Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Network Monitoring and Threat Detection In-Depth (Security 503)"
at http://www.giac.org/registration/gcia

Recognizing Suspicious Network Connections with Python

GIAC (GCIA) Gold Certification

Author: Gregory Melton, Gregory.Melton@student.sans.edu
Advisor: Clay Risenhoover

Accepted: May 10, 2020

Abstract

Endpoint protection solutions tend to focus on system indicators and known malicious code to defend both enterprise and Small Office-Home Office (SOHO) users. In the absence of a Security Operations Center (SOC) or paid antivirus services, there are few proactive defense options for hobbyists and SOHO owners. A significant problem is how advanced persistent threat (APT) actors' Tactics, Techniques, and Procedures (TTPs) have changed over the years; it is common for advanced actors to exploit poorly defended subcontractors and seemingly less relevant targets. This brings the Small Office-Home Office into the picture as a pivotal defense point against advanced attackers. This research intends to focus on attackers using Shell, terminal, or Remote Access Tool (RAT) connections to SOHO endpoints. This research seeks to block interactive connections with system-level network logging and blacklist automation. This method will recognize malicious connections and automatically block them in near real-time.

© 2020 The SANS Institute, Author Retains Full Rights

© 2020 The SANS Institute Author retains full rights.

1. Introduction

Meaningful endpoint security against interactive threats is a difficult security challenge to manage. Typically, if an attacker obtains a Shell or terminal connection, they have already defeated the system's security protocols. From a developer's standpoint, it may not be worth the effort to wrangle Windows Application Programming Interfaces (APIs) against the unlikely occurrence of an interactive threat to home or small office users. Unfortunately, APT groups commonly target smaller entities and search for backend routes into larger targets by abusing contractor access. By focusing a defensive posture on network connections as a chokepoint, it may be possible to create a module that recognizes suspicious IP connections and adds said IP address to the system's firewall blacklist. Near real-time connection assessments and automated remediation is the chosen method to increase endpoint security for SOHO users. This paper will provide hobbyists and security fledglings an opportunity to understand a major chokepoint in endpoint security for interactive threats.

To provide measurable evidence of a proposed solution, proper logging of vital network activity will be implemented. By logging all activity before any preventative measures are taken, the reality of system communications prior to defensive tampering will be baselined. Comparing this baseline to system traffic after defense implementations will provide metrics as to the level of success. Additionally, defensive modules will record actions, output notifications, and alerts, which can be aggregated to see how effective they are. Finally, tests will be conducted against the two primary endpoint market share giants, OSX and Windows 10, to ensure an inclusive solution. The solution should be simple, affordable, and maintainable with readily available resources and open source products.

1.1 Endpoints and SOHOs

An endpoint can be defined as any computing device that is the final destination for data on a network. The most common user examples include tablets, smartphones,

© 2020 The SANS Institute Author retains full rights.

desktops, and laptops (Lord 2018). Any internet-connected device, including servers, are endpoints; however, this research will focus on user workstations in a Small Office/Home Office (or Single Office/Home Office, or SOHO). A SOHO typically refers to the category of business that employs anywhere from 1 to 10 workers. In New Zealand, for example, the Ministry of Business, Innovation, and Employment (MBIE) defines a small office as 6 to 19 employees and a micro office as 1 to 5 employees.

For this research, a SOHO is defined as an organization with less than ten workers or a home network (2018 US Legal Inc). Both a small office and home network would include a Gateway Router, Primary workstation (desktop or laptop), and mobile devices. The test environment for this research will represent a 'typical' home office network. The defensive measures proposed are at their core a CIRT methodology and forensics check to find indications of compromise. With a slight adaptation, the same principles should work for any workstation, but a SOHO environment is specified because the malicious indicator will be more evident and actionable.

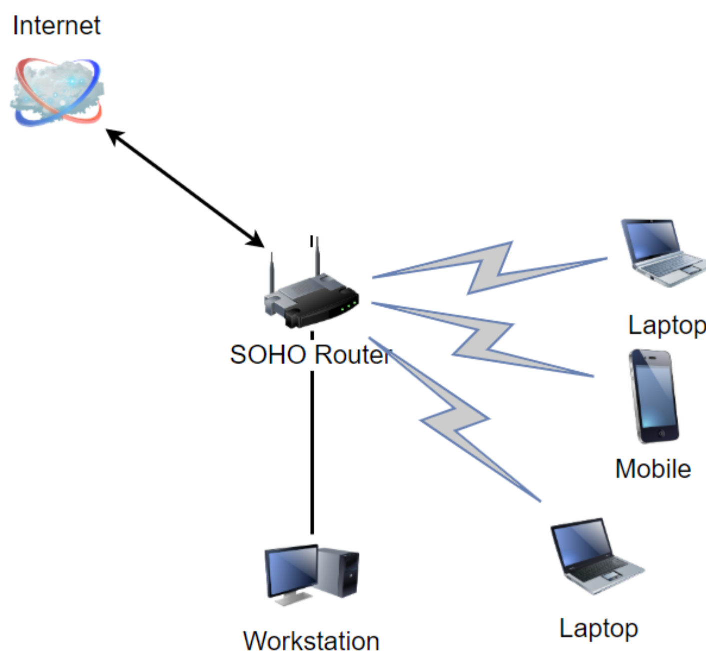

Figure 1. SOHO Example

© 2020 The SANS Institute Author retains full rights.

1.2 Threats to Endpoints

Each year, an increasing number of people work from home, which exposes a vast number of workstations with potentially sensitive data to the internet. One 2017 report found at least 2.8% of people in the U.S. workforce work from home at least half the time (Shepherd 2019). A more staggering statistic claims that as much as 70% of working professionals work remotely at least one day a week (Browne, 2018). The end-users in both environments are often seen as low hanging fruit and low barriers to entry for both cybersecurity professionals, cybercriminals, and APTs alike. Simply put, the largest threat of organizational breach often occurs at the endpoint level (Murray 2019). End-user systems are notoriously difficult to defend due to human interaction, which can often negate technical controls put in place by security measures. Organizations try to mitigate this risk through compliance requirements, mandatory training, and periodic phishing email simulations to gauge effectiveness. Both technical controls and user training are common defenses against such attacks, but since a Small/Home Office lacks this training, it remains a weak point. With an increasing number of employees working from home every year, these security concerns are more relevant than ever.

On a positive note, a typical home environment has a relatively small attack surface. Most home networks are not running internet-connected servers, which cuts down the number of devices and services that can be exploited from the outside. The downside may be a less-aware user population that may give little thought to system security, click malicious links, or download malware in email attachments.

The goal for this research will not be to stop initial compromise but to recognize when exploitation occurs and automatically shut down the external TCP connection. To help with such defenses, one must first understand how two primary network protocols function: DNS and TCP.

2. DNS

© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

To understand modern network activity, a person must have a basic understanding of the Domain Name System (DNS). DNS is one of the most important protocols used to run the modern internet and is functionally used for most activities the average user performs when online. When a user wants to load a webpage, a translation must occur between what a user types into their web browser and the machine-friendly address necessary to locate a webpage (How DNS works n.d.). In short, DNS is the method by which systems locate resources on the internet. The details of a DNS request are beyond the scope of this paper, but a basic example is shown in Figure 2.

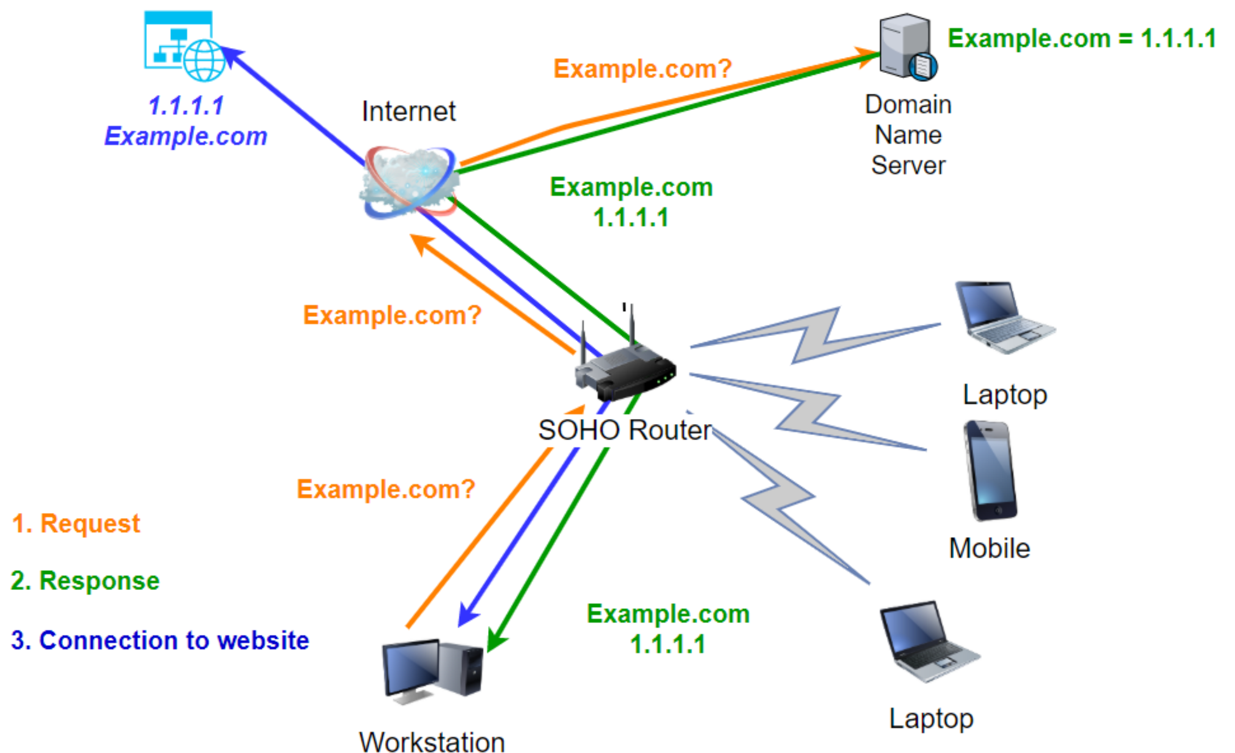

Figure 2. Basic Domain Name Resolution

The request and response process in Figure 2 typically occurs over UDP port 53 as the standard. A DNS cache is a log kept by a system or application of the domain-to-IP mapping resolved by the process shown in Figure 2. This locally cached correlation helps decrease redundant network traffic and increases the speed for which internet resources are located by removing the comparably lengthy external resolution process. Different

© 2020 The SANS Institute Author retains full rights.

operating systems handle DNS caches in different ways. The Windows operating system, for example, keeps a cache that a user can view at the command prompt with the command `ipconfig /displaydns`.

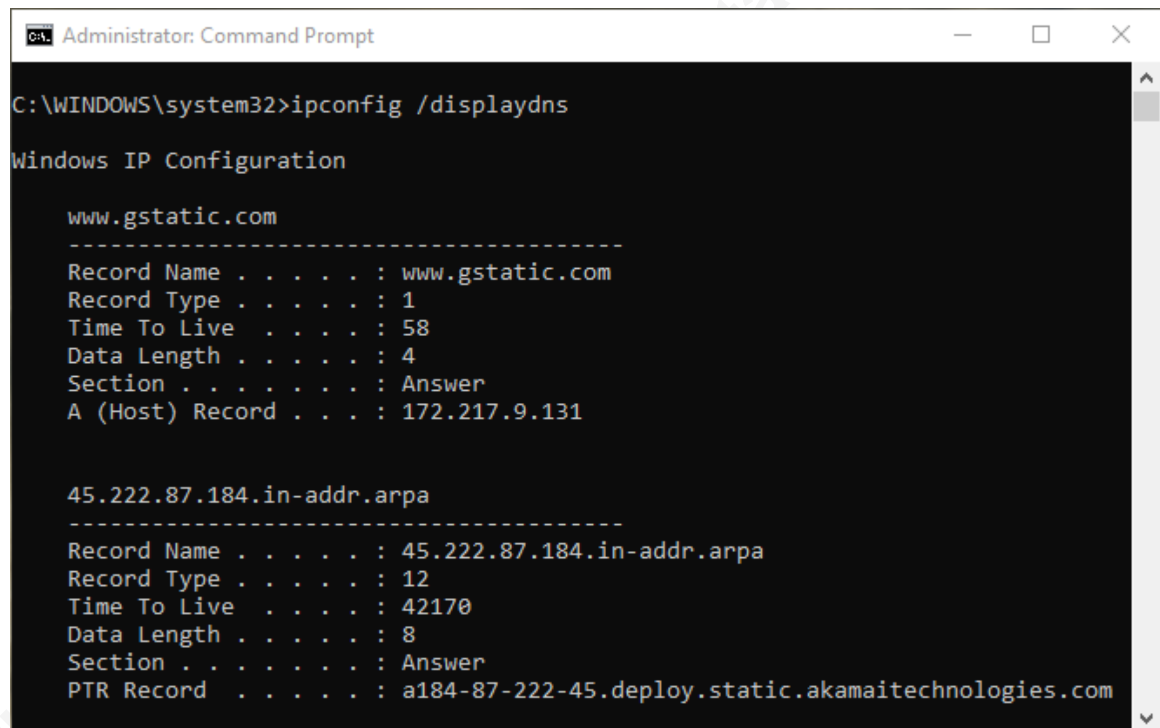

Figure 3. DisplayDNS

The Windows cache is in addition to the web browser/application DNS cache. Linux operating systems, on the other hand, do not cache DNS entries and outsource that job to specific applications. MacOS heavily depends on the Operating System's version.

Since DNS is so pervasive on the internet, it would be relatively uncommon for an internet connection to be established without having a DNS entry somewhere in the host operating systems logs, caches, or processes. An internet connection that does not make a DNS request first is a fundamental indicator for recognizing suspicious connections. While this principle will be the crux of how automation identifies malicious connections, it is important to note that not all non-DNS connections are malicious. Legitimate software may also be hard-coded with IP addresses or may obtain their

© 2020 The SANS Institute Author retains full rights.

resource destinations through less standard means. With that in mind, it is still worth checking up on any connection made that does not have a corresponding DNS entry.

3. TCP Network Connections

Once an internet resource is located by the DNS process as previously described, the end-user's system will request the now known IP address for that resource. Every major end-user operating system frequently establishes connections to resources using the Transmission Control Protocol (RFC-793). TCP is intended to provide a reliable process-to-process communication service in a multinetwork environment (1981, IETF). In short, TCP is the primary way in which most modern computer systems transfer data. Due to the prevalence of this protocol, and its rampant use in end-user systems, the scope of this research will focus on TCP connections as the primary chokepoint for end-user defense against interactive threats.

Modern operating systems are in constant communication with various resources using TCP. Background processes will download system updates, weather applications will get the latest local forecast, and various third-party applications will constantly communicate with an abundance of resources. While downloads and updates may be automated and nearly invisible to the human user, the operating system must be aware of each connection. All major operating systems can, therefore, be queried, at will (assuming the right level of user permissions), to request a list of active connections as shown below with the 'netstat' command:

WINDOWS: `netstat -na | findstr -i "established"`

MACOS: `netstat -n | grep -i established`

LINUX: `netstat -na | grep -i established`

© 2020 The SANS Institute Author retains full rights.

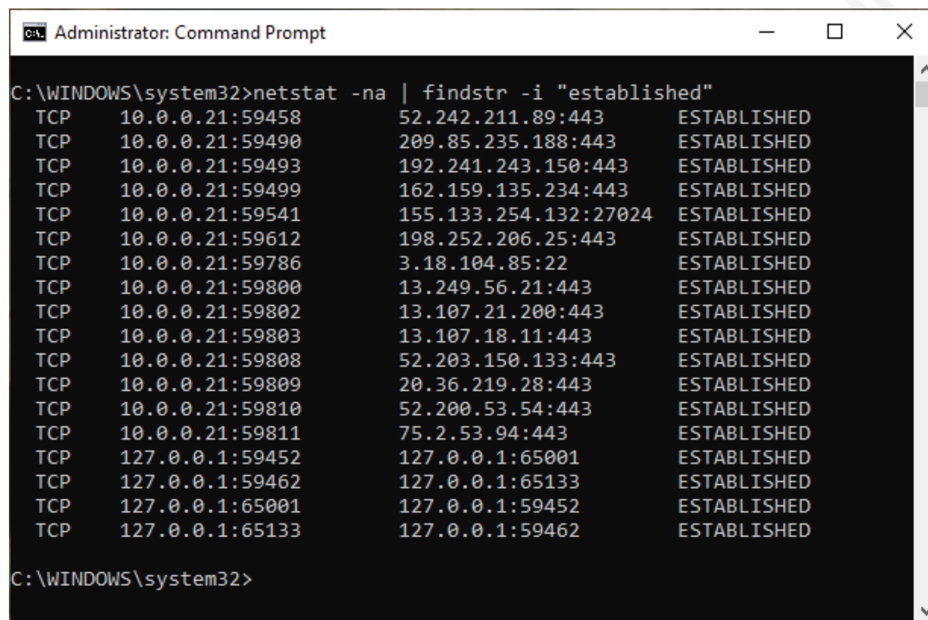

Figure 4. Windows netstat

With a list of established connections, an analyst, user, or program can take follow-on actions to verify legitimate connections or notice suspicious connections. An attacker must exfiltrate data to be considered interactive. Even an automated process to steal files must connect to an external resource controlled by the malicious actor at some point. Having both lists for DNS entries and established connections would enable a user to identify anything suspicious by comparing lists and looking for anomalies. A suspicious entry, in this case, would be an established connection that does not have a corresponding DNS entry.

4. Blocking Direct IP Calls as a Theory

Once a suspicious connection is determined to be malicious, a user could then block access to that external IP address. This concept hinges on the malware making a connection without creating a DNS entry on exfiltration. Some forms of malware, such as Emotet (a trojan virus), are hard-coded with a direct connection to the C2 node's IP address. A hardcoded IP means the attacker predetermined an internet resource and added that IP address to the compiled code delivered to the user. Since it is hard-coded, the IP will not change and does not use DNS to locate the resource. A hard-coded IP address

© 2020 The SANS Institute Author retains full rights.

would effectively bypass DNS servers since no name resolution is requested and, therefore, would not be present in the DNS logs. It is not uncommon to have DNS protections in place that would automatically deny a connection to known malicious resources, but hard-coding defeats this protection. Therefore, the logical solution might be to disallow direct IP calls to external networks.

There are very few reasons an endpoint for a non-technical person would make connections to external IP addresses without first requesting the resource from a DNS server. The difficulty lies in how to check connections against the DNS cache effectively. A higher-level scripting language could manage the tasks necessary to vet connections, but network latency would be a problem. This research will work toward prevention by tracking network connections and DNS requests/responses at the host/endpoint level. Enabling such visibility is the first step toward automating follow-on actions like blocking malicious connections through the local firewall.

5. Scapy and Python

Most operating systems have third party software available to do the heavy lifting for greater system visibility. Windows System Internals System Monitor (sysmon) for example, can be downloaded and configured to keep track of both network traffic and DNS traffic. Tools like sysmon are certainly viable but require some setup and configuration, which can be cumbersome to automate. If a user runs a local DNS server, they have a log of requests on the server. Unfortunately, it is difficult to then compare those specific DNS logs to network connections by endpoints at the network level. To maintain scope for this research, it will not be possible to proactively block direct IP connections for an entire network at the server level. Instead, the focus will be on a lightweight script/program that will use local resources to recognize and potentially break suspicious connections.

To generate the necessary data at an endpoint, a system can use a Python module called “Scapy”. Scapy is a packet manipulation tool and python library which can perform an abundance of packet production, manipulation and sniffing. The packet

© 2020 The SANS Institute Author retains full rights.

sniffing functionality of Scapy's robust capabilities will allow an endpoint to watch and record both DNS requests/responses and monitor for established network connections. Fortunately, Python is a cross-platform language, which means that Scapy's sniffing functionality is as well. Such flexibility makes Python and Scapy a good choice to produce a functional Proof of Concept (POC) with which to work.. The following script will set up a simple packet sniffer for DNS and TCP related traffic and act as a starting point for additional packet manipulation.

```
From scapy.all import *

packets = sniff(filter="tcp[tcpflags] & (tcp-syn) !=0 or port
53", session=IPSession,  count=2, prn=lambda x: x.summary())
```

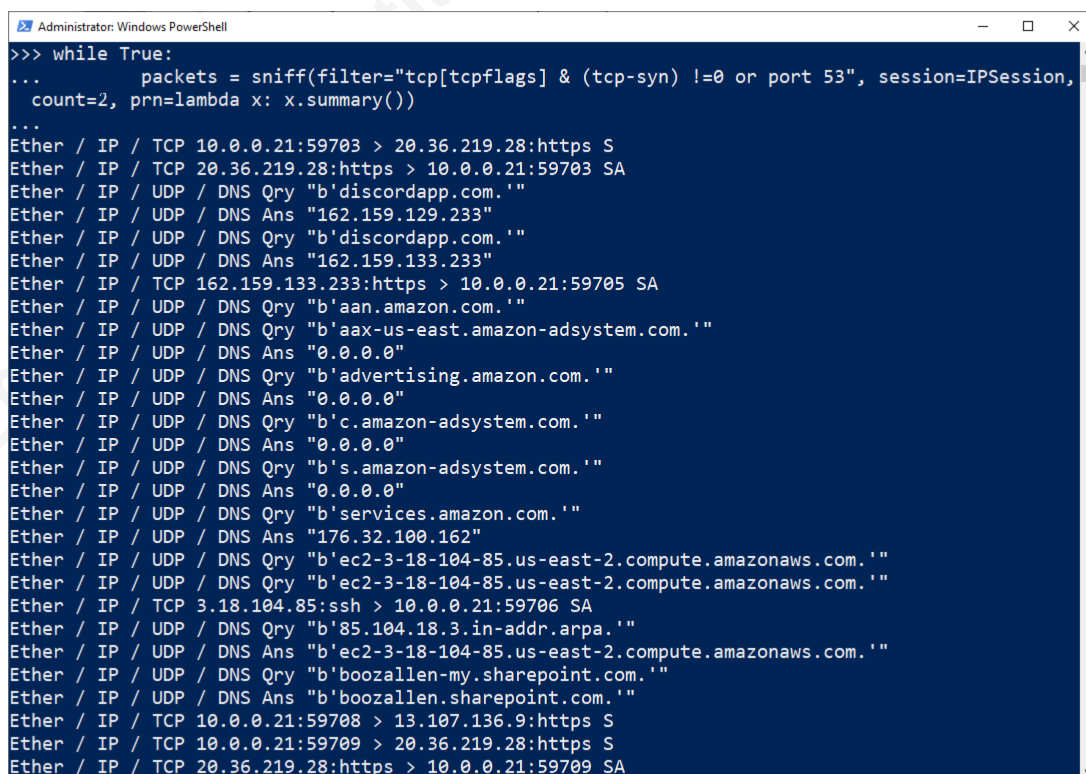

Figure 5. Scapy Sniffing with PoC Module

To replicate the code in Figure 5, a Windows system only needs to install WinPcap, Python/Scapy and have administrator privileges. MacOS and Linux only require Python, Scapy, and sudo permissions.

© 2020 The SANS Institute Author retains full rights.

© 2020 SANS Institute, Author Retains Full Rights

6. Simulation

For this research, an endpoint is assumed to be compromised. The scope of this research does not include intrusion techniques, as such a process is irrelevant for the proposed solution. At this point, hardening a network means stopping data exfiltration or preventing a Command and Control (C2) node from communicating with an endpoint. To simulate this scenario, the affected system will attempt a connection over SSH to an external resource.

Proof of Concept Path:

1) Sniff on the primary interface
2) Filter for DNS requests
    a. Add IP addresses from DNS response to a list
3) Filter for Syn/Syn-Ack TCP connection initiations
    a. Command Scapy to look for Syn-SynAck TCP connections
4) For each connection; if the external IP is not in the DNS list:
    a. Create a list of suspicious IP addresses
    b. Run system commands to add the external IP to the firewall block list
5) Flush firewall rules at regular intervals to reduce system impact.

Case 1: Control - Connections that will not be discovered:

First, it is vital to understand the primary limitation of this POC. A malicious connection where an attacker uses a registered domain will not be noticed by the proposed solution. To illustrate this point, the domain ec2-3-18-104-85.us-east-2.compute.amazonaws.com represents a malicious C2 node that has gained access to the test system.

Once configured, the SSH connection will succeed and remain open because it doesn’t meet the threshold for recognizing a suspicious connection (an established TCP connection without a DNS entry). When SSH connects to a domain, it will resolve the domain for the user. This research focuses entirely on threats that specify their IP address in the payload or through a command injection vulnerability where an IP address is a

© 2020 The SANS Institute Author retains full rights.

malicious endpoint. In short, the SSH connection will proceed as normal. The DNS query, response, and subsequent TCP connection is shown in Figure 6.

```
Administrator: Command Prompt
Windows

Windows IP Configuration

Successfully flushed the DNS Resolver Cache.
Ether / IP / UDP / DNS Qry "b'ec2-3-18-104-85.us-east-2.compute.amazonaws.com.'"
Ether / IP / UDP / DNS Ans "3.18.104.85"
Ether / IP / TCP 3.18.104.85:ssh > 10.0.0.21:60632 SA
Ether / IP / UDP / DNS Qry "b'ctldl.windowsupdate.com.'"
Ether / IP / TCP 8.249.183.254:http > 10.0.0.21:60633 SA

program exiting gracefully

C:\Users\fsugr\Desktop\ISE5901\python>
```

Figure 6. DNS process and TCP connection.

As expected, Figure 7 shows proof that the connection proceeded as expected due to DNS resolution on the host.

```
ec2-user@kali: ~
Using username "ec2-user".
Authenticating with public key "imported-openssh-key"
Linux kali 4.19.0-kali4-amd64 #1 SMP Debian 4.19.28-2kali1 (2019-03-18)

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Apr 13 19:14:24 2020 from 99.151.200.27
ec2-user@kali:~$ curl ifconfig.io
3.18.104.85
ec2-user@kali:~$
```

Figure 7. Stable connection to the kali C2 node over SSH successful

Case 2 – Automatic Suspicious IP detection and response:

© 2020 The SANS Institute Author retains full rights.

The next connection will avoid name resolution by connecting directly to the kali C2 node's IP address. This method simulates a piece of malware with a hardcoded IP address and will, therefore, not have a DNS entry.

```
Administrator: Command Prompt
Windows

Windows IP Configuration

Successfully flushed the DNS Resolver Cache.
Ether / IP / TCP 10.0.0.21:60677 > 3.18.104.85:ssh S
Ether / IP / TCP 3.18.104.85:ssh > 10.0.0.21:60677 SA
Ether / IP / UDP / DNS Qry "b'tile-service.weather.microsoft.com.'"
Ether / IP / UDP / DNS Ans "b'wildcard.weather.microsoft.com.edgekey.net.'"
Ether / IP / UDP / DNS Qry "b'pollserver.lastpass.com.'"
Ether / IP / UDP / DNS Ans "b'lastpass.com.edgekey.net.'"
Ether / IP / UDP / DNS Qry "b'play.google.com.'"
Ether / IP / UDP / DNS Ans "172.217.2.238"

program exiting gracefully

C:\Users\fsugr\Desktop\ISE5901\python>
```

Figure 8. Direct IP connection

Once a connection is attempted (Figure 8) with an IP address instead of a domain name, the DNS protocol is completely bypassed, and the connection proceeds. Notice the lack of DNS query and response prior to the highlighted TCP connection in Figure 8. Methods such as DNS blacklists are incapable of preventing or responding to this type of connection.

At this point, a defensive strategy would depend on proper logging paired with Incident Response (IR) methodologies to catch the direct connection. Since the SSH connection in the example used a specific IP address instead of a domain name, the correct condition triggers, and the IP address can be added to the Firewall block list (Adams 2018). The manual commands to perform firewall updates for a single IP address are as follows:

Windows:

```
netsh advfirewall firewall add rule name="IP Block" dir=in
interface=any action=block remoteip=x.x.x.x
```


© 2020 The SANS Institute Author retains full rights.

MacOS:

```
sudo echo "block drop from any to x.x.x.x" >> /ect/pf.conf" &&
sudo pfctl -e -f /etc/pf.conf
```

7. Proof of Concept Walkthrough and Demonstration

Armed with the knowledge that most TCP connections are preceded by DNS requests, an analyst can take steps to automate the process of focusing on suspicious IP addresses. Using the code from Figure 5 as the base sniffing filter, a Python script can be created to monitor the necessary system communications and isolate rogue IP connections. The code from Figure 8 is only producing a summary for output, which is convenient, but incomplete data. It is important to expand further upon the DNS responses and parse through entries with multiple return values. DNS entries, as seen by the Scapy sniff function, return data as seen in Figure 9.

© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

```
###[ DNS ]###
          id        = 10226
          qr        = 1
          opcode    = QUERY
          aa        = 0
          tc        = 0
          rd        = 1
          ra        = 1
          z         = 0
          ad        = 0
          cd        = 0
          rcode     = ok
          qdcount   = 1
          ancount   = 2
          nscount   = 0
          arcount   = 0
          \qd        \
           |###[ DNS Question Record ]###
           |  qname     = 'us-west-2.prod.twitchcmelolapiservice.s.twitch.a2z.com.'
           |  qtype     = A
           |  qclass    = IN
          \an        \
           |###[ DNS Resource Record ]###
           |  rrname    = 'us-west-2.prod.twitchcmelolapiservice.s.twitch.a2z.com.'
           |  type      = A
           |  rclass    = IN
           |  ttl       = 10
           |  rdlen     = None
           |  rdata     = 54.189.137.128
           |###[ DNS Resource Record ]###
           |  rrname    = 'us-west-2.prod.twitchcmelolapiservice.s.twitch.a2z.com.'
           |  type      = A
           |  rclass    = IN
           |  ttl       = 10
           |  rdlen     = None
           |  rdata     = 34.212.81.223
```

Figure 9. Scapy DNS Response Example with Multiple Entries

Fortunately, Scapy returns data that can be referenced by name. The ‘ancount’ stands for ‘answer count’ and can be incorporated into the Python code for an accurate expectation of results. Furthermore, the ‘rdata’ value for a DNS A record will correspond to the IP address needed for further processing.

© 2020 The SANS Institute Author retains full rights.

```
43 # DNS parsing for to record multiple DNS entries and extract/normalize PTR requests
44 ## Without this loop only the first DNS entry will be returned and the PTR records will be missed.
45                 if packet.haslayer(UDP): # Triggers if a UDP packet
46                     UDPipS.append(packet[IP].dst) #Adds the packet to ta temp list for PCAP incl
47                     if packet.haslayer(DNSRR): # If there is a DNS Response
48                         a_count = packet[DNS].ancount #Find how many answers returned
49                         i = a_count + 4
50                         arp = "arpa"
51                         while i > 4:
52                             if str(packet[0][i].rdata)[0].isnumeric():
53                                 #print(packet[0][i].rdata)
54                                 UDPipS.append(packet[0][i].rdata)
55                             #Useing 'count' to see if the telltale PTR lookup string is in the rrname field
56                             elif packet[0][i].rrname.decode().count("in-addr.arpa")>0:
57                                 #print(packet[0][i].rrname.decode())
58                                 base =(packet[0][i].rrname.decode())
59                                 chop = base[:-14]
60                                 work = chop.split('.')
61                                 final = work[3]+"."+work[2]+"."+work[1]+"."+work[0]
62                                 UDPipS.append(final)
63                             i -= 1
64                 inTnotU = list(set(TCPipS)-set(UDPipS))
```

Figure 10. Python Code for DNS Entries and PTR Records

Using the code in Figure 10, the base lists are primed for comparison. Any data that is present in TCP connections but not the DNS responses can now be added to a list of suspicious IPs.

With a collected list of suspicious IP addresses, it is simple to script out system commands to include in the firewall block list. An example for Windows would look something like Figure 11, where each IP in the file 'suspicious.txt' is added to the Window's Firewall as a block rule.

```
15 #if suspicious.txt exists, start a loop
16 while path.exists("suspicious.txt"):
17 #   signal.signal(signal.SIGINT, signal_handler)
18     list = []
19     inTnotU = []
20     f = open('suspicious.txt', 'r+')
21     for i in f:
22         inTnotU.append(i)
23     f.close()
24     for i in inTnotU:
25         i=(i[:-1])
26         if i not in list:
27             list.append(i)
28     for i in list:
29         os.system('netsh advfirewall firewall add rule name="{}" dir=out interface=any action=block remoteip={}'.
format(i,i))
30     time.sleep(15)
31     for i in list:
32         os.system('netsh advfirewall firewall delete rule name="{}"'.format(i))
33         time.sleep(2)
```

Figure 11. Firewall Loop for Windows

© 2020 The SANS Institute Author retains full rights.

As expected, each suspicious IP address has been automatically added to the Windows firewall block list. The connection to the external server was severed and any additional attempts to connect to the external C2 node will fail.

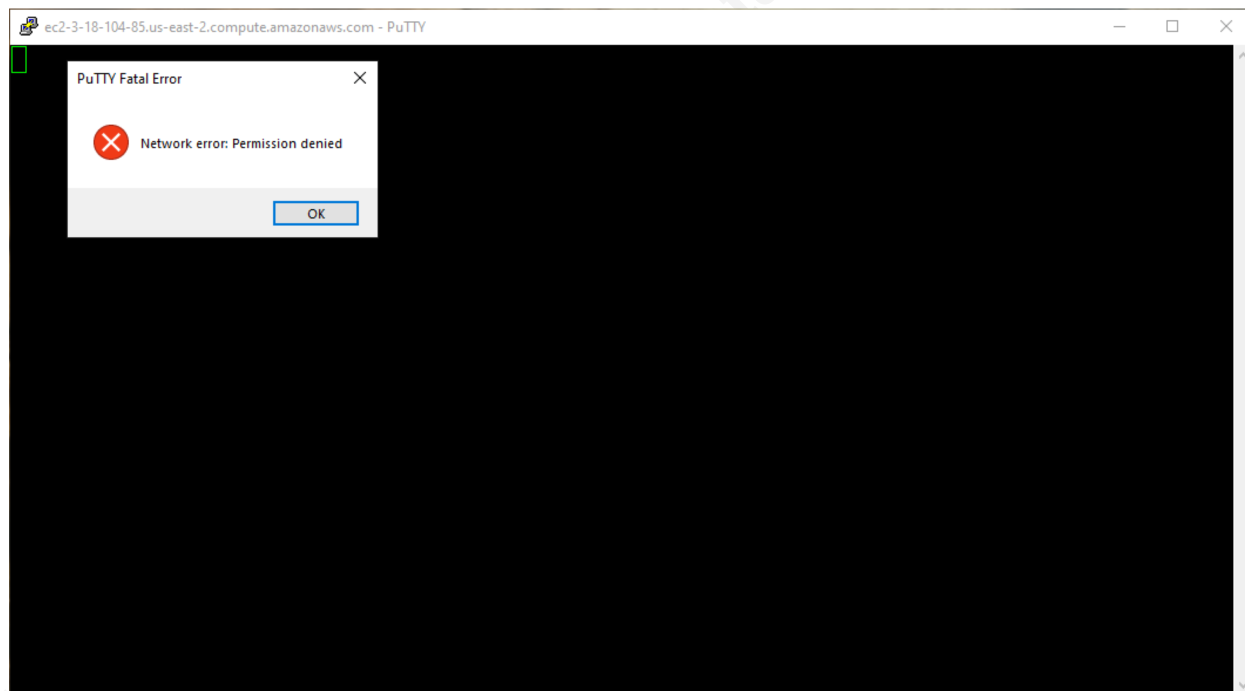

Figure 12 PoC success of blocked malicious connection.

When examining suspicious.txt or the implemented block rules through automation, there will often be false positives. Not every connection without a DNS entry will be malicious, and the proposed PoC did not have a method by which to validate each network node for malicious intent.

Melton, Gregory [USA]

© 2020 The SANS Institute Author retains full rights.

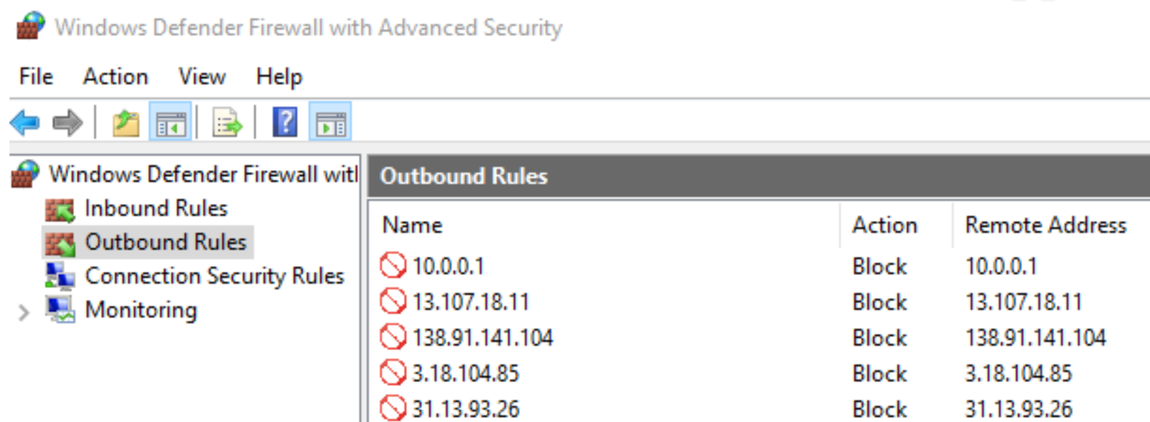

Figure 13. Windows block list after running the PoC

Figure 13 shows the PoC successfully blocked IPs that did not make a DNS request, but there is a problem. All but one of the IPs in Figure 13 are legitimate resources performing normal system operations. Benign programs previously established connections, and local network resources are all likely to be false positives when strictly implementing the TCP-without-DNS block rules. Therefore, it is essential to properly vet IP addresses before blocking them. During the testing phases for the proposed PoC, Amazon, Google, and Microsoft were the biggest contributors to false positives. To first ensure a trusted vendor isn't responsible for the IP address in the suspicious list, a parser can first be used to look up the IP information. The following PowerShell script is one example of how a user can automate the lookup process (2006, Snover):

```
Get-Content suspicious.txt | foreach-object
{[System.Net.Dns]::GetHostbyAddress("$_")}
```

Once a list of domain-to-IP mappings return, an analyst can often discern benign domains from malicious domains based solely on their registration information. Approximately 70% of the false positives returned during the PoC development were the globally recognized vendors previously discussed (google, yahoo, Microsoft etc) or IPs of servers run by the developer. The false positives that did not have a domain associated

© 2020 The SANS Institute Author retains full rights.

were often a part of legitimate cloud services where the specific IP address was not associated with a particular domain, but rather an update resource cached by other programs on the system.

8. Conclusion

Achieving a heightened security posture for an endpoint system begins with awareness and system visibility. It is certainly possible to take a proactive approach to endpoint security using freely available tools and techniques, however, it is always more complicated than it first appears. Benign software on endpoint systems may locate resources through less common methods not accounted for or that have pre-coded IP pools that they can use to make external connections. The proposed PoC was successful in its attempts to block direct IP connections, but it also blocked legitimate resources. The proposed PoC can help any endpoint achieve greater network visibility by keeping track of DNS requests, parsing DNS responses, logging TCP connection attempts, and writing out a shortlist for research. The current implementation, however, is best used for awareness. The false positives in each trial were typically easy to rule out as malicious, but even still, the PoC has ample room to grow before it should be used on a productive endpoint. The results of mismatched connections to DNS entries still requires an analyst to make judgment calls to avoid some system inefficiencies. Therefore, each operating system's firewall is better updated as-needed only after analyst research. While the PoC provides a good starting point for future automation, the desired result still relies on human interaction.

© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

Full Rights

References

Browne, R. (2018, May 30). 70% of people globally work remotely at least once a week, study says Retrieved 3 September 2019, from CNBC.com. Available at: https://www.cnbc.com/2018/05/30/70-percent-of-people-globally-work-remotely-at-least-once-a-week-iwg-study.html

How DNS works. (n.d.). Retrieved August 1, 2019, from https://www.cloudflare.com/learning/dns/what-is-dns/.

IETF. (1981, September). Transmission Control Protocol. Retrieved April 4, 2020, from https://tools.ietf.org/html/rfc793

Lord, N. (2018, December 21). What is Endpoint Security Data Protection 101. Retrieved 12 August 2019, from https://digitalguardian.com/blog/what-endpoint-security-data-protection-101

Murray, K. (2019, April 18). Top Threats to Endpoints and How To Stay Protected. Retrieved from https://www.brighttalk.com/webcast/288/348505/top-threats-to-endpoints-and-how-to-stay-protected

Shepherd, Maddie. (2019, July 23)"11 Surprising Working From Home Statistics." *Fundera*, Fundera, 23 July 2019, https://www.fundera.com/resources/working-from-home-statistics [Accessed 3 Sep. 2019].

Snover, Jeffrey (2006, July). Windows PowerShell One Liner: Name to IP Address. Retrieved September 9, 2019, from https://devblogs.microsoft.com/powershell/windows-powershell-one-liner-name-to-ip-address/

Stevens, D. (2019, June 16). Sysmon Version 10: DNS Logging. Retrieved September 19, 2019, from https://isc.sans.edu/diary/Sysmon+Version+10%3A+DNS+Logging/25036

US Legal, Inc. (2018) Small Office/Home Office [SOHO] Law and Legal Definition. Retrieved July 14, 2019 from https://definitions.uslegal.com/s/small-office-home-office-soho

© 2020 The SANS Institute Author retains full rights.

Appendix A
Python Code:

1) Cross Platform Sniffer

```
#Must have winPcap installed for windows version to work: 
www.winpcap.org


from scapy.all import *
from scapy.utils import PcapWriter
from datetime import *#datetime, timedelta
import os, sys, signal


#Defines then runs a signal handler because the 'while' loop can get 
sticky in the console. This helps ctrl-c out nicely.
def signal_handler(signal, frame):
    print("\nprogram exiting gracefully")
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)
winplat = ["win32", "win64"]
linux = ["linux","linux2","linux3"]
platform = sys.platform


if platform in winplat:
      print("Windows")
      now = datetime.now()
      stop = now + timedelta(seconds=120) #amount of time the script 
will run for. Can do minutes=x or hours=x
      flushdns = os.system("ipconfig /flushdns") # flush out the 
windows DNS cache : ipconfig /displaydns to see
      #os.system("pythonw.exe firewallrules.py")


      try:
            while datetime.now() < stop:
                  #TCP Syn + SynAck packet capture and DNS port 53 
packet capture.
                  packets = sniff(filter="tcp[tcpflags] & (tcp-syn) 
!=0 or port 53",
                        session=IPSession,  # defragment on-the-flow
```


© 2020 The SANS Institute Author retains full rights.

```
                                count=2,# 2 packets before the script
continues
                                prn=lambda x: x.summary()) #prints out the
tcp syn/synack and DNS req/resp


                          #Append to 'sniffed.pcap' all Syn/Ack traffic or
port 53 request/responses.
                                 # Will create the file if it doesn't already
exist
                          pktdump1 = PcapWriter("sniffed.pcap", append=True,
sync=True)
                          pktdump1.write(packets)


                          pcap = 'sniffed.pcap'
                          pkts = rdpcap(pcap)
                          UDPipS = []
                          TCPipS = []


      #Add TCP destination packets to a temp list for follow on comparison
                          for packet in pkts:
                                if packet.haslayer(TCP):
                                      TCPipS.append(packet[IP].dst)
      # DNS parsing for to record multiple DNS entries and extract/normalize
PTR requests
      ## Without this loop only the first DNS entry will be returned and the
PTR records will be missed.
                                if packet.haslayer(UDP): # Triggers if a UDP
packet
                                      UDPipS.append(packet[IP].dst) #Adds the
packet to ta temp list for PCAP incl
                                      if packet.haslayer(DNSRR): # If there
is a DNS Response
                                            a_count = packet[DNS].ancount
#Find how many answers returned
                                            i = a_count + 4
                                            arp = "arpa"
                                            while i > 4:
                                                  if
str(packet[0][i].rdata)[0].isnumeric():
```


© 2020 The SANS Institute Author retains full rights.

```
        #print(packet[0][i].rdata)

        UDPipS.append(packet[0][i].rdata)
                                                            #Useing 'count' to see if
the telltale PTR lookup string is in the rrname field
                                                            elif
packet[0][i].rrname.decode().count("in-addr.arpa")>0:

        #print(packet[0][i].rrname.decode())
                                                                  base
=(packet[0][i].rrname.decode())
                                                                  chop = base[:-14]
                                                                  work =
chop.split('.')
                                                                  final =
work[3]+"."+work[2]+"."+work[1]+"."+work[0]

        UDPipS.append(final)
                                                            i -= 1
                                        inTnotU = list(set(TCPipS)-set(UDPipS))

                                 with open('suspicious.txt','w+') as f:
                                        for i in inTnotU:
                                               f.write(str(i))
                                               f.write("\n") #Added because the above
line does not allow for iteration. This was a work around.
                                        f.close()
               except KeyboardInterrupt:
                      print('Sniffer turned off!')
        elif platform in linux:
               print("Linux")
               now = datetime.now()
               stop = now + timedelta(seconds=120) #amount of time the script
will run for. Can do minutes=x or hours=x

               try:
                      while datetime.now() < stop:
```


© 2020 The SANS Institute Author retains full rights.

```
                        #TCP Syn + SynAck packet capture and DNS port 53
packet capture.
                        packets = sniff(filter="tcp[tcpflags] & (tcp-syn)
!=0 or port 53",
                              session=IPSession,  # defragment on-the-flow
                              count=2,# 2 packets before the script
continues
                              prn=lambda x: x.summary()) #prints out the
tcp syn/synack and DNS req/resp


                        #Append to 'sniffed.pcap' all Syn/Ack traffic or
port 53 request/responses.
                               # Will create the file if it doesn't already
exist
                        pktdump1 = PcapWriter("sniffed.pcap", append=True,
sync=True)
                        pktdump1.write(packets)


                        pcap = 'sniffed.pcap'
                        pkts = rdpcap(pcap)
                        UDPipS = []
                        TCPipS = []


      #Add TCP destination packets to a temp list for follow on comparison
                        for packet in pkts:
                              if packet.haslayer(TCP):
                                    TCPipS.append(packet[IP].dst)
      # DNS parsing for to record multiple DNS entries and extract/normalize
PTR requests
      ## Without this loop only the first DNS entry will be returned and the
PTR records will be missed.
                              if packet.haslayer(UDP): # Triggers if a UDP
packet
                                    UDPipS.append(packet[IP].dst) #Adds the
packet to ta temp list for PCAP incl
                                    if packet.haslayer(DNSRR): # If there
is a DNS Response
                                          a_count = packet[DNS].ancount
#Find how many answers returned
                                          i = a_count + 4
```


© 2020 The SANS Institute Author retains full rights.

```
                                        arp = "arpa"
                                        while i > 4:
                                            if
str(packet[0][i].rdata)[0].isdigit():

    #print(packet[0][i].rdata)

    UDPipS.append(packet[0][i].rdata)
                                            #Useing 'count' to see if
the telltale PTR lookup string is in the rrname field
                                            elif
packet[0][i].rrname.decode().count("in-addr.arpa")>0:

    #print(packet[0][i].rrname.decode())
                                                base
=(packet[0][i].rrname.decode())

                                                chop = base[:-14]
                                                work =
chop.split('.')

                                                final =
work[3]+"."+work[2]+"."+work[1]+"."+work[0]

    UDPipS.append(final)
                                            i -= 1
                                inTnotU = list(set(TCPipS)-set(UDPipS))


                        with open('suspicious.txt','w+') as f:
                                for i in inTnotU:
                                    f.write(str(i))
                                    f.write("\n") #Added because the above
line does not allow for iteration. This was a work around.
                                f.close()
                except KeyboardInterrupt:
                        print('Sniffer turned off!')
        elif platform == "darwin":
                print("MacOS")
                now = datetime.now()
                stop = now + timedelta(seconds=120) #amount of time the script
will run for. Can do minutes=x or hours=x
```


© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

```
            try:
                  while datetime.now() < stop:
                        #TCP Syn + SynAck packet capture and DNS port 53 
packet capture.
                        packets = sniff(filter="tcp[tcpflags] & (tcp-syn) 
!=0 or port 53",
                              session=IPSession,  # defragment on-the-flow
                              count=2,# 2 packets before the script 
continues
                              prn=lambda x: x.summary()) #prints out the 
tcp syn/synack and DNS req/resp

                        #Append to 'sniffed.pcap' all Syn/Ack traffic or 
port 53 request/responses.
                               # Will create the file if it doesn't already 
exist
                        pktdump1 = PcapWriter("sniffed.pcap", append=True, 
sync=True)
                        pktdump1.write(packets)

                        pcap = 'sniffed.pcap'
                        pkts = rdpcap(pcap)
                        UDPipS = []
                        TCPipS = []

      #Add TCP destination packets to a temp list for follow on comparison
                        for packet in pkts:
                              if packet.haslayer(TCP):
                                    TCPipS.append(packet[IP].dst)
      # DNS parsing for to record multiple DNS entries and extract/normalize 
PTR requests
      ## Without this loop only the first DNS entry will be returned and the 
PTR records will be missed.
                              if packet.haslayer(UDP): # Triggers if a UDP 
packet
                                    UDPipS.append(packet[IP].dst) #Adds the 
packet to ta temp list for PCAP incl
```


© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

```
                                        if packet.haslayer(DNSRR): # If there
is a DNS Response
                                                a_count = packet[DNS].ancount
#Find how many answers returned
                                                i = a_count + 4
                                                arp = "arpa"
                                                while i > 4:
                                                        if
str(packet[0][i].rdata)[0].isdigit():
        #print(packet[0][i].rdata)
        UDPipS.append(packet[0][i].rdata)
                                                        #Useing 'count' to see if
the telltale PTR lookup string is in the rrname field
                                                        elif
packet[0][i].rrname.decode().count("in-addr.arpa")>0:
        #print(packet[0][i].rrname.decode())
                                                                base
=(packet[0][i].rrname.decode())
                                                                chop = base[:-14]
                                                                work =
chop.split('.')
                                                                final =
work[3]+"."+work[2]+"."+work[1]+"."+work[0]
        UDPipS.append(final)
                                                        i -= 1
                                inTnotU = list(set(TCPipS)-set(UDPipS))

                        with open('suspicious.txt','w+') as f:
                                for i in inTnotU:
                                        f.write(str(i))
                                        f.write("\n") #Added because the above
line does not allow for iteration. This was a work around.
                                f.close()
                except KeyboardInterrupt:
                        print('Sniffer turned off!')
        else:
```


© 2020 The SANS Institute Author retains full rights.

```
        print("whoknowsman")
```

2) Windows Firewall Automation Follow on Script

```
import os, time, signal, sys
import os.path
from os import path

def signal_handler(signal, frame):
        print("\nCleaning firewall rules and exiting gracefully")
        for i in list:
                os.system('netsh advfirewall firewall delete rule 
name="{}"'.format(i))
        sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)

while not path.exists("suspicious.txt"):  #Just giving the sniffer 
script a few seconds to create a pcap and 'suspicious.txt'
        time.sleep(3)

#if suspicious.txt exists, start a loop
while path.exists("suspicious.txt"):
#       signal.signal(signal.SIGINT, signal_handler)
        list = []
        inTnotU = []
        f = open('suspicious.txt', 'r+')
        for i in f:
                inTnotU.append(i)
        f.close()
        for i in inTnotU:
                i=(i[:-1])
                if i not in list:
                        list.append(i)
        for i in list:
                os.system('netsh advfirewall firewall add rule name="{}" 
dir=out interface=any action=block remoteip={}'.format(i,i))
        time.sleep(15)
```


© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights

```
        for i in list:
                os.system('netsh advfirewall firewall delete rule
name="{}"'.format(i))
```


© 2020 The SANS Institute Author retains full rights.

© 2020 The SANS Institute, Author Retains Full Rights