consolidated into the table seen in Figure 4 below:

Protocol	Layer 2	Layer 3 (IPv4)	Layer 3 (IPv6)	Layer 4	Port
CDP/DTP/VTP	01-00-0c-cc-cc-cc	N/A	N/A	N/A	N/A
DHCP	ff-ff-ff-ff-ff-ff	Broadcast	Broadcast	UDP (17)	68
HSRP	01-00-5e-00-00-02	224.0.0.2 224.0.0.102	ff02::66	UDP (17)	1985 2029
LLDP	01-80-c2-00-00-00 01-80-c2-00-00-03 01-80-c2-00-00-0e	N/A	N/A	N/A	N/A
LLMNR	01-00-5e-00-00-fc 33-33-00-00-01-03	224.0.0.252	ff02::1:3	UDP (17)	5355
mDNS	01-00-5e-00-00-fb 33-33-00-00-00-fb	224.0.0.251	ff02::fb	UDP (17)	5353
NBNS	ff-ff-ff-ff-ff-ff	Broadcast	Broadcast	UDP (17)	137
OSPF	01-00-5e-00-00-05 01-00-5e-00-00-06 33-33-00-00-00-05 33-33-00-00-00-06	224.0.0.5 224.0.0.6	ff0::5 ff02::6	OSPF (89)	N/A

STP	01-80-c2-00-00-00	N/A	N/A	N/A	N/A
VRRP	01-00-5e-00-00-12	224.0.0.18	ff02::12	VRRP (112)	N/A

Figure 4: Protocol Identifying Information

Prior to initiating communication at layer 3 and above, a client must determine the layer 2 address it must use for forwarding traffic. When a client does not know the layer 2 address for a host, it uses the Address Resolution Protocol (ARP) to discover the appropriate address for transmission. A client is likely to send many packets to the same host during communication. As a result, the client maintains layer 2 to layer 3 address mappings in the ARP cache of the sending host (Stevens & Wright, 1994). An example can be seen in Figure 5 below:

```
Interface: 192.168.176.1 --- 0x5
  Internet Address      Physical Address      Type
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  239.255.255.250       01-00-5e-7f-ff-fa     static

Interface: 192.168.1.67 --- 0x6
  Internet Address      Physical Address      Type
  192.168.1.82          88-de-a9-5d-5c-e5     dynamic
  192.168.1.253         b4-b5-2f-11-5b-ba     dynamic
  192.168.1.254         f8-2c-18-33-23-b9     dynamic
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static

Interface: 192.168.211.1 --- 0x16
  Internet Address      Physical Address      Type
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```

Figure 5: ARP Cache Example

When a client observes network traffic from any of the protocols of interest, it will cache the layer 2 and layer 3 addresses in the ARP cache. Inspection of the cache entries provides a quick check indicator to determine whether the protocol has been recently observed. This technique can be used identify all of the protocols above except for DHCP and NBNS which use the broadcast MAC address to send traffic of interest.

At layer 3 and above, detection must occur through the reception and inspection of packets. This can occur synchronously by using a packet sniffer to receive, parse, and

display interesting information to the user. Alternatively, asynchronous inspection can be used as well. Instead of decoding and displaying traffic in a live environment, the packets are stored on disk, transferred to a different computer, and analyzed using a tool like windump, tcpdump, Wireshark, or Microsoft Message Analyzer. The former supports quick detection and analysis on the target host while the latter requires file transfer and offline analysis which may preclude attack. Where possible, online analysis is preferred due to time and access constraints during an attack.

The native PowerShell interpreter on Microsoft Windows operating systems prior to Windows 8.1 only supports saving packet data as an Event Tracing for Windows (ETW) file. Newer Microsoft Windows operating systems can save packet captures directly to the familiar PCAP format. Where ETW format must be used, the trace file can be transferred to a computer with Microsoft Message Analyzer installed, opened using this tool, and saved to the PCAP format.

With an understanding of the protocols of interest and the methods of detection available, a solution can be developed to provide quick triage detection capabilities. In the event that a vulnerable protocol is detected, the tester can then choose an appropriate tool for follow-on attack.

3. Script Solution

The initial concept for this paper involved asynchronous collection and analysis of traffic on the same computer. However, a great deal of time was spent attempting to collect and parse the information using the native PowerShell trace capabilities. Analysis using the produced trace files was cumbersome even within the Microsoft Message Analyzer tool. Packet data was encapsulated within XML messages which added a layer of indirection which was difficult to navigate and poorly documented.

Taking a step back from these difficulties caused the tool concept to evolve into a script with three capabilities. The first capability was to parse the ARP cache for quick identification of interesting MAC and IP addresses. The next capability was to perform live analysis and output notifications to the console of the computer where the script was running. The final capability was the collection of network traffic for a configurable

amount of time so that the resulting PCAP or ETW file could be analyzed on a separate computer.

3.1.1. ARP Cache Analysis

ARP cache analysis was performed using a combination of old and new techniques. Newer versions of Microsoft Windows (8.1 and above) support the Get-NetAdapter and Get-NetNeighbor PowerShell commandlets. These commandlets provide object- oriented access to the installed network adapter details and the ARP cache entries respectively. Each was encapsulated in a try/catch block to provide fallback support if a recent version of PowerShell was not installed. The fallback for these capabilities involved use of the netsh command.

As an equivalent to the Get-NetAdapter commandlet, the "netsh int show int" command was executed and the output was parsed in a format compatible with that produced by Get-NetAdapter. This functionality was encapsulated in the PowerShell function Get-ParsedAdapterNames.

To produce output equivalent to the Get-NetNeighbor commandlet, the "netsh int ipv4 show neigh" and "netsh int ipv6 show neigh" commands were executed using the adapter names produced above as arguments. The result was collected and parsed in a fashion similar to that described above. This functionality was encapsulated in the PowerShell function Get-ParsedArpTables.

After collecting the information necessary for analysis, the resulting data was passed through two switch statements which simply inspected for the known multicast MAC and IP addresses identified in Table 1. Figure 6 shows output from script execution on a Windows 10 host, as seen below.

```
PS C:\Temp> Invoke-NeighborCacheAnalysis
[+] Checking Neighbor Entries for Known Protocol Addresses (Ethernet)
[+] Checking Neighbor Entries for Known Protocol Addresses (Wi-Fi)
   [-] IPv4 mDNS Address Found in Neighbor Cache
[+] Checking Neighbor Entries for Known Protocol Addresses (VMware Network Adapter VMnet8)
   [-] IPv4 mDNS Address Found in Neighbor Cache
[+] Checking Neighbor Entries for Known Protocol Addresses (VMware Network Adapter VMnet1)
   [-] IPv4 mDNS Address Found in Neighbor Cache
PS C:\Temp>
```

Figure 6: Invoke-NeighborCacheAnalysis Output

3.1.2. Live Analysis

The live analysis capability of this script was inspired and informed by the Inveigh.ps1 script. The Inveigh tool was developed by Kevin Robertson as an NBNS and LLMNR cache poisoning and attack tool (Robertson, 2015). The sniffer module and general workflow from Inveigh were adopted due to the simplicity. The sniffer code methodology, described below, was observed in use in several other projects as well.

This script uses a raw IP socket to collect and inspect the traffic that the host computer is able to observe. The sniffer runs in a separate thread and communicates with the calling script through a global "analyzer" object. This object contains an ArrayList which is populated by the sniffer and consumed by the calling script. Messages found in the ArrayList are emitted to the user via the console.

Each packet received by the computer is passed to the raw socket where it is processed. First the IP header is parsed and passed to a switch statement which inspects the embedded protocol number. If one of the identified protocols (UDP, OSPF, or VRRP) is found, then the protocol header is processed. In the case of OSPF and VRRP, the full payload is processed and any interesting results are passed to the user via the console.

If a UDP packet is discovered, the packet is passed to another switch statement which inspects the destination port. If the port is found to contain the value of any of the protocols of interest then the remaining payload is parsed (after checking for the appropriate multicast address). Once again, results identifying the presence of any vulnerable protocols are passed to the user via the console.

The full functionality of this script is embodied in a single PowerShell commandlet which requires no arguments named Invoke-LiveAnalysis. While the script is running, the user can alter its behavior through runtime interaction. The script listens for available keystrokes and acts if one is received. Any unrecognized keystroke displays the available keystroke options. Recognized keystroke options are used to toggle output for specific protocols and to shut down the analyzer as described in Figure 7 below:

Keystroke	Action
D	Toggle DHCP Display
H	Toggle HSRP Display
L	Toggle LLMNR Display

M	Toggle mDNS Display
N	Toggle NBNS Display
O	Toggle OSPF Display
V	Toggle VRRP Display
Q	Shut Down Analyzer

Figure 7: Runtime Interaction Keystrokes

Each of the embedded protocols was parsed according to the field definitions and rules indicated by the RFC or details found in TCP/IP Illustrated (Stevens & Wright, 1994). Sufficient traffic and time were not available to perform exhaustive testing against each one of the protocols of interest. During testing, packet captures made available at wireshark.org and packetlife.net (Stretch) were used in conjunction with live traffic and traffic generated using the scapy tool.

Output from the analysis console can be seen below in Figures 8-13. Each graphic shows results from performing DHCP, LLMNR, mDNS, OSPF, NBNS, HSRP, and VRRP analysis. In each instance, the target protocol is parsed with details from the parsing activity printed to the console. Unfortunately, no native support for collection and analysis of layer 2 traffic could be found during the period in which the research was accomplished.

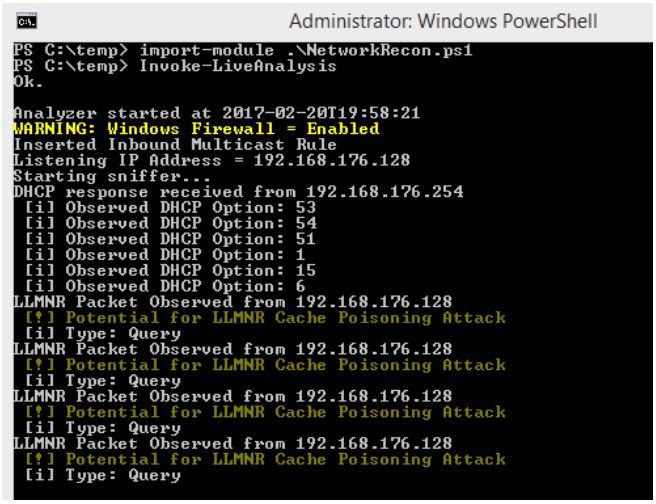

Figure 8: Invoke-LiveAnalysis Startup, DHCP, and LLMNR Processing

Figure 9: Invoke-LiveAnalysis mDNS Processing

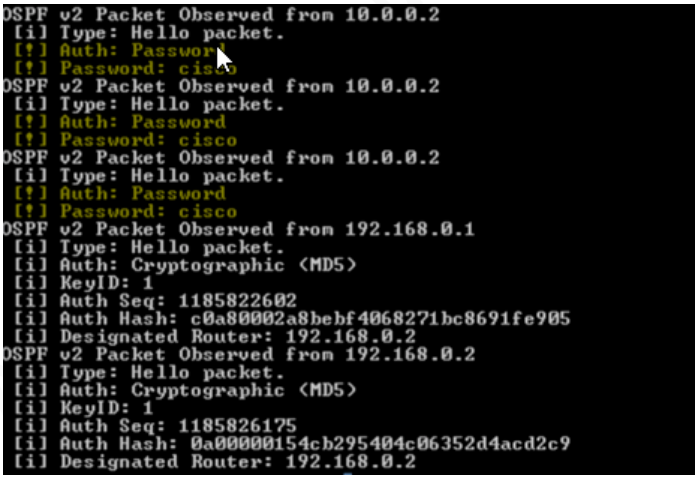

Figure 10: Invoke-LiveAnalysis OSPF Processing

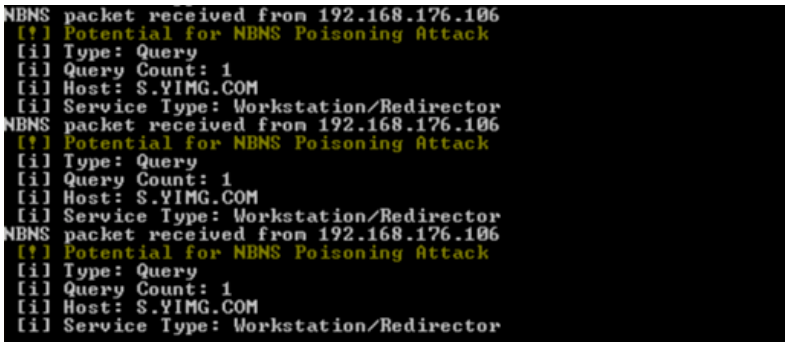

Figure 11: Invoke-LiveAnalysis NBNS Processing

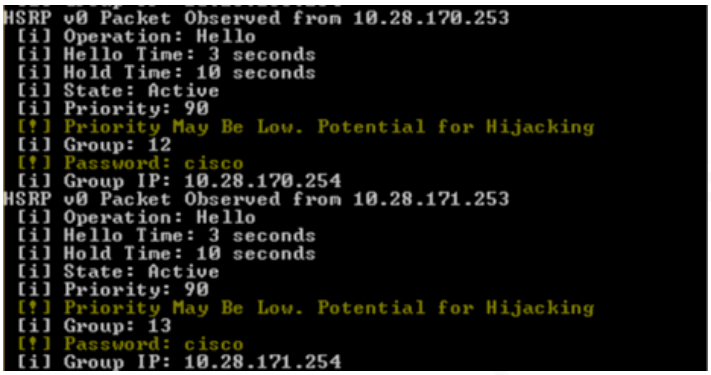

Figure 12: Invoke-LiveAnalysis HSRP Processing

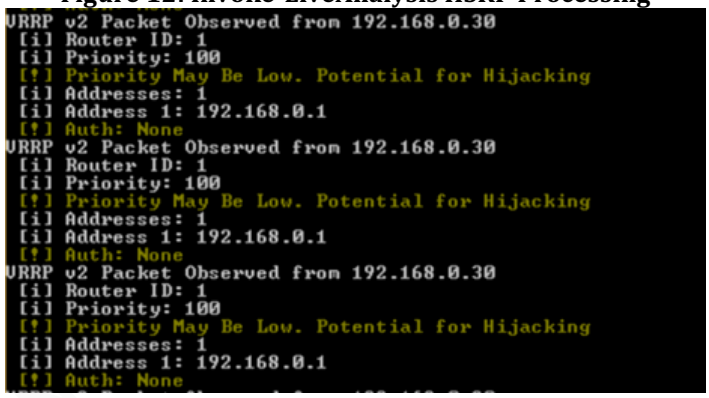

Figure 13: Invoke-LiveAnalysis VRRP Processing

Full source code for the Invoke-LiveAnalysis commandlet can be found in the appendix which accompanies this report.

3.1.3. Traffic Collection

Neither Invoke-NeighborCacheAnalysis nor Invoke-LiveAnalysis are able to catalog the full breadth of interesting protocols individually. As a result, a final commandlet named Invoke-TraceCollect was included to perform network packet capture. Similar to Invoke-NeighborCacheAnalysis, recent versions of Microsoft Windows operating systems natively support advanced PowerShell features.

Specifically, the Protocol Engineering Framework (PEF) trace module allows the use of the "Microsoft-Windows-NDIS-PacketCapture" provider. Availability of this module and provider captures all traffic at layer 2 and above and can be saved in PCAP format natively. Once again, should this module fail to load, the script falls back on the netsh command.

Netsh is a component of the Microsoft Windows operating system which is able to save a trace file in the Event Trace Log (ETL) file format. This file can be subsequently opened using Microsoft Message Analyzer and saved in PCAP format. The script executes the "netsh trace start provider=Microsoft-Windows-NDIS-PacketCapture" command with arguments specifying the path and size of the trace file.

Unlike the two previous commandets, the Invoke-TraceCollect function supports four optional parameters specifying the target folder, file name, trace duration, and maximum trace size. If none of these parameters are provided, the script will perform collection for five minutes saving the trace file to the C:\temp directory using the timestamp at execution for the filename. Execution in this manner can be seen below in Figure 14. The resulting packet capture can be seen in Figure 15:

```
PS C:\temp> import-module .\NetworkRecon.ps1
PS C:\temp> Invoke-TraceCollect
[+] Starting capture session
   [-] Trying PEF trace first...
   [-] Successfully created PEF Trace Session...
   [-] Output will be saved to C:\temp\capture_2017220204831.cap
   [-] Trace will execute for 5 minutes while packet capture is running
Session: capture_2017220204831  Received: 188  Filtered: 169  Pending: 0  Droppe
d: 0
Session: capture_2017220204831  Received: 188  Filtered: 169  Pending: 0  Droppe
d: 0
   [-] Session stopped
[+] Packet capture complete
PS C:\temp> _
```

Figure 14: Invoke-TraceCollect Execution

Name	Date modified	Type	Size
capture_2017220204831.cap	2/20/2017 8:54 PM	Wireshark capture...	58 KB
NetworkRecon.ps1	2/20/2017 12:29 PM	Windows PowerS...	97 KB

Figure 15: Recorded Packet Capture

4. Conclusion

During vulnerability analysis and penetration testing, network protocols, their visibility, and their configuration should not be overlooked. Many of the available vulnerability analysis tools focus largely on end host configuration and patching while ignoring the packets traversing the network. In addition, due to the nature of the protocols, a misconfiguration may be isolated to a small segment of the network, thus making it difficult to detect and correct.

David R Fletcher Jr., 6fletch9@gmail.com

The protocols identified in this paper represent several different opportunities for attack via varying methods. The script produced as a result of this paper provides an easy method to identify many of the protocols and vulnerable conditions discussed. This tool can be used on Microsoft Windows clients without the need to install third-party applications where rules of engagement prohibit this activity. In any case, this script should be useful to both network defenders and penetration testers for identification of network protocol based vulnerabilities.

5. Future Enhancements

The current version of the NetworkRecon.ps1 script is focused on the detection of dangerous protocols and conditions as supported by IPv4. While the IPv6 header is currently being parsed, there was not enough time to ensure that IPv6 capabilities were operating properly. Full IPv6 support should be attained to ensure that identification is uniform in all environments.

During development, no method of Ethernet frame collection was identified. This prevented parsing of LLDP, CDP, DTP, and VTP traffic. The presence of these protocols can still be identified through Invoke-NeighborCacheAnalysis. However, valuable attack opportunities are lost due to inability to parse this data.

David R Fletcher Jr., 6fletch9@gmail.com

References

Aboba, B., Thaler, D., & Esibov, L. (2007, January). RFC 4795 - Link-local Multicast Name Resolution (LLMNR). Retrieved February 20, 2017, from https://tools.ietf.org/html/rfc4795

Barroso, D., & Andres, A. (2005). Yersinia: Framework for Layer 2 Attacks. Retrieved February 20, 2017, from https://www.blackhat.com/presentations/bh-europe-05/BH_EU_05-Berrueta_Andres/BH_EU_05_Berrueta_Andres.pdf

Gaffié, L. (2013, January 24). Owning Windows Networks with Responder 1.7. Retrieved February 20, 2017, from https://www.trustwave.com/Resources/SpiderLabs-Blog/Owning-Windows-Networks-with-Responder-1-7/

IANA. (2016, November 17). Dynamic Host Configuration Protocol (DHCP) and Bootstrap Protocol (BOOTP) Parameters. Retrieved February 20, 2017, from http://www.iana.org/assignments/bootp-dhcp-parameters/bootp-dhcp-parameters.xhtml

IEEE. (2004, June 4). Media Access Control (MAC) Bridges. Retrieved February 20, 2017, from http://standards.ieee.org/getieee802/download/802.1D-2004.pdf

IEEE. (2009, September 17). Station and Media Access Control Connectivity Discovery. Retrieved February 20, 2017, from http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf

IETF. (1987, March). RFC 1002 - Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications. Retrieved February 20, 2017, from https://tools.ietf.org/html/rfc1002

IETF. (1987, March). RFC 1001 - Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods. Retrieved February 20, 2017, from https://tools.ietf.org/html/rfc1001

Li, T., Cole, B., Morton, P., & Li, D. (1998, March). Retrieved February 20, 2017, from https://www.ietf.org/rfc/rfc2281.txtMicrosoft. (2008, May 8). Managing Network Boot Programs. Retrieved February 20, 2017, from https://technet.microsoft.com/en-us/library/cc732351(v=ws.10).aspx

David R Fletcher Jr., 6fletch9@gmail.com

Microsoft. (2008, May 8). Managing Network Boot Programs. Retrieved February 20, 2017, from https://technet.microsoft.com/en-us/library/cc732351(v=ws.10).aspx

Microsoft. (2014, August 24). Windows Deployment Services Overview. Retrieved February 20, 2017, from https://technet.microsoft.com/en-us/library/hh831764(v=ws.11).aspx

Moy, J. (1998, April). OSPF Version 2. Retrieved February 20, 2017, from https://www.ietf.org/rfc/rfc2328.txt

Nadas, S., & Ericsson, E. (2010, March). RFC 5798 - Virtual Router Redundancy Protocol Version 3 for IPv4 and IPv6. Retrieved February 20, 2017, from https://tools.ietf.org/html/rfc5798Robertson, K. (2015, April 5).

Inveigh. Retrieved February 20, 2017, from https://github.com/Kevin-Robertson/Inveigh

Rouiller, S. A. (n.d.). Virtual LAN Security: Weaknesses and Countermeasures. Retrieved February 20, 2017, from https://www.sans.org/reading-room/whitepapers/networkdevs/virtual-lan-security-weaknesses-countermeasures-1090

Sternstein, J. (n.d.). Local Network Attacks: LLMNR and NBT-NS Poisoning - Stern Security. Retrieved February 20, 2017, from https://www.sternsecurity.com/blog/local-network-attacks-llmnr-and-nbt-ns-poisoning

Stevens, W. R., & Wright, G. R. (1994). TCP/IP Illustrated, Volume 1 (Vol. 1). Reading, MA: Addison-Wesley.

Stretch. (n.d.). Packet Life. Retrieved February 20, 2017, from http://packetlife.net/

Wireshark. (2008, April). NetBIOS/NBNS. Retrieved February 20, 2017, from https://wiki.wireshark.org/NetBIOS/NBNS

Wireshark. (2013, May 6). CDP. Retrieved February 20, 2017, from https://wiki.wireshark.org/CDP

Wireshark. (2014, September 5). LinkLayerDiscoveryProtocol. Retrieved February 20, 2017, from https://wiki.wireshark.org/LinkLayerDiscoveryProtocol

Wright, J. (2015). Network Attacks for Penetration Testers (A10_01 ed., Vol. 1). Bethesda, MD: SANS.

David R Fletcher Jr., 6fletch9@gmail.com

David R Fletcher Jr., 6fletch9@gmail.com

Appendix
NetworkRecon.ps1 Script

```
function Invoke-TraceCollect
{
<#
.SYNOPSIS

This module performs a network trace using PowerShell network tracing functionality.
After the trace is complete, the module will perform analysis based on user provided
arguments to determin whether potentially vulnerable traffic exists in the targeted
trace.

Function: Invoke-TraceCollect
Author: David Fletcher
License: BSD 3-Clause
Required Dependencies: User must be administrator to capture traffic.
Optional Dependencies: None

.DESCRIPTION

This module performs a network trace using PowerShell network tracing functionality.
After the trace is complete, the module will perform analysis based on user provided
arguments to determin whether potentially vulnerable traffic exists in the targeted
trace.

.PARAMETER Duration

This parameter is optional and will specify the duration, in minutes, that traffic
will be collected before analysis is performed. If no value is specified, then the
network trace will run for 5 minutes by default.

.PARAMETER Folder

This parameter is optional and will specify the folder where the packet capture will be
stored. This is useful if the user wants to export and convert the resulting event trace
file to .pcap format using Microsoft Message Analyzer. If no value is specified, then the
script will use the folder C:\temp

.PARAMETER File

This parameter is optional and will specify the file name used for the stored event trace
log.
If no value is specified, then the file will be named capture_[DateTime.ToString()].etl.

.PARAMETER Size

This parameter is optional and will specify the maximum size of the capture file. If no
value
is specified, then the system default will be used. This is usually 250 MB.

.EXAMPLE

C:\PS> Invoke-TraceCollect

Description
-----------
This invocation will execute a network event trace with default arguments (collect for 5
minutes, store the trace
at C:\temp\capture_[DateTime.ToString()].etl, and perform all checks.

.EXAMPLE

C:\PS> Invoke-TraceCollect -CaptureFolder "C:\Users\Test" -CaptureFile "capture.etl" -
Duration 10

Description
-----------
This invocation will execute a network event trace for 10 minutes saving the output to
"C:\Users\Test\capture.etl"
and perform allchecks

#>
Param(
 [Parameter(Position = 0, Mandatory = $false)]
 [string]
 $Folder = "C:\temp",
```

David R Fletcher Jr., 6fletch9@gmail.com

```
 [Parameter(Position = 1, Mandatory = $false)]
 [string]
 $Name = ("capture_" + (Get-Date).Year + (Get-Date).Month + (Get-Date).Day + (Get-
Date).Hour + (Get-Date).Minute + (Get-Date).Second),

 [Parameter(Position = 2, Mandatory = $false)]
 [int]
 $Duration = 5,

 [Parameter(Position = 3, Mandatory = $false)]
 [int]
 $Size = 250

)
If (-NOT ([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole(`
    [Security.Principal.WindowsBuiltInRole] "Administrator"))
{
    Write-Warning "Administrator rights are required in order to execute trace
collection.`nThis function uses standard Windows conventions which requires elevation."
    Break
}

# Check to see if the target folder exists
If ((Test-Path "$folder") -eq $false)
{
    $createFolder = ("cmd.exe /C mkdir " + $folder)
}

# Set the path to the output file
$seconds = $Duration * 60

# Start the session to begin collecting packets
Write-Host "[+] Starting capture session"

# Try running a PEF trace first.  If the PEF module is available it is capable
# of generating a cap file which can be consumed and analyzed with Wireshark.
# The PEF modules are only available with Windows 8 and above. If this fails
# then we fall back to running netsh trace to capture packet data using the Windows
# NDIS provider.
try
{
    Write-Host "   [-] Trying PEF trace first..."
    Import-Module PEF
    # We're using PEF so we can generate a cap file instead of etl
    $Path= ($Folder + "\" + $Name + ".cap")
    # Set up the session using the provided parameters. I have not found a way to specify
    # the maximum trace size, so the default of 250 MB is used. This should be plenty of
    # storage space for the resulting file.
    $session = New-PefTraceSession -Name $Name -Path $Path -SaveOnStop Linear -Force
    # Add the NDIS-PacketCapture provider to the session
    Add-PefMessageProvider -PEFSession $session -Provider "Microsoft-Windows-NDIS-
PacketCapture" > $null
    # TODO: Windows 10 Supports promiscuous mode by using Add-NetEventNetworkAdapter
commandlet.
    # Add support for this commandlet to ensure we are getting everything.
    # Create a TimeSpanTrigger to stop the capture.  Once the capture starts we lose
interactive control
    $trigger = New-PefTimeSpanTrigger -TimeSpan (New-TimeSpan -Seconds $seconds)
    # Output status messages to the user
    Write-Host "   [-] Successfully created PEF Trace Session..."
    Write-Host ("   [-] Output will be saved to " + $Path)
    Write-Host ("   [-] Trace will execute for " + $Duration + " minutes while packet
capture is running")
    # Assign the trigger event to the Stop-PefTraceSession commandlet
    Stop-PefTraceSession -PEFSession $session -Trigger $trigger > $null
    # Start the session. When the specified time has elapsed, the trace will stop
    Start-PefTraceSession -PEFSession $session > $null
    Write-Host "   [-] Session stopped"
    Write-Host "[+] Packet capture complete"
}
catch
{
    Write-Host "   [!] Unable to create PEF trace...falling back to netsh..."
    $Path= ($Folder + "\" + $Name + ".etl")
    Write-Host "   [-] Output will be saved to " $Path
    $traceCommand = ("netsh trace start provider=Microsoft-Windows-NDIS-PacketCapture
tracefile=" + $Path + " maxSize=" + $Size + " capture=yes overwrite=yes filemode=single")
    Invoke-Expression $traceCommand
```

David R Fletcher Jr., 6fletch9@gmail.com

```
    Write-Host ("   [-] Sleeping for " + $Duration + " minutes while packet capture is
running")
    Start-Sleep -s $seconds

    # Stop the session to cease packet collection
    Write-Host "[+] Packet capture complete"
    Write-Host "   [-] Stopping capture session"
    netsh trace stop
}

}

function Invoke-NeighborCacheAnalysis
{
<#
.SYNOPSIS

This module performs a check of the layer 2 cache on the local computer to determine
whether addresses of interest are cached. Given the frequency with which the
interesting protocols communicate, it is likely that the presence of these cached
entries identify that the host is able to observe these potentially vulnerable protocols.


Function: Invoke-NeighborCacheAnalysis
Author: David Fletcher
License: BSD 3-Clause
Required Dependencies: None
Optional Dependencies: None

.DESCRIPTION

This module performs a check of the layer 2 cache on the local computer to determine
whether addresses of interest are cached. Given the frequency with which the
interesting protocols communicate, it is likely that the presence of these cached
entries identify that the host is able to observe these potentially vulnerable protocols.

LLMNR could be a false-positive since it appears to be a static entry present on all
Windows hosts.


.EXAMPLE

C:\PS> Invoke-NeighborCacheAnalysis

Description
-----------
This invocation will inspect the layer 2 cache of each of the connected network adapters
and
identify whether multicast addresses for a given protocol are present.  If so, the output
reports the presence of the protocol and which OSI layer it was observed at.

#>
Param(
)

# Get the list of connected network adapters
# Ge-NetAdapter doesn't work in Windows 7
# See if we support Get-NetAdapter, if not, we have to use
# netsh output and parse results
$parseOld = $false
try
{
    $adapters = Get-NetAdapter
    $parseOld = $false
}
catch
{
   $adapters = Get-ParsedAdapterNames
   $parseOld = $true
}

    foreach ($adapter in $adapters)
    {
        if ($parseOld -eq $true)
        {
            $neighbors = Get-ParsedArpTables -InterfaceIndex $adapter.Name
        }
        else
        {
            $neighbors = Get-NetNeighbor -InterfaceAlias $adapter.Name
```

```
        }

        Write-Host ("[+] Checking Neighbor Entries for Known Protocol Addresses (" +
$adapter.Name + ")")
        foreach ($neighbor in $neighbors)
        {
            # Check for Known Ethernet Multicast Adddresses to Determine Potential
Exposed Protocols
            switch ($neighbor.LinkLayerAddress)
            {
                # Check for the CDP/VTP Multicast Address
                "01000ccccccc"
                {
                    Write-Host "   [-] Layer 2 CDP/VTP Address Found in Neighbor Cache"
                }
                # Check for the STP Multicast Address
                "0180c2000000"
                {
                    Write-Host "   [-] Layer 2 STP Address Found in Neighbor Cache"
                }
                # Check for the LLDP Multicast Addresses
                "0180c2000000"
                {
                    Write-Host "   [-] Layer 2 LLDP Address Found in Neighbor Cache"
                }
                "0180c2000003"
                {
                    Write-Host "   [-] Layer 2 LLDP Address Found in Neighbor Cache"
                }
                "0180c200000E"
                {
                    Write-Host "   [-] Layer 2 LLDP Address Found in Neighbor Cache"
                }
                # Check this one, it is listed as "All Routers" multicast group
                "01005e000002"
                {
                    Write-Host "   [-] Layer 2 HSRP Address Found in Neighbor Cache"
                }
                # Check for the OSPF HELLO Multicast Address
                "01005e000005"
                {
                    Write-Host "   [-] Layer 2 OSPF HELLO Address Found in Neighbor
Cache"
                }
                "333300000005"
                {
                    Write-Host "   [-] Layer 2 OSPF HELLO Address Found in Neighbor
Cache"
                }
                # Check for the OSPF DR Multicast Address
                "01005e000006"
                {
                    Write-Host "   [-] Layer 2 OSFP DR Address Found in Neighbor Cache"
                }
                "333300000006"
                {
                    Write-Host "   [-] Layer 2 OSPF DR Address Found in Neighbor Cache"
                }
                # Check for the VRRP Multicast Address
                "01005e000012"
                {
                    Write-Host "   [-] Layer 2 VRRP Address Found in Neighbor Cache"
                }
                # Check for the mDNS Multicast Address
                "01005e0000fb"
                {
                    Write-Host "   [-] Layer 2 mDNS Address Found in Neighbor Cache"
                }
                "33330000000fb"
                {
                    Write-Host "   [-] Layer 2 mDNS Address Found in Neighbor Cache"
                }
                # Check for the LLMNR Multicast Address
                "01005e0000fc"
                {
                    Write-Host "   [-] Layer 2 LLMNR Address Found in Neighbor Cache"
                }
                "333300000103"
                {
                    Write-Host "   [-] Layer 2 LLMNR Address Found in Neighbor Cache"
                }
```

```
                }
                # Check IP Addresses for Known IP Multicast
                switch ($neighbor.IPAddress)
                {
                    # Check for the IPv4 OSPF HELLO Multicast Address
                    "224.0.0.5"
                    {
                        Write-Host "   [-] IPv4 OSPF HELLO Address Found in Neighbor Cache"
                    }
                    # Check for the IPv4 OSPF DR Multicast Address
                    "224.0.0.6"
                    {
                        Write-Host "   [-] IPv4 OSFP DR Address Found in Neighbor Cache"
                    }
                    # Check for the IPv4 VRRP Multicast Address
                    "224.0.0.18"
                    {
                        Write-Host "   [-] IPv4 VRRP Address Found in Neighbor Cache"
                    }
                    # Check for the IPv4 mDNS Multicast Address
                    "224.0.0.251"
                    {
                        Write-Host "   [-] IPv4 mDNS Address Found in Neighbor Cache"
                    }
                    # Check for the IPv4 LLMNR Multicast Address
                    "224.0.0.252"
                    {
                        Write-Host "   [-] IPv4 LLMNR Address Found in Neighbor Cache"
                    }
                    # Check for the IPv6 OSPF HELLO Multicast Address
                    "ff02::5"
                    {
                        Write-Host "   [-] IPv6 OSPF HELLO Address Found in Neighbor Cache"
                    }
                    # Check for the IPv6 OSPF DR Multicast Address
                    "ff02::6"
                    {
                        Write-Host "   [-] IPv6 OSFP DR Address Found in Neighbor Cache"
                    }
                    # Check for the IPv6 LLMNR Multicast Address
                    "ff02::1:3"
                    {
                        Write-Host "   [-] IPv6 LLMNR Address Found in Neighbor Cache"
                    }
                    # Check for the IPv6 mDNS Multicast Address
                    "ff0x::fb"
                    {
                        Write-Host "   [-] IPv6 mDNS Address Found in Neighbor Cache"
                    }
                }
            }
        }
    }

function Get-ParsedAdapterNames
{
<#
.SYNOPSIS

This module simulates the behavior of the Get-NetNeighbor commandlet available
in Windows 8 and above. It does not return NetNeighbor objects. Only the information
(MAC and IP address) returned from the netsh commands that are used within the
functionality of the exposed commandlets in this package.

Function: Get-ParsedArpTables
Author: David Fletcher
License: BSD 3-Clause
Required Dependencies: User must be administrator to capture traffic.
Optional Dependencies: None

.DESCRIPTION

This module returns the MAC and IP addresses found within the output of the
following commands:

netsh int ipv4 show neigh interface=$InterfaceIndex
netsh int ipv6 show neigh interface=$InterfaceIndex

The results are returned in a PowerShell cusom object having the properties
LinkLayerAddress
and IPAddress which conforms with the results returned by Get-NetNeighbor.
```

David R Fletcher Jr., 6fletch9@gmail.com

```
.PARAMETER InterfaceIndex

This parameter is mandatory and identifies the interface for which arp table entries are
being parsed.
This can be the integer interface index or the string interface name. The latter is
generated by the
Get-ParsedAdapterNames function.

#>
    $cmdOutput = netsh int show int
    foreach ($line in $cmdOutput)
    {
        if (($line.Trim() -eq '') -or $line.Contains('Admin State') -or $line.Contains('-
--'))
        {
            # The first line in the output is null, so skip it
            # The second line in the output is the table header, so skip it
            continue
        }
        else
        {
            $elements = ($line -replace " {2,}"," ").Split(' ')
            $adapter = @{}
            $adapter.Name = $elements[3]
            Write-Output $adapter
        }
    }
}

function Get-ParsedArpTables
{
<#
.SYNOPSIS

This module simulates the behavior of the Get-NetNeighbor commandlet available
in Windows 8 and above. It does not return NetNeighbor objects. Only the information
(MAC and IP address) returned from the netsh commands that are used within the
functionality of the exposed commandlets in this package.

Function: Get-ParsedArpTables
Author: David Fletcher
License: BSD 3-Clause
Required Dependencies: User must be administrator to capture traffic.
Optional Dependencies: None

.DESCRIPTION

This module returns the MAC and IP addresses found within the output of the
following commands:

netsh int ipv4 show neigh interface=$InterfaceIndex
netsh int ipv6 show neigh interface=$InterfaceIndex

The results are returned in a PowerShell cusom object having the properties
LinkLayerAddress
and IPAddress which conforms with the results returned by Get-NetNeighbor.

.PARAMETER InterfaceIndex

This parameter is mandatory and identifies the interface for which arp table entries are
being parsed.
This can be the integer interface index or the string interface name. The latter is
generated by the
Get-ParsedAdapterNames function.

#>
Param(
 [Parameter(Position = 0, Mandatory = $true)]
 [string]
 $InterfaceIndex
)
    # Array of netsh commands to retrieve the arp cache entries for the local computer
    $commands = ("netsh int ipv4 show neigh interface=" + $InterfaceIndex),("netsh int
ipv6 show neigh interface=" + $InterfaceIndex)

    # Process each command and process the resulting output
    foreach ($command in $commands)
    {
        # Exectute the command expression and save the results
        $cmdOutput = Invoke-Expression $command
```

```
        # Process each line of output
        foreach ($line in $cmdOutput)
        {
            # Throw away unnecessary header information
            if (($line.Trim() -eq '') -or $line.Contains('Internet Address') -or
$line.Contains('---') -or $line.Contains($InterfaceIndex))
            {
                # The first line in the output is null, so skip it
                # The second line in the output is the table header, so skip it
                continue
            }
            else
            {
                # This output is space delimited but the space count is asymmetric so we
need to normalize the input
                # Here we are replacing 2 or more spaces with a single space then
splitting the result on the single space
                $elements = ($line -replace " {2,}"," ").Split(' ')

                # Create our output object to place on the pipeline
                $neighbor = @{}
                $neighbor.IPAddress = $elements[0]
                # Change the format of the MAC address to match the output of Get-
NetNeighbor
                $neighbor.LinkLayerAddress = $elements[1].Replace('-','').ToLower()

                # Write the output to the pipeline
                Write-Output $neighbor
            }
        }
    }
}

function Invoke-LiveAnalysis
{
<#
.SYNOPSIS

This module performs a network trace using PowerShell network tracing functionality.
After the trace is complete, the module will perform analysis based on user provided
arguments to determin whether potentially vulnerable traffic exists in the targeted
trace.

This module performs live analysis of network traffic observable by the host computer.
This
module can be used to confirm or augment the results returned by Invoke-
NeighborCacheAnalysis.
Unlike, Invoke-NeighborCacheAnalysis, this module will detect DHCP and NBNS traffic and
can parse
details from other protocols but cannot identify cdp/dtp/vtp or other layer 2 only
protocols.

This module borrows heavily from the sniffer module implemented in the Invoke-Inveigh
module but
currently uses this functionality to implement identify and parse capabilities. Future
enhancements
may include the ability to attack the network through information disclosure, route
manipulation,
malicious boot and other attacks currently provided by tools that are predominately
linux.

Function: Invoke-LiveAnalysis
Author: David Fletcher
License: BSD 3-Clause
Required Dependencies: User must be administrator to capture traffic.
Optional Dependencies: None

.DESCRIPTION

This module performs live analysis of network traffic observable by the host computer.
This
module can be used to confirm or augment the results returned by Invoke-
NeighborCacheAnalysis.
Unlike, Invoke-NeighborCacheAnalysis, this module will detect DHCP and NBNS traffic and
can parse
details from other protocols but cannot identify cdp/dtp/vtp or other layer 2 only
protocols.

.EXAMPLE
```

```
C:\PS> Invoke-LiveAnalysis

Description
-----------
This invocation will execute live network analysis with all default parameters (console
output provided, no log file, infinite duration).

#>
Param(
)

# Get the IP Address of the network interface
# This may need to be changed to support a computer with multiple interfaces
if(!$IP)
{
    $IP = (Test-Connection 127.0.0.1 -count 1 | Select-Object -ExpandProperty
Ipv4Address)
}

if(!$analyzer)
{
    $global:analyzer = [HashTable]::Synchronized(@{})
    $analyzer.console_queue = New-Object System.Collections.ArrayList
    $analyzer.show_dhcp = $true
    $analyzer.show_hsrp = $true
    $analyzer.show_llmnr = $true
    $analyzer.show_mdns = $true
    $analyzer.show_nbns = $true
    $analyzer.show_ospf = $true
    $analyzer.show_vrrp = $true
    $analyzer.rule_name = "Multicast Inbound Allow"
}

$analyzer.sniffer_socket = $null
$analyzer.running = $true

$analyzer.console_queue.Add("Analyzer started at $(Get-Date -format 's')")  > $null

$firewall_status = netsh advfirewall show allprofiles state | Where-Object {$_ -match
'ON'}

if($firewall_status)
{
    $analyzer.console_queue.Add("Windows Firewall = Enabled")  > $null
    $firewall_rules = New-Object -comObject HNetCfg.FwPolicy2
    $firewall_powershell = $firewall_rules.rules | Where-Object {$_.Enabled -eq $true -
and $_.Direction -eq 1} |Select-Object -Property Name | Select-String "Windows
PowerShell}"

    if($firewall_powershell)
    {
        $analyzer.console_queue.Add("Windows Firewall - PowerShell.exe = Allowed")  >
$null
    }

    # The Windows firewall does not allow inbound multicast packets by default. As a
result, if the firewall
    # is enabled we won't be able to check for some of the interesting protocols.
Therefore, we can either
    # attempt to disable the firewall using
    # netsh advfirewall set allprofiles state off < This increases our exposure to
attack. We only want to see inbound traffic
    # a better option is to allow the multicast addresses we're interested in inbound
    # netsh advfirewall firewall add rule name="Multicast Inbound Allow" dir=in
action=allow localip="224.0.0.0/24"
    $analyzer.console_queue.Add("Inserted Inbound Multicast Rule") > $null
    netsh advfirewall firewall add rule name="Multicast Inbound Allow" dir=in
action=allow localip="224.0.0.0/24"
}

$analyzer.console_queue.Add("Listening IP Address = $IP")  > $null

# Begin ScriptBlocks

# Shared Basic Functions ScriptBlock
$shared_basic_functions_scriptblock =
{

    function DataToUInt16($field)
    {
            [Array]::Reverse($field)
```

David R Fletcher Jr., 6fletch9@gmail.com

```
            return [System.BitConverter]::ToUInt16($field,0)
    }

    function DataToUInt32($field)
    {
            [Array]::Reverse($field)
            return [System.BitConverter]::ToUInt32($field,0)
    }

    function DataLength2
    {
        param ([Int]$length_start,[Byte[]]$string_extract_data)

        $string_length =
[System.BitConverter]::ToUInt16($string_extract_data[$length_start..($length_start +
1)],0)
        return $string_length
    }

    function DataLength4
    {
        param ([Int]$length_start,[Byte[]]$string_extract_data)

        $string_length =
[System.BitConverter]::ToUInt32($string_extract_data[$length_start..($length_start +
3)],0)
        return $string_length
    }

    function DataToString
    {
        param ([Int]$string_start,[Int]$string_length,[Byte[]]$string_extract_data)

        $string_data =
[System.BitConverter]::ToString($string_extract_data[$string_start..($string_start +
$string_length - 1)])
        $string_data = $string_data -replace "-00",""
        $string_data = $string_data.Split("-") | ForEach-
Object{[Char][System.Convert]::ToInt16($_,16)}
        $string_extract = New-Object System.String ($string_data,0,$string_data.Length)
        return $string_extract
    }
    function DataToHexString
    {
        param ([Int]$string_start,[Int]$string_length,[Byte[]]$string_extract_data)

        $string_data =
[System.BitConverter]::ToString($string_extract_data[$string_start..($string_start +
$string_length - 1)])
        $string_data = $string_data -replace "-",""
        $string_extract = New-Object System.String ($string_data,0,$string_data.Length)
        return $string_extract.ToLower()
    }

}

$sniffer_scriptblock =
{
    param ($IP,$RunTime)

    $byte_in = New-Object System.Byte[] 4
    $byte_out = New-Object System.Byte[] 4
    $byte_data = New-Object System.Byte[] 4096
    $byte_in[0] = 1
    $byte_in[1-3] = 0
    $byte_out[0] = 1
    $byte_out[1-3] = 0
    $analyzer.sniffer_socket = New-Object
System.Net.Sockets.Socket([Net.Sockets.AddressFamily]::InterNetwork,[Net.Sockets.SocketTy
pe]::Raw,[Net.Sockets.ProtocolType]::IP)
    $analyzer.sniffer_socket.SetSocketOption("IP","HeaderIncluded",$true)
    $analyzer.sniffer_socket.ReceiveBufferSize = 1024
    $end_point = New-Object System.Net.IPEndpoint([System.Net.IPAddress]"$IP",0)
    $analyzer.sniffer_socket.Bind($end_point)

$analyzer.sniffer_socket.IOControl([System.Net.Sockets.IOControlCode]::ReceiveAll,$byte_i
n,$byte_out)

    while($analyzer.running)
    {
        # Inveigh sniffer is only configured to parse IPv4 Packets
```

David R Fletcher Jr., 6fletch9@gmail.com

```
        $packet_data =
$analyzer.sniffer_socket.Receive($byte_data,0,$byte_data.Length,[System.Net.Sockets.Socke
tFlags]::None)
        $memory_stream = New-Object System.IO.MemoryStream($byte_data,0,$packet_data)
        $binary_reader = New-Object System.IO.BinaryReader($memory_stream)
        $version_more = $binary_reader.ReadByte()
        $IP_version = [Int]"0x$(('{0:X}' -f $version_more)[0])"


        if ($IP_version -eq 4)
        {
            # Process the IPv4 Header
            $header_length = [Int]"0x$(('{0:X}' -f $version_more)[1])" * 4
            $type_of_service= $binary_reader.ReadByte()
            $total_length = DataToUInt16 $binary_reader.ReadBytes(2)
            $identification = $binary_reader.ReadBytes(2)
            $flags_offset = $binary_reader.ReadBytes(2)
            $TTL = $binary_reader.ReadByte()
            $protocol_number = $binary_reader.ReadByte()
            $header_checksum =
[System.Net.IPAddress]::NetworkToHostOrder($binary_reader.ReadInt16())
            $source_IP_bytes = $binary_reader.ReadBytes(4)
            $source_IP = [System.Net.IPAddress]$source_IP_bytes
            $destination_IP_bytes = $binary_reader.ReadBytes(4)
            $destination_IP = [System.Net.IPAddress]$destination_IP_bytes
         }
         elseif ($IP_version -eq 6)
         {
            # Process the IPv6 Header
            # Intially, we won't process traffic class and flow label
            # since they aren't needed for analysis
            $traffic_high = 0 # Get low order nibble from $version_more
            $traffic_flow = $binary_reader.ReadBytes(3)
            $traffic_low = 0 # Get high order nibble from $traffic_flow
            $flow_label = 0 # Zero out 4 high order bits from $traffic_flow
            $total_length = DataToUInt16 $binary_reader.ReadBytes(2)
            # This is next header but we may not need to do anything with this
            # depending on whether additional headers are typically seen in the
            # protocols we are interested in. May be useful to report this value
            # for debugging purposes. If the protocols of interest have several
            # extension headers, it may be useful to have a function dedicated to
            # IPv6 next header chain walking to deteremine if one of the interesting
            # protocols is present. Will test with IPv6.
            $protocol_number= $binary_reader.ReadByte()
            $TTL = $binary_Reader.ReadByte()
            $source_IP_bytes = $binary_reader.ReadBytes(16)
            $source_IP = [System.Net.IPAddress]$source_IP_bytes
            $destination_IP_bytes = $binary_reader.ReadBytes(16)
            $destination_IP = [System.Net.IPAddress]$destination_IP_bytes
         }
         else
         {
            continue
         }

        # Packet processing starts here. The flow consists of inspecting the embedded
protocol number first
        # OSPF and VRRP do not use standard protocol numbers (TCP and UDP). Then we will
inspect the specific protocol further
        switch ($protocol_number)
        {
            # TCP Processing
            6
            {
                $source_port = DataToUInt16 $binary_reader.ReadBytes(2)
                $destination_port = DataToUInt16 $binary_reader.ReadBytes(2)
                $sequence_number = DataToUInt32 $binary_reader.ReadBytes(4)
                $ack_number = DataToUInt32 $binary_reader.ReadBytes(12)
                $TCP_header_length = [Int]"0x$(('{0:X}' -f
$binary_reader.ReadByte())[0])" * 4
                $TCP_flags = $binary_reader.ReadByte()
                $TCP_window = DataToUInt16 $binary_reader.ReadBytes(2)
                $TCP_checksum =
[System.Net.IPAddress]::NetworkToHostOrder($binary_reader.ReadInt16())
                $TCP_urgent_pointer = DataToUInt16 $binary_reader.ReadBytes(2)
                $payload_bytes = $binary_reader.ReadBytes($total_length - ($header_length
+ $TCP_header_length))

            }
            # UDP Processing
            17
```

David R Fletcher Jr., 6fletch9@gmail.com

```
            {
                $source_port = $binary_reader.ReadBytes(2)
                $endpoint_source_port = DataToUInt16 ($source_port)
                $destination_port = DataToUInt16 $binary_reader.ReadBytes(2)
                $UDP_length = $binary_reader.ReadBytes(2)
                $UDP_length_uint  = DataToUInt16 ($UDP_length)
                $binary_reader.ReadBytes(2)

                switch ($destination_port)
                {
                    # DHCP Packet/Options Inspection
                    68
                    {
                        if ($analyzer.show_dhcp)
                        {
                            $dhcp_opcode = $binary_reader.ReadByte()

                            # We are only interested in DHCP Responses which may contain
                            # a boot file location which we may be able to use for boot
                            # image analysis or malicious boot attack
                            if ($dhcp_opcode -eq 2)
                            {
                                $analyzer.console_queue.Add("DHCP response received from
" + $source_IP.ToString()) > $null

                                # Parse the remainder of the packet
                                $dhcp_hwtype = $binary_reader.ReadByte()
                                $dhcp_hwaddlength = $binary_reader.ReadByte()
                                $dhcp_hopcount = $binary_reader.ReadByte()
                                $dhcp_trans_id_bytes = $binary_reader.ReadBytes(4)
                                $dhcp_trans_id = DataToUInt32 $dhcp_trans_id_bytes
                                $dhcp_lease_duration = DataToUInt16
$binary_reader.ReadBytes(2)
                                $dhcp_flags = DataToUInt16 $binary_reader.ReadBytes(2)
                                $dhcp_client_ip_bytes = $binary_Reader.ReadBytes(4)
                                $dhcp_sender_ip_bytes = $binary_reader.ReadBytes(4)
                                $dhcp_server_ip_bytes = $binary_reader.ReadBytes(4)
                                $dhcp_server_ip = [System.Net.IPAddress]
$dhcp_server_ip_bytes
                                $dhcp_gateway_ip_bytes = $binary_reader.ReadBytes(4)
                                $dhcp_client_hw_addr_bytes = $binary_reader.ReadBytes(6)
                                $dhcp_client_hw_addr_padding =
$binary_reader.ReadBytes(10)
                                $dhcp_server_hostname_bytes =
$binary_reader.ReadBytes(64)
                                $dhcp_server_hostname_bytes = DataToString
$dhcp_server_hostname_bytes
                                $dhcp_server_boot_filename_bytes =
$binary_reader.ReadBytes(128)
                                $dhcp_server_boot_filename = DataToString
$dhcp_server_boot_filename_bytes

                                if ($dhcp_server_ip.Trim() -ne "")
                                {
                                    $analyzer.console_queue.Add(" [i] DHCP Server IP: " +
$dhcp_server_ip) > $null
                                }

                                if ($dhcp_server_hostname.Trim() -ne "")
                                {
                                    $analyzer.console_queue.Add(" [i] DHCP Server Name: "
+ $dhcp_server_hostname) > $null
                                }

                                if ($dhcp_server_boot_filename.Trim() -ne "")
                                {
                                    $analyzer.console_queue.Add(" [!] Boot File: " +
$dhcp_server_boot_filename) > $null
                                    $analyzer.console_queue.Add(" [!] This File Could
Contain Credentials") > $null
                                }

                                $dhcp_cookie_bytes = $binary_reader.ReadBytes(4)

                                # Process DHCP Options
                                $dhcp_option = $binary_reader.ReadByte()

                                # DHCP Option 255 signifies "End Of Options"
                                while ($dhcp_option -ne 255)
                                {
                                    # Process padding bytes
```

```
                                    switch ($dhcp_option)
                                    {
                                        # Handle Padding
                                        0
                                        {
                                            $dhcp_option = $binary_reader.ReadByte()
                                            continue
                                        }
                                        # Handle Standard PXE/Network Boot
                                        66
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $tftp_server_name = DataToString
$dhcp_option_bytes
                                            $analyzer.console_queue.Add(" [!] TFTP Server
Name: " + $tftp_server_name) > $null
                                        }
                                        67
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $tftp_boot_filename = DataToString
$dhcp_option_bytes
                                            $analyzer.console_queue.Add(" [!] TFTP Boot
Filename: " + $tftp_boot_filename) > $null
                                            $analyzer.console_queue.Add(" [!] This File
Could Contain Credentials") > $null
                                        }
                                        128
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $tftp_server_ip =
[System.Net.IPAddress]$dhcp_option_bytes
                                            $analyzer.console_queue.Add(" [!] TFTP Server
IP: " + $tftp_server_ip) > $null
                                        }
                                        150
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $tftp_server_ip =
[System.Net.IPAddress]$dhcp_option_bytes
                                            $analyzer.console_queue.Add(" [!] TFTP Server
IP: " + $tftp_server_ip) > $null
                                        }
                                        # Handle PXELINUX Requests
                                        208
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $analyzer.console_queue.Add(" [!] PXELINUX
Magic Option Observed") > $null
                                        }
                                        209
                                        {
                                            $dhcp_option_length =
$binary_reader.ReadByte()
                                            $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                            $pxelinux_config = DataToString
$dhcp_option_bytes
                                            $analyzer.console_queue.Add(" [!] PXELINUX
Config: " + $pxelinux_config) > $null
                                            $analyzer.console_queue.Add(" [!] This File
Should Be Inspected") > $null
                                        }
                                        210
                                        {
```

```
                                        $dhcp_option_length =
$binary_reader.ReadByte()
                                        $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                        $pxelinux_path_prefix = DataToString
$dhcp_option_bytes
                                        $analyzer.console_queue.Add(" [!] PXELINUX
Prefix: " + $pxelinux_path_prefix) > $null
                                    }
                                    # Handle All Others
                                    default
                                    {
                                        $dhcp_option_length =
$binary_reader.ReadByte()
                                        $dhcp_option_bytes =
$binary_reader.ReadBytes($dhcp_option_length)
                                        $analyzer.console_queue.Add(" [i] Observed
DHCP Option: " + $dhcp_option.ToString()) > $null
                                        $dhcp_option = $binary_reader.ReadByte()
                                        continue
                                    }
                                }
                            }
                        }
                    }
                }
                # NBNS Packet Inspection
                137
                {
                    if ($analyzer.show_nbns)
                    {
                        $analyzer.console_queue.Add("NBNS packet received from " +
$source_IP.ToString()) > $null
                        $nbns_queryid = DataToUInt16 $binary_reader.ReadBytes(2)
                        $nbns_control = $binary_reader.ReadByte()
                        # split the control field so we can tell if this is query or
response
                        $nbns_control_high = [Int]"0x$(('{0:X}' -f
$nbns_version_type)[0])"
                        $nbns_control_low = [Int]"0x$(('{0:X}' -f
$nbns_version_type)[1])"
                        $nbns_rcode = $binary_reader.ReadByte()
                        $nbns_qdcount = DataToUInt16 $binary_reader.ReadBytes(2)
                        $nbns_ancount = DataToUInt16 $binary_reader.ReadBytes(2)
                        $nbns_nscount = DataToUInt16 $binary_reader.ReadBytes(2)
                        $nbns_arcount = DataToUInt16 $binary_reader.ReadBytes(2)

                        if ($nbns_control_high -lt 8)
                        {
                            $analyzer.console_queue.Add(" [!] Potential for NBNS
Poisoning Attack") > $null
                            $analyzer.console_queue.Add(" [i] Type: Query") > $null
                            $analyzer.console_queue.Add(" [i] Query Count: " +
$nbns_qdcount.ToString()) > $null

                            for ($i = 1; $i -le $nbns_qdcount; $i++)
                            {
                                $nbns_field_length = $binary_reader.ReadByte()
                                $nbns_name = ""

                                while ($nbns_field_length -ne 0)
                                {
                                    $nbns_field_value_bytes =
$binary_reader.ReadBytes($nbns_field_length - 2)
                                    $nbns_query_suffix =
[System.BitConverter]::ToString($binary_reader.ReadBytes(2))
                                    # Used NBNS Name decoding code from Inveigh.ps1
below
                                    $nbns_query =
[System.BitConverter]::ToString($nbns_field_value_bytes)
                                    $nbns_query = $nbns_query -replace "-00",""
                                    $nbns_query = $nbns_query.Split("-") | ForEach-
Object{[Char][System.Convert]::ToInt16($_,16)}
                                    $nbns_query_string_encoded = New-Object
System.String ($nbns_query,0,$nbns_query.Length)
                                    $nbns_query_string_encoded =
$nbns_query_string_encoded.Substring(0,$nbns_query_string_encoded.IndexOf("CA"))
                                    $nbns_query_string_subtracted = ""
                                    $nbns_query_string = ""
                                    $n = 0
```

```
                                        do
                                        {
                                            $nbns_query_string_sub =
(([Byte][Char]($nbns_query_string_encoded.Substring($n,1))) - 65)
                                            $nbns_query_string_subtracted +=
([System.Convert]::ToString($nbns_query_string_sub,16))
                                            $n += 1
                                        }
                                        until($n -gt ($nbns_query_string_encoded.Length -
1))

                                        $n = 0

                                        do
                                        {
                                            $nbns_query_string +=
([Char]([System.Convert]::ToInt16($nbns_query_string_subtracted.Substring($n,2),16)))
                                        $n += 2
                                        }
                                        until($n -gt
($nbns_query_string_subtracted.Length - 1) -or $nbns_query_string.Length -eq 15)
                                        # Name Conversion is complete

                                        $nbns_name = $nbns_name + $nbns_query_string

                                        # Read Next Length for Loop Execution, for NBNS
there should only be one record
                                        $nbns_field_length = $binary_reader.ReadByte()

                                        if ($nbns_field_length -ne 0)
                                        {
                                            $nbns_name = ($nbns_name + ".")
                                        }

                                        switch ($nbns_query_suffix)
                                        {
                                            '41-41'
                                            {
                                                $nbns_service = "Workstation/Redirector"
                                            }
                                            '41-44'
                                            {
                                                $nbns_service = "Messenger"
                                            }
                                            '43-47'
                                            {
                                                $nbns_service = "Remote Access"
                                            }
                                            '43-41'
                                            {
                                                $nbns_service = "Server"
                                            }
                                            '43-42'
                                            {
                                                $nbns_service = "Remote Access Client"
                                            }
                                            '42-4C'
                                            {
                                                $nbns_service = "Domain Master Browser"
                                            }
                                            '42-4D'
                                            {
                                                $nbns_service = "Domain Controllers"
                                            }
                                            '42-4E'
                                            {
                                                $nbns_service = "Master Browser"
                                            }
                                            '42-4F'
                                            {
                                                $nbns_service = "Browser Election"
                                            }
                                        }
                                    }

                                    $nbns_record_type = DataToUInt16
$binary_reader.ReadBytes(2)
                                    $nbns_record_class = DataToUInt16
$binary_reader.ReadBytes(2)
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    $analyzer.console_queue.Add(" [i] Host: " +
$nbns_name) > $null
                                    $analyzer.console_queue.Add(" [i] Service Type: " +
$nbns_service) > $null
                                }
                            }
                            else
                            {
                                $analyzer.console_queue.Add(" [i] Type: Response") >
$null
                                $analyzer.console_queue.Add(" [i] Response Count: " +
$nbns_ancount.ToString()) > $null
                                # May Parse NBNS Responses Further In The Future
                            }
                        }
                    }
                    # HSRP Packet Inspection
                    1985
                    {
                        if ($analyzer.show_hsrp)
                        {
                            # This is for HSRP v0/1. HSRP v2 uses multicast IP
224.0.0.102
                            # HSRP destination should be 224.0.0.2
                            if ($destination_IP.ToString() -eq "224.0.0.2")
                            {
                                $hsrp_version = $binary_reader.ReadByte()
                                $hsrp_opcode = $binary_reader.ReadByte()
                                $hsrp_state = $binary_reader.ReadByte()
                                $hsrp_hellotime = $binary_reader.ReadByte()
                                $hsrp_holdtime = $binary_reader.ReadByte()
                                $hsrp_priority = $binary_reader.ReadByte()
                                $hsrp_group = $binary_reader.ReadByte()
                                $hsrp_reserved = $binary_reader.ReadByte()
                                $hsrp_auth_bytes = $binary_reader.ReadBytes(8)
                                $hsrp_auth = DataToString 0 8 $hsrp_auth_bytes
                                $hsrp_groupip_bytes = $binary_reader.ReadBytes(4)
                                $hsrp_groupip = [System.Net.IPAddress]
$hsrp_groupip_bytes

                                $analyzer.console_queue.Add("HSRP v" +
$hsrp_version.ToString() + " Packet Observed from " + $source_IP.ToString()) > $null

                                switch ($hsrp_opcode)
                                {
                                    0
                                    {
                                        $analyzer.console_queue.Add(" [i] Operation:
Hello") > $null
                                        $analyzer.console_queue.Add(" [i] Hello Time: " +
$hsrp_hellotime.ToString() + " seconds") > $null
                                        $analyzer.console_queue.Add(" [i] Hold Time: " +
$hsrp_holdtime.ToString() + " seconds") > $null
                                    }
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Operation:
Coup") > $null
                                    }
                                    2
                                    {
                                        $analyzer.console_queue.Add(" [i] Operation:
Resign") > $null
                                    }
                                }

                                switch ($hsrp_state)
                                {
                                    0
                                    {
                                        $analyzer.console_queue.Add(" [i] State:
Initial") > $null
                                    }
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] State: Learn")
> $null
                                    }
                                    2
                                    {
```

```
                                    $analyzer.console_queue.Add(" [i] State: Listen")
> $null
                                }
                                4
                                {
                                    $analyzer.console_queue.Add(" [i] State: Speak")
> $null
                                }
                                8
                                {
                                    $analyzer.console_queue.Add(" [i] State:
Standby") > $null
                                }
                                16
                                {
                                    $analyzer.console_queue.Add(" [i] State: Active")
> $null
                                }
                            }

                            $analyzer.console_queue.Add(" [i] Priority: " +
$hsrp_priority.ToString()) > $null
                            if ($hsrp_priority -lt 250)
                            {
                                $analyzer.console_queue.Add(" [!] Priority May Be
Low. Potential for Hijacking")
                            }

                            $analyzer.console_queue.Add(" [i] Group: " +
$hsrp_group.ToString()) > $null
                            $analyzer.console_queue.Add(" [!] Password: " +
$hsrp_auth) > $null
                            $analyzer.console_queue.Add(" [i] Group IP: " +
$hsrp_groupip.ToString()) > $null
                            }
                            else
                            {
                                $analyzer.console_queue.Add("Packet received on HSRP UDP
Port with wrong destination address") > $null
                            }
                        }
                    }
                    # mDNS Packet Inspection
                    5353
                    {
                        if ($analyzer.show_mdns)
                        {

                            # Need to gather full payload up front because of DNS
compression
                            $payload_bytes = $binary_reader.ReadBytes(($UDP_length_uint -
2) * 4)

                            # mDNS destination should be 224.0.0.251
                            if ($destination_IP.ToString() -eq "224.0.0.251")
                            {
                                $analyzer.console_queue.Add("mDNS Packet Observed from "
+ $source_IP.ToString()) > $null
                                $mdns_queryid = DataToUInt16 $payload_bytes[0..1]
                                $mdns_control = $payload_bytes[2]
                                # split the control field so we can tell if this is query
or response
                                $mdns_control_high = [Int]"0x$(('{0:X}' -f
$mdns_control)[0])"
                                $mdns_control_low = [Int]"0x$(('{0:X}' -f
$mdns_version_type)[1])"
                                $mdns_rcode = $payload_bytes[3]
                                $mdns_qdcount = DataToUInt16 $payload_bytes[4..5]
                                $mdns_ancount = DataToUInt16 $payload_bytes[6..7]
                                $mdns_nscount = DataToUInt16 $payload_bytes[8..9]
                                $mdns_arcount = DataToUInt16 $payload_bytes[10.11]

                                if ($mdns_control_high -lt 8)
                                {
                                    $analyzer.console_queue.Add(" [!] Potential for mDNS
Cache Poisoning Attack") > $null
                                    $analyzer.console_queue.Add(" [i] Type: Query") >
$null
                                    $analyzer.console_queue.Add(" [i] Count: " +
$mdns_qdcount.ToString()) > $null
                                    $payload_index = 12
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    for ($i = 1; $i -le $mdns_qdcount; $i++)
                                    {
                                        $mdns_field_length =
$payload_bytes[$payload_index]
                                        $payload_index = $payload_index + 1
                                        $name = ""
                                        while ($mdns_field_length -ne 0)
                                        {
                                            $mdns_field_value_bytes =
$payload_bytes[$payload_index..($payload_index + $mdns_field_length - 1)]
                                            $payload_index = $payload_index +
$mdns_field_length
                                            $mdns_field_value = DataToString 0
$mdns_field_length $mdns_field_value_bytes
                                            $name = $name + $mdns_field_value
                                            $mdns_field_length =
$payload_bytes[$payload_index]
                                            $payload_index = $payload_index + 1
                                            # When DNS Compression is in use, the record
will not be terminated with a null
                                            # Instead, a byte value of 192 (or C0) will
be found indicating that the next byte
                                            # represents the offset into the DNS packet
where the request/response continues.
                                            if ($mdns_field_length -eq 192)
                                            {
                                                $mdns_ptr_offset =
$payload_bytes[$payload_index]
                                                $payload_index = $payload_index + 1
                                                $mdns_field_length =
$payload_bytes[$mdns_ptr_offset]
                                                $mdns_ptr_offset = $mdns_ptr_offset + 1
                                                while ($mdns_field_length -ne 0)
                                                {
                                                    $mdns_field_value_bytes =
$payload_bytes[$mdns_ptr_offset..($mdns_ptr_offset + $mdns_field_length - 1)]
                                                    $mdns_ptr_offset = $mdns_ptr_offset +
$mdns_field_length
                                                    $mdns_field_value = DataToString 0
$mdns_field_length $mdns_field_value_bytes
                                                    $name = $name + $mdns_field_value
                                                    $mdns_field_length =
$payload_bytes[$mdns_ptr_offset]
                                                    $mdns_ptr_offset = $mdns_ptr_offset +
1
                                                    if ($mdns_field_length -ne 0)
                                                    {
                                                        $name = ($name + ".")
                                                    }
                                                }
                                                break
                                            }
                                            if ($mdns_field_length -ne 0)
                                            {
                                                $name = ($name + ".")
                                            }
                                        }
                                        $mdns_record_type =
$payload_bytes[$payload_index..($payload_index + 1)]
                                        $payload_index = $payload_index + 2
                                        $mdns_record_class =
$payload_bytes[$payload_index..($payload_index + 1)]
                                        $payload_index = $payload_index + 2
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    $analyzer.console_queue.Add(" [i] Host: " +
$name) > $null
                                }
                            }
                            else
                            {
                                $analyzer.console_queue.Add(" [i] Type: Response") >
$null
                                $analyzer.console_queue.Add(" [i] Count: " +
$mdns_ancount.ToString()) > $null
                                # May Parse mDNS Responses Further In The Future
                            }
                        }
                        else
                        {
                            $analyzer.console_queue.Add("Packet received on mDNS UDP
Port with wrong destination address") > $null
                        }
                    }
                }
                # LLMNR Packet Inspection
                5355
                {
                    if ($analyzer.show_llmnr)
                    {
                        if ($destination_IP.ToString() -eq "224.0.0.252")
                        {
                            $analyzer.console_queue.Add("LLMNR Packet Observed from "
+ $source_IP.ToString()) > $null
                            $llmnr_queryid = DataToUInt16 $payload_bytes[0..1]
                            llmnr_control = $payload_bytes[2]
                            # split the control field so we can tell if this is query
or response
                            $llmnr_control_high = [Int]"0x$(('{0:X}' -f
$llmnr_control)[0])"
                            $llmnr_control_low = [Int]"0x$(('{0:X}' -f
$llmnr_version_type)[1])"
                            $llmnr_rcode = $payload_bytes[3]
                            $llmnr_qdcount = DataToUInt16 $payload_bytes[4..5]
                            $llmnr_ancount = DataToUInt16 $payload_bytes[6..7]
                            $llmnr_nscount = DataToUInt16 $payload_bytes[8..9]
                            $llmnr_arcount = DataToUInt16 $payload_bytes[10.11]

                            if ($llmnr_control_high -lt 8)
                            {
                                $analyzer.console_queue.Add(" [!] Potential for LLMNR
Cache Poisoning Attack") > $null
                                $analyzer.console_queue.Add(" [i] Type: Query") >
$null
                                $analyzer.console_queue.Add(" [i] Count: " +
$llmnr_qdcount.ToString()) > $null
                                $payload_index = 12

                                for ($i = 1; $i -le $llmnr_qdcount; $i++)
                                {

                                    $llmnr_field_length =
$payload_bytes[$payload_index]
                                    $payload_index = $payload_index + 1

                                    $name = ""

                                    while ($llmnr_field_length -ne 0)
                                    {
                                        $llmnr_field_value_bytes =
$payload_bytes[$payload_index..($payload_index + $llmnr_field_length - 1)]
                                        $payload_index = $payload_index +
$llmnr_field_length

                                        $llmrn_field_value = DataToString 0
$mdns_field_length $llmnr_field_value_bytes

                                        $name = $name + $llmnr_field_value

                                        $llmnr_field_length =
$payload_bytes[$payload_index]
                                        $payload_index = $payload_index + 1

                                        # When DNS Compression is in use, the record
will not be terminated with a null
```

David R Fletcher Jr., 6fletch9@gmail.com

```
be found indicating that the next byte
where the request/response continues.

$payload_bytes[$payload_index]

$payload_bytes[$llmnr_ptr_offset]

$payload_bytes[$llmnr_ptr_offset..($llmnr_ptr_offset + $llmnr_field_length - 1)]
+ $llmnr_field_length

$llmnr_field_length $llmnr_field_value_bytes

$payload_bytes[$llmnr_ptr_offset]
+ 1

$payload_bytes[$payload_index..($payload_index + 1)]

$payload_bytes[$payload_index..($payload_index + 1)]

$name) > $null

$null

$llmnr_ancount.ToString()) > $null

Port with wrong destination address") > $null
```

```
                                    # Instead, a byte value of 192 (or C0) will
                                    # represents the offset into the DNS packet
                                    if ($llmnr_field_length -eq 192)
                                    {
                                        $llmnr_ptr_offset =
                                        $payload_index = $payload_index + 1
                                        $llmnr_field_length =
                                        $llmnr_ptr_offset = $mdns_ptr_offset + 1
                                        while ($llmnr_field_length -ne 0)
                                        {
                                            $llmnr_field_value_bytes =
                                            $llmnr_ptr_offset = $llmnr_ptr_offset
                                            $llmnr_field_value = DataToString 0
                                            $name = $name + $llmnr_field_value
                                            $llmnr_field_length =
                                            $llmnr_ptr_offset = $llmnr_ptr_offset
                                            if ($llmnr_field_length -ne 0)
                                            {
                                                $name = ($name + ".")
                                            }
                                        }
                                        break
                                    }
                                    if ($llmnr_field_length -ne 0)
                                    {
                                        $name = ($name + ".")
                                    }
                                }
                                $llmnr_record_type =
                                $payload_index = $payload_index + 2
                                $llmnr_record_class =
                                $payload_index = $payload_index + 2
                                $analyzer.console_queue.Add(" [i] Host: " +
                            }
                        }
                        else
                        {
                            $analyzer.console_queue.Add(" [i] Type: Response") >
                            $analyzer.console_queue.Add(" [i] Count: " +
                            # May Parse LLMNR Responses Further In The Future
                        }
                    }
                    else
                    {
                        $analyzer.console_queue.Add("Packet received on LLMNR UDP
                    }
                }
            }
            default
            {
                # Do Nothing
            }
        }

    }
    # OSPF Processing
```

David R Fletcher Jr., 6fletch9@gmail.com

```
            89
            {
                if ($analyzer.show_ospf)
                {
                    if ($destination_IP.ToString() -eq "224.0.0.5")
                    {
                        $ospf_version = $binary_reader.ReadByte()
                        $ospf_type = $binary_reader.ReadByte()
                        $ospf_length = DataToUInt16 $binary_reader.ReadBytes(2)
                        $ospf_router_bytes = $binary_reader.ReadBytes(4)
                        $ospf_router = [System.Net.IPAddress]$ospf_router_bytes
                        $ospf_area_bytes = $binary_reader.ReadBytes(4)
                        $ospf_area = [System.Net.IPAddress]$ospf_area_bytes
                        $ospf_checksum = DataToUInt16 $binary_reader.ReadBytes(2)
                        $ospf_authType = DataToUInt16 $binary_reader.ReadBytes(2)

                        $analyzer.console_queue.Add("OSPF v" + $ospf_version.ToString() +
" Packet Observed from " + $source_IP.ToString()) > $null

                        switch($ospf_authType)
                        {
                            # Handle OSPF Packets with NULL Auth
                            0
                            {
                                switch($ospf_type)
                                {
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                    }
                                    2
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: DB
Descriptor packet.") > $null
                                    }
                                    3
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS
Request packet.") > $null
                                    }
                                    4
                                    {
                                        $analyzer.console_queue.Add(" [!] Type: LS Update
packet.") > $null
                                    }
                                    5
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS Ack
packet.") > $null
                                    }
                                }

                                $analyzer.console_queue.Add(" [!] Auth: NULL") > $null
                            }
                            # Handle OSPF Packets with Password Auth
                            1
                            {
                                switch($ospf_type)
                                {
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                    }
                                    2
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: DB
Descriptor packet.") > $null
                                    }
                                    3
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS
Request packet.") > $null
                                    }
                                    4
                                    {
                                        $analyzer.console_queue.Add(" [!] Type: LS Update
packet.") > $null
                                    }
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    5
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS Ack
packet.") > $null
                                    }
                                }

                                    $analyzer.console_queue.Add(" [!] Auth: Password") >
$null

                                    $password_bytes = $binary_reader.ReadBytes(8)
                                    $ospf_authData = DataToString 0 8 $password_bytes
                                    $analyzer.console_queue.Add(" [!] Password: " +
$ospf_authData) > $null
                            }
                            # Handle OSPF Packets With Cryptographic Auth
                            2
                            {
                                $null_bytes = $binary_reader.ReadBytes(2)
                                $ospf_key_id = $binary_reader.ReadByte()
                                $ospf_auth_length = $binary_reader.ReadByte()
                                $ospf_auth_sequence_bytes = $binary_reader.ReadBytes(4)
                                $ospf_auth_sequence = DataToUInt32
$ospf_auth_sequence_bytes

                                switch($ospf_type)
                                {
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                        $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                        $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                        $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null
                                        $ospf_netmask_bytes = $binary_reader.ReadBytes(4)
                                        $ospf_netmask =
[System.Net.IPAddress]$ospf_netmask_bytes
                                        $opsf_hello_interval = DataToUInt16
$binary_reader.ReadBytes(2)
                                        $ospf_hello_options = $binary_reader.ReadByte()
                                        $ospf_hello_router_pri =
$binary_reader.ReadByte()
                                        $ospf_dead_interval_bytes =
$binary_reader.ReadBytes(4)
                                        $ospf_dead_interval = DataToUInt32
$ospf_dead_interval_bytes
                                        $ospf_dr_bytes = $binary_reader.ReadBytes(4)
                                        $ospf_dr_ip =
[System.Net.IPAddress]$ospf_dr_bytes
                                        $ospf_br_bytes = $binary_reader.ReadBytes(4)
                                        $ospf_br_ip =
[System.Net.IPAddress]$ospf_br_bytes
                                        $ospf_crypt_hash_bytes =
$binary_reader.ReadBytes(16)
                                        $ospf_crypt_hash = DataToHexString 0 16
$ospf_crypt_hash_bytes
                                        $analyzer.console_queue.Add(" [i] Auth Hash: " +
$ospf_crypt_hash.ToString())
                                        $analyzer.console_queue.Add(" [i] Designated
Router: " + $ospf_dr_ip.ToString())
                                    }
                                    2
                                    {
                                        # May need to expand on DB Descriptor Packets
(Just to get routing table).
                                        $analyzer.console_queue.Add(" [i] Type: DB
Descriptor packet.") > $null
                                        $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                        $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                        $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null

                                    }
                                    3
                                    {
                                        # Link-State Request Packets are Less Interesting
```

```
                                    $analyzer.console_queue.Add(" [i] Type: LS
Request packet.") > $null
                                    $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                    $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                    $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null
                                }
                                4
                                {
                                    # Link-State Update Packets Can Be Used to Build
a Routing Table
                                    $analyzer.console_queue.Add(" [!] Type: LS Update
packet.") > $null
                                    $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                    $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                    $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null
                                }
                                5
                                {
                                    # Link-State Acknowledgement Packets May Need to
be Used to Validate Updates
                                    $analyzer.console_queue.Add(" [i] Type: LS Ack
packet.") > $null
                                    $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                    $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                    $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null
                                }
                            }
                        }
                    }
                }
                elseif ($destination_IP.ToString() -eq "224.0.0.6")
                {
                    $ospf_version = $binary_reader.ReadByte()
                    $ospf_type = $binary_reader.ReadByte()
                    $ospf_length = DataToUInt16 $binary_reader.ReadBytes(2)
                    $ospf_router_bytes = $binary_reader.ReadBytes(4)
                    $ospf_router = [System.Net.IPAddress]$ospf_router_bytes
                    $ospf_area_bytes = $binary_reader.ReadBytes(4)
                    $ospf_area = [System.Net.IPAddress]$ospf_area_bytes
                    $ospf_checksum = DataToUInt16 $binary_reader.ReadBytes(2)
                    $ospf_authType = DataToUInt16 $binary_reader.ReadBytes(2)

                    $analyzer.console_queue.Add("OSPF v" + $ospf_version.ToString() +
" Packet Observed from " + $source_IP.ToString()) > $null

                    switch($ospf_authType)
                    {
                        # Handle OSPF Packets with NULL Auth
                        0
                        {
                            switch($ospf_type)
                            {
                                1
                                {
                                    $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                }
                                2
                                {
                                    $analyzer.console_queue.Add(" [i] Type: DB
Descriptor packet.") > $null
                                }
                                3
                                {
                                    $analyzer.console_queue.Add(" [i] Type: LS
Request packet.") > $null
                                }
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    4
                                    {
                                        $analyzer.console_queue.Add(" [!] Type: LS Update
packet.") > $null
                                    }
                                    5
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS Ack
packet.") > $null
                                    }
                                }
                                $analyzer.console_queue.Add(" [!] Auth: NULL") > $null
                            }
                            # Handle OSPF Packets with Password Auth
                            1
                            {
                                switch($ospf_type)
                                {
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                    }
                                    2
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: DB
Descriptor packet.") > $null
                                    }
                                    3
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS
Request packet.") > $null
                                    }
                                    4
                                    {
                                        $analyzer.console_queue.Add(" [!] Type: LS Update
packet.") > $null
                                    }
                                    5
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: LS Ack
packet.") > $null
                                    }
                                }
                                $analyzer.console_queue.Add(" [!] Auth: Password") > $null
                                $password_bytes = $binary_reader.ReadBytes(8)
                                $ospf_authData = DataToString 0 8 $password_bytes
                                $analyzer.console_queue.Add(" [!] Password: " +
$ospf_authData) > $null
                            }
                            # Handle OSPF Packets With Cryptographic Auth
                            2
                            {
                                $null_bytes = $binary_reader.ReadBytes(2)
                                $ospf_key_id = $binary_reader.ReadByte()
                                $ospf_auth_length = $binary_reader.ReadByte()
                                $ospf_auth_sequence_bytes = $binary_reader.ReadBytes(4)
                                $ospf_auth_sequence = DataToUInt32
$ospf_auth_sequence_bytes

                                switch($ospf_type)
                                {
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [i] Type: Hello
packet.") > $null
                                        $analyzer.console_queue.Add(" [i] Auth:
Cryptographic (MD5)") > $null
                                        $analyzer.console_queue.Add(" [i] KeyID: " +
$ospf_key_id.ToString()) > $null
                                        $analyzer.console_queue.Add(" [i] Auth Seq: " +
$ospf_auth_sequence.ToString()) > $null
                                        $ospf_netmask_bytes = $binary_reader.ReadBytes(4)
                                        $ospf_netmask =
[System.Net.IPAddress]$ospf_netmask_bytes
                                        $opsf_hello_interval = DataToUInt16
$binary_reader.ReadBytes(2)
                                        $ospf_hello_options = $binary_reader.ReadByte()
```

```
$binary_reader.ReadByte()
$binary_reader.ReadBytes(4)
$ospf_dead_interval_bytes

[System.Net.IPAddress]$ospf_dr_bytes

[System.Net.IPAddress]$ospf_br_bytes
$binary_reader.ReadBytes(16)
$ospf_crypt_hash_bytes
$ospf_crypt_hash.ToString())
Router: " + $ospf_dr_ip.ToString())
                                    }
                                    2
                                    {
(Just to get routing table).
Descriptor packet.") > $null
Cryptographic (MD5)") > $null
$ospf_key_id.ToString()) > $null
$ospf_auth_sequence.ToString()) > $null
                                    }
                                    3
                                    {
Request packet.") > $null
Cryptographic (MD5)") > $null
$ospf_key_id.ToString()) > $null
$ospf_auth_sequence.ToString()) > $null
                                    }
                                    4
                                    {
a Routing Table
packet.") > $null
Cryptographic (MD5)") > $null
$ospf_key_id.ToString()) > $null
$ospf_auth_sequence.ToString()) > $null
                                    }
                                    5
                                    {
be Used to Validate Updates
packet.") > $null
Cryptographic (MD5)") > $null
$ospf_key_id.ToString()) > $null
$ospf_auth_sequence.ToString()) > $null
                                    }
                                }
                            }
                        }
                    }
                    else
```

```
$ospf_hello_router_pri =
$ospf_dead_interval_bytes =
$ospf_dead_interval = DataToUInt32
$ospf_dr_bytes = $binary_reader.ReadBytes(4)
$ospf_dr_ip =
$ospf_br_bytes = $binary_reader.ReadBytes(4)
$ospf_br_ip =
$ospf_crypt_hash_bytes =
$ospf_crypt_hash = DataToHexString 0 16
$analyzer.console_queue.Add(" [i] Auth Hash: " +
$analyzer.console_queue.Add(" [i] Designated

    # May need to expand on DB Descriptor Packets
    $analyzer.console_queue.Add(" [i] Type: DB
    $analyzer.console_queue.Add(" [i] Auth:
    $analyzer.console_queue.Add(" [i] KeyID: " +
    $analyzer.console_queue.Add(" [i] Auth Seq: " +

    # Link-State Request Packets are Less Interesting
    $analyzer.console_queue.Add(" [i] Type: LS
    $analyzer.console_queue.Add(" [i] Auth:
    $analyzer.console_queue.Add(" [i] KeyID: " +
    $analyzer.console_queue.Add(" [i] Auth Seq: " +

    # Link-State Update Packets Can Be Used to Build
    $analyzer.console_queue.Add(" [!] Type: LS Update
    $analyzer.console_queue.Add(" [i] Auth:
    $analyzer.console_queue.Add(" [i] KeyID: " +
    $analyzer.console_queue.Add(" [i] Auth Seq: " +

    # Link-State Acknowledgement Packets May Need to
    $analyzer.console_queue.Add(" [i] Type: LS Ack
    $analyzer.console_queue.Add(" [i] Auth:
    $analyzer.console_queue.Add(" [i] KeyID: " +
    $analyzer.console_queue.Add(" [i] Auth Seq: " +
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                    {
                        $analyzer.console_queue.Add("Packet received for OSPF Protocol ID
with wrong destination address") > $null
                    }
                }

            }
            # VRRP Processing
            112
            {
                if ($analyzer.show_vrrp)
                {
                    if ($destination_IP.ToString() -eq "224.0.0.18")
                    {
                        $vrrp_version_type = $binary_reader.ReadByte()
                        $vrrp_version = [Int]"0x$(('{0:X}' -f $vrrp_version_type)[0])"
                        # Only type 1 is defined in the RFC, all others are non-existent
                        $vrrp_type = [Int]"0x$(('{0:X}' -f $vrrp_version_type)[1])"
                        $vrrp_rtr_id = $binary_reader.ReadByte()
                        $vrrp_priority = $binary_reader.ReadByte()
                        $vrrp_addr_count = $binary_reader.ReadByte()

                        $analyzer.console_queue.Add("VRRP v" + $vrrp_version + " Packet
Observed from " + $source_IP.ToString()) > $null
                        $analyzer.console_queue.Add(" [i] Router ID: " +
$vrrp_rtr_id.ToString())
                        $analyzer.console_queue.Add(" [i] Priority: " +
$vrrp_priority.ToString())
                        if ($vrrp_priority -lt 250)
                        {
                            $analyzer.console_queue.Add(" [!] Priority May Be Low.
Potential for Hijacking")
                        }

                        $analyzer.console_queue.Add(" [i] Addresses: " +
$vrrp_addr_count.ToString())

                        # VRRP v2 is IPv4 Only
                        if ($vrrp_version -lt 3)
                        {
                            $vrrp_auth_type = $binary_reader.ReadByte()
                            $vrrp_advert_interval = $binary_reader.ReadByte()
                            $vrrp_checksum = DataToUInt16 $binary_reader.ReadBytes(2)

                            # Might be wise to validate this against packet length to
handle malformed packets
                            for ($i = 1; $i -le $vrrp_addr_count; $i++)
                            {
                                try
                                {
                                    $vrrp_address_bytes = $binary_reader.ReadBytes(4)
                                    $vrrp_address =
[System.Net.IPAddress]$vrrp_address_bytes

                                    $analyzer.console_queue.Add(" [i] Address " +
$i.ToString() + ": " + $vrrp_address.ToString()) > $null
                                }
                                catch
                                {
                                    $analyzer.console_queue.Add(" [w] Malformed
Packet!!")
                                }
                            }

                            try
                            {
                                switch ($vrrp_auth_type)
                                {
                                    0
                                    {
                                        $analyzer.console_queue.Add(" [!] Auth: None") >
$null
                                    }
                                    1
                                    {
                                        $analyzer.console_queue.Add(" [!] Auth: Simple
Text Password") > $null
                                        $vrrp_auth_data_bytes =
$binary_reader.ReadBytes(8)
                                        $vrrp_auth_data = DataToString 0 8
$vrrp_auth_data_bytes
```

David R Fletcher Jr., 6fletch9@gmail.com

```
                                    $analyzer.console_queue.Add(" [!] Password: " +
$vrrp_auth_data) > $null
                                }
                                2
                                {
                                    $analyzer.console_queue.Add(" [i] Auth: IP Auth
Header") > $null
                                }
                            }
                        }
                        catch
                        {
                        }
                    }
                    elseif ($IP_version -eq 4)
                    {
                        $vrrp_rsv_advert_interval_bytes = $binary_reader.ReadBytes(4)
                        $vrrp_rsv_advert_interval = DataToUInt32
$vrrp_rsv_advert_interval_bytes
                        $vrrp_checksum = DataToUInt16 $binary_reader.ReadBytes(2)

                        # Might be wise to validate this against packet length to
handle malformed packets
                        for ($i = 1; $i -le $vrrp_addr_count; $i++)
                        {
                            try
                            {
                                $vrrp_address_bytes = $binary_reader.ReadBytes(4)
                                $vrrp_address =
[System.Net.IPAddress]$vrrp_address_bytes

                                $analyzer.console_queue.Add(" [i] Address " +
$i.ToString() + ": " + $vrrp_address.ToString()) > $null
                            }
                            catch
                            {
                            $analyzer.console_queue.Add(" [w] Malformed Packet!!")
                            }
                        }
                    }
                    elseif ($IP_version -eq 6)
                    {
                        $vrrp_rsv_advert_interval_bytes = $binary_reader.ReadBytes(4)
                        $vrrp_rsv_advert_interval = DataToUInt32
$vrrp_rsv_advert_interval_bytes
                        $vrrp_checksum = DataToUInt16 $binary_reader.ReadBytes(2)

                        # Might be wise to validate this against packet length to
handle malformed packets
                        for ($i = 1; $i -le $vrrp_addr_count; $i++)
                        {
                            try
                            {
                                $vrrp_address_bytes = $binary_reader.ReadBytes(16)
                                $vrrp_address =
[System.Net.IPAddress]$vrrp_address_bytes

                                $analyzer.console_queue.Add(" [i] Address " +
$i.ToString() + ": " + $vrrp_address.ToString()) > $null
                            }
                            catch
                            {
                                $analyzer.console_queue.Add(" [w] Malformed
Packet!!")
                            }
                        }
                    }
                }
                else
                {
                    $analyzer.console_queue.Add("Packet received on VRRP Protocol ID
with wrong destination address") > $null
                }
            }
        }
    }

    $binary_reader.Close()
    $memory_stream.Dispose()
    $memory_stream.Close()
```

David R Fletcher Jr., 6fletch9@gmail.com

```
}

# Moved sniffer to main script instead of function so thread can be properly shut down
$analyzer.console_queue.Add("Starting sniffer...") > $null
$sniffer_runspace = [RunspaceFactory]::CreateRunspace()
$sniffer_runspace.Open()
$sniffer_runspace.SessionStateProxy.SetVariable('analyzer',$analyzer)
$sniffer_powershell = [PowerShell]::Create()
$sniffer_powershell.Runspace = $sniffer_runspace
$sniffer_powershell.AddScript($shared_basic_functions_scriptblock) > $null
$sniffer_powershell.AddScript($sniffer_scriptblock).AddArgument($IP).AddArgument($RunTime
) > $null
$sniffer_powershell.BeginInvoke() > $null

    while ($analyzer.running -or ($analyzer.console_queue.Count -gt 0))
    {

        while($analyzer.console_queue.Count -gt 0)
        {
                switch -wildcard ($analyzer.console_queue[0])
                {
                    "*[!]*"
                    {
                        Write-Host $analyzer.console_queue[0] -ForegroundColor
"DarkYellow"
                        $analyzer.console_queue.RemoveAt(0)
                    }
                    "Windows Firewall = Enabled"
                    {
                        Write-Warning($analyzer.console_queue[0])
                        $analyzer.console_queue.RemoveAt(0)
                    }

                    default
                    {
                        Write-Output $analyzer.console_queue[0]
                        $analyzer.console_queue.RemoveAt(0)
                    }

                }
        }

        if([Console]::KeyAvailable)
        {
            $key = [System.Console]::ReadKey()

            switch ($key.KeyChar)
            {
                'h'
                {
                    $analyzer.show_hsrp = !$analyzer.show_hsrp
                    if ($analyzer.show_hsrp)
                    {
                        $analyzer.console_queue.Add("HSRP Toggle: ON") > $null
                    }
                    else
                    {
                        $analyzer.console_queue.Add("HSRP Toggle: OFF") > $null
                    }
                }
                'd'
                {
                    $analyzer.show_dhcp = !$analyzer.show_dhcp
                    if ($analyzer.show_dhcp)
                    {
                        $analyzer.console_queue.Add("DHCP Toggle: ON") > $null
                    }
                    else
                    {
                        $analyzer.console_queue.Add("DHCP Toggle: OFF") > $null
                    }
                }
                'o'
                {
                    $analyzer.show_ospf = !$analyzer.show_ospf
                    if ($analyzer.show_ospf)
                    {
                        $analyzer.console_queue.Add("OSPF Toggle: ON") > $null
                    }
                    else
                    {
```

```
            $analyzer.console_queue.Add("OSPF Toggle: OFF") > $null
        }

    }
    'v'
    {
        $analyzer.show_vrrp = !$analyzer.show_vrrp
        if ($analyzer.show_vrrp)
        {
            $analyzer.console_queue.Add("VRRP Toggle: ON") > $null
        }
        else
        {
            $analyzer.console_queue.Add("VRRP Toggle: OFF") > $null
        }
    }
    'l'
    {
        $analyzer.show_llmnr = !$analyzer.show_llmnr
        if ($analyzer.show_llmnr)
        {
            $analyzer.console_queue.Add("LLMNR Toggle: ON") > $null
        }
        else
        {
            $analyzer.console_queue.Add("LLMNR Toggle: OFF") > $null
        }

    }
    'm'
    {
        $analyzer.show_mdns = !$analyzer.show_mdns
        if ($analyzer.show_mdns)
        {
            $analyzer.console_queue.Add("mDNS Toggle: ON") > $null
        }
        else
        {
            $analyzer.console_queue.Add("mDNS Toggle: OFF") > $null
        }

    }
    'n'
    {
        $analyzer.show_nbns = !$analyzer.show_nbns
        if ($analyzer.show_nbns)
        {
            $analyzer.console_queue.Add("NBNS Toggle: ON") > $null
        }
        else
        {
            $analyzer.console_queue.Add("NBNS Toggle: OFF") > $null
        }
    }
    'q'
    {
        Write-Host ("Shuting Down Analyzer...Please Wait") > $null
        # Set analyzer to stopped and reset show variables
        $analyzer.running = $false
        $analyzer.show_dhcp = $true
        $analyzer.show_hsrp = $true
        $analyzer.show_llmnr = $true
        $analyzer.show_mdns = $true
        $analyzer.show_nbns = $true
        $analyzer.show_ospf = $true
        $analyzer.show_vrrp = $true

        # Kill the sniffer objects
        $sniffer_powershell.Dispose()
        $sniffer_runspace.CloseAsync()
        $sniffer_runspace.Dispose()
        Write-Host ("Shutdown Complete") > $null
        return
    }
    default
    {
        $analyzer.console_queue.Add("Runtime Interactive Help:") > $null
        $analyzer.console_queue.Add("D = DHCP Toggle") > $null
        $analyzer.console_queue.Add("H = HSRP Toggle") > $null
        $analyzer.console_queue.Add("L = LLMNR Toggle") > $null
        $analyzer.console_queue.Add("M = mDNS Toggle") > $null
```

```
                    $analyzer.console_queue.Add("O = OSPF Toggle") > $null
                    $analyzer.console_queue.Add("V = VRRP Toggle") > $null
                    $analyzer.console_queue.Add("Q = Shut Down Analyzer") > $null
                }
            }
        }

        Start-Sleep -m 5
    }
}
```