Jay Chow, Nihal Shah, Wu Guan Long, Shivani Deshpande, Abdullah Alghofaili

EN.650.431 Ethical Hacking Homework 1

20th February 2019

Background:

Commercial Wifi-based UAVs are vulnerable to common and basic security attacks, capable by hackers. The standard ARDiscovery Connection Process and the Wifi access point used in the Parrot Bebop UAV are exploitable such that the ability to fly can be disrupted mid-flight by a remote attacker. The Normal operation such as ARDiscovery Connection process over Wifi was exploited using a fuzzing technique to discover that the Parrot Bebop UAV is vulnerable to DoS and buffer-overflow attacks during the ARDiscovery Connection process. The exploitation resulted in catastrophic and immediate disabling of the UAV's rotors mid-flight. The UAC was also vulnerable to ARP Cache poisoning attack, which can disconnect the disconnect the mobile device user, causing the UAV to land or return home. We learned that Wifi-based commercial UAVs require a defense-in depth approach.

The academic paper attempts to address the question: Are these devices secured in a way that they are safe to operate and therefore protected against exploitation? We learned specifically about UAV vulnerabilities. For example, Defcon 23 hackers were able to exploit open telnet access and issue a deauthentication attack to force Parrot AR and Bebob UAVs from the air. In this work, the author utilized the telnet application to fuzz(i.e point telnet at port 44444 to deliver fuzzed JSON records without logging into the telnet server on the UAC or kill any processes). There is another attack called Skyjack which exploits the UAV's onboard Wifi through a de-authentication technique.

We learned that the ARDiscovery process is the process by which networked devices identify and connect to nearby nodes. The purpose of the protocol is to identify and permit the connection of a wireless device to an existing network. More specifically, the Parrot uses ARDiscovery to establish a connection between the aircraft running the software and a controller. The discovery protocol works over a combination of TCP and UDP channels, establishing a handshake over a TCP port. The controller initiates a TCP handshake from UAV to controller and controller to UAV in order to establish the communication channel. Data sent between the connected devices is managed through JSON records sent via UDP. For example, the controller would first send a JSON record to port 44444 on the UAV and the UAV would respond back to the controller with a JSON record if it does not have a controller(controller accepted). Otherwise, the UAV would respond back to the controller with another JSON record(controller rejected).

Exploit 1: Buffer-Overflow

The results of fuzzing the ARDiscovery process by increasing the number of characters in the first field of the JSON record sent by a potential controller, revealed that the the developers did not consider this case. When a valid controller was already flying the UAV, the UAV was unable to handle JSON records sent from potential controllers with the first field larger than 931 characters.

Exploit 2: DoS Attack

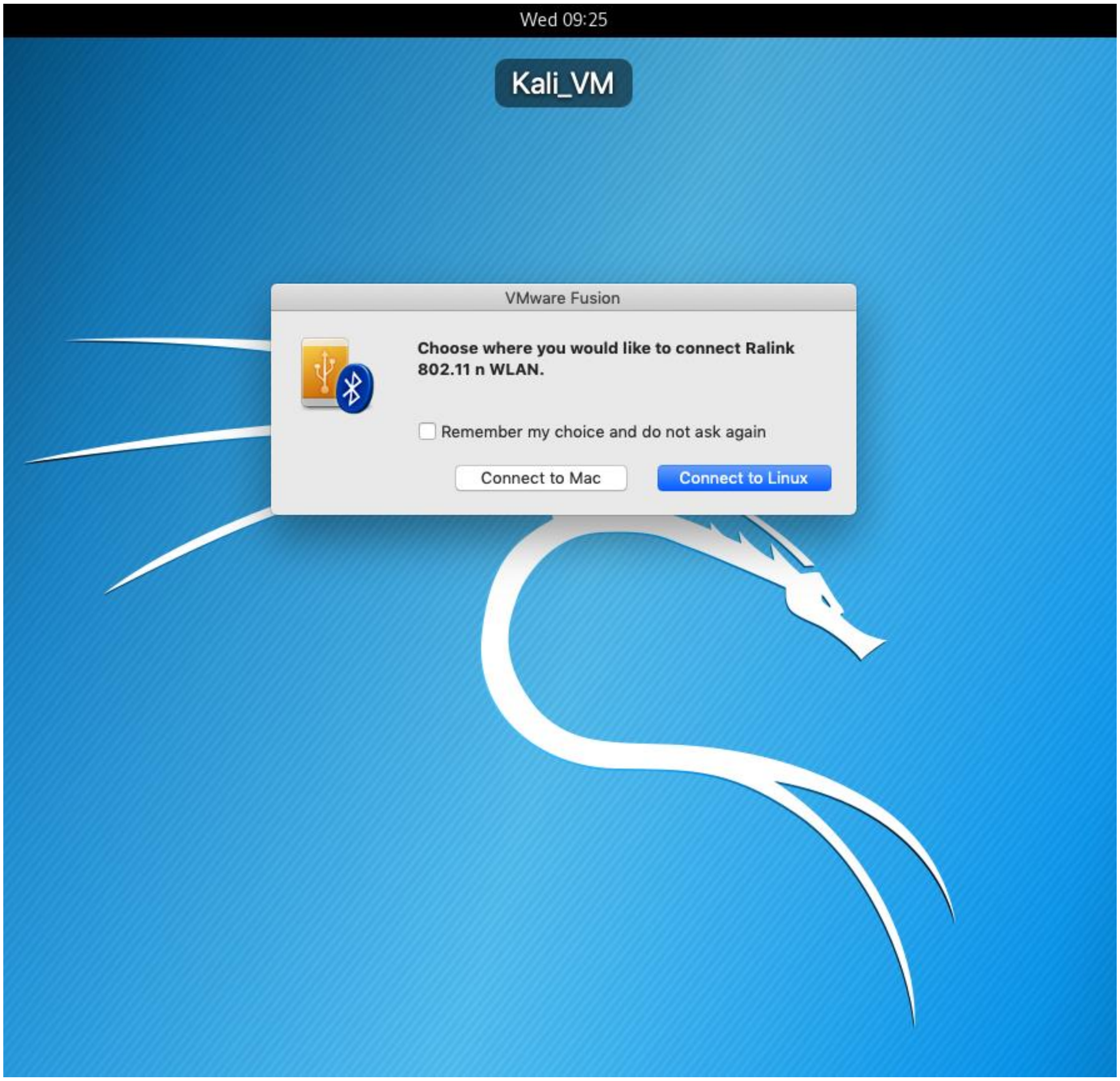
Fuzzing the ARDiscovery with simultaneous requests from other controllers reveal that Parrot developers did not account for this scenario and the UAV was incapable of handling up to 1000 simultaneous requests from other controllers while a valid controller was already flying the UAV

Exploit 3: ARP Cache Poison Attack

The result of the ARP Cache Poisoning experiment disconnected the primary smartphone controller. When an attack laptop spoofs the UAV’s IP address, there is a conflict on the UAV’s wireless network and thus no communication can occur between the controller and the UAV.

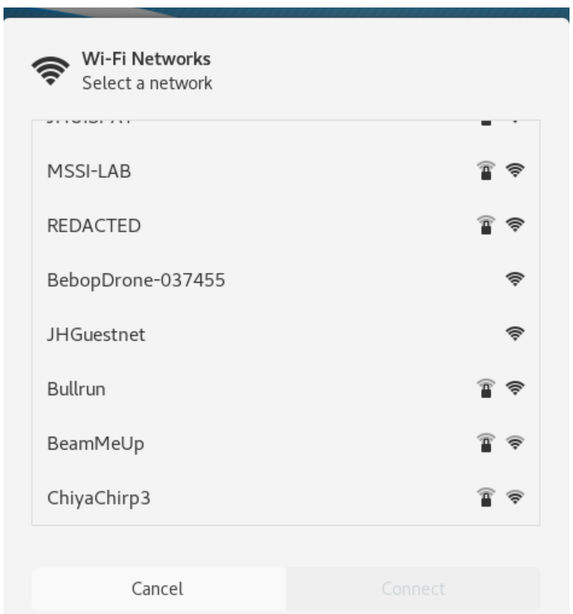
Setup:

After connecting the external wireless adapter,

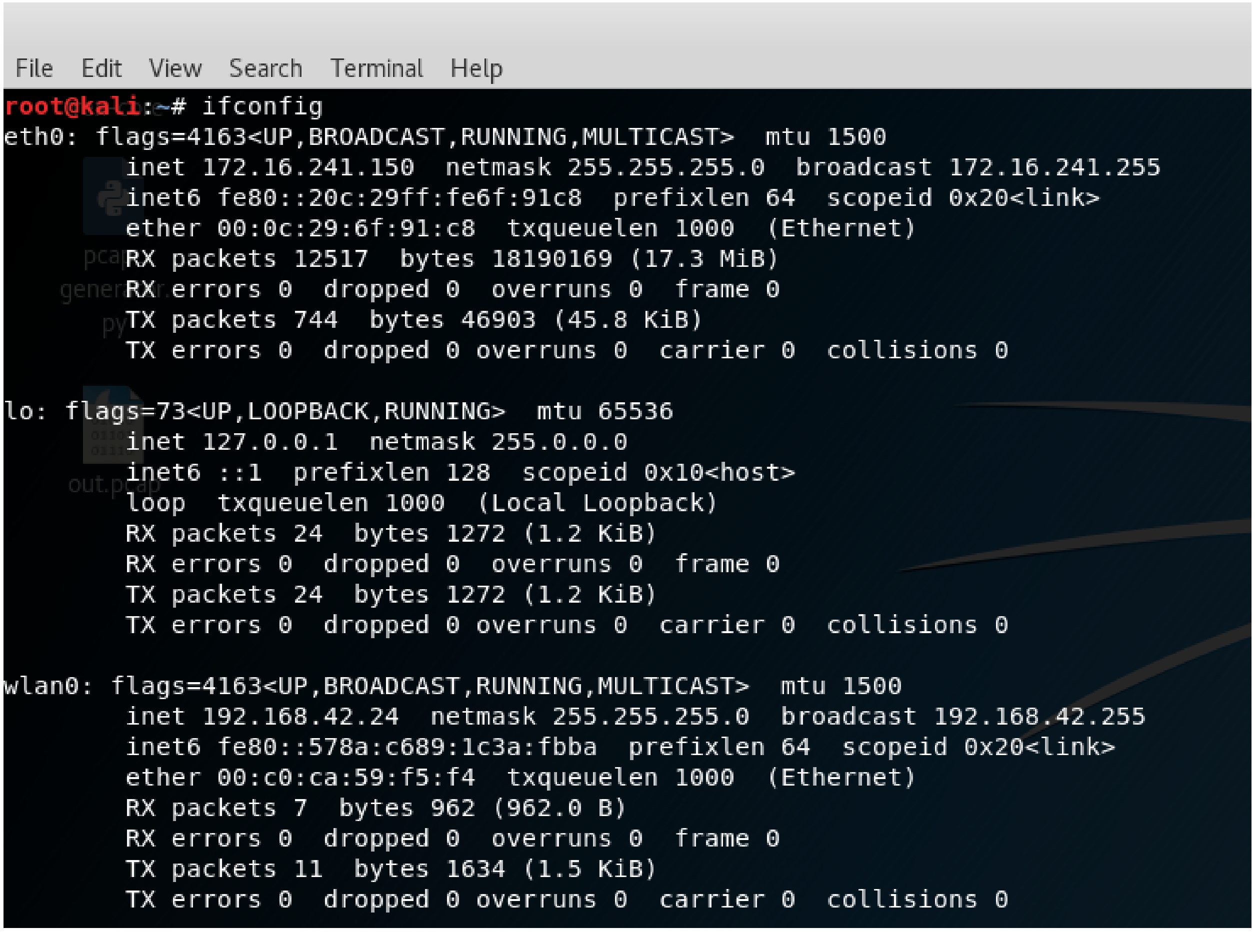


We clicked “connect to Linux”. Then, we pressed the button on the Bebop 1, which created a wireless LAN(WLAN). Note that the green light was flickering after we pressed the button.

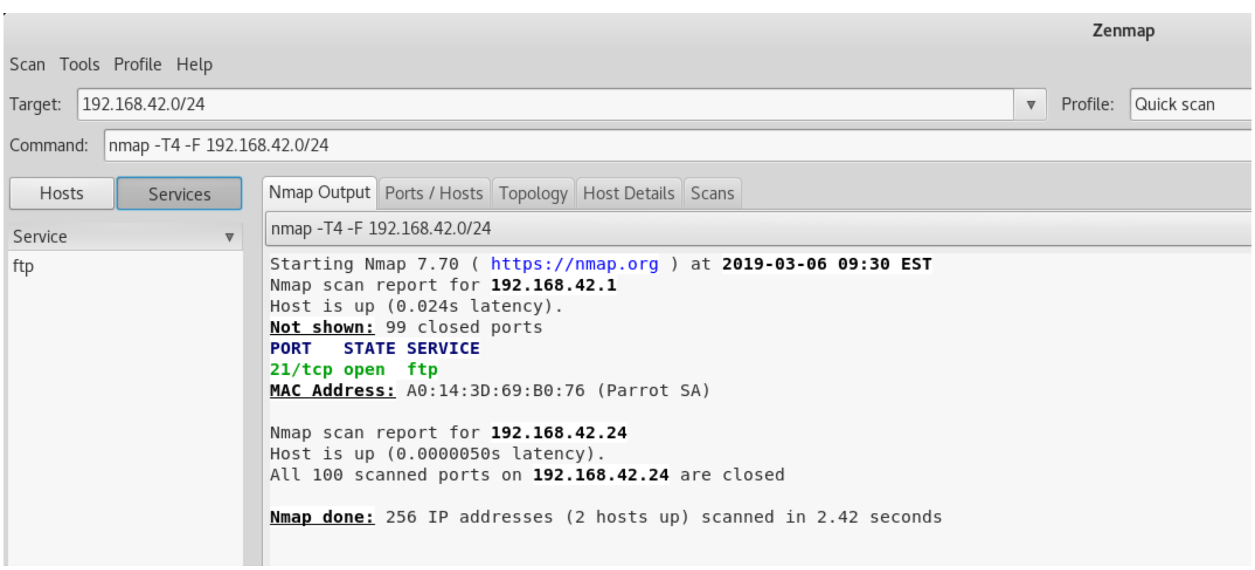
Now, the WLAN was created. As shown below, we could view “BebopDrone-037455” network in Kali.



After connecting to the Bebop 1, we could see that Kali has an IP address 192.168.42.24 on wlan0:

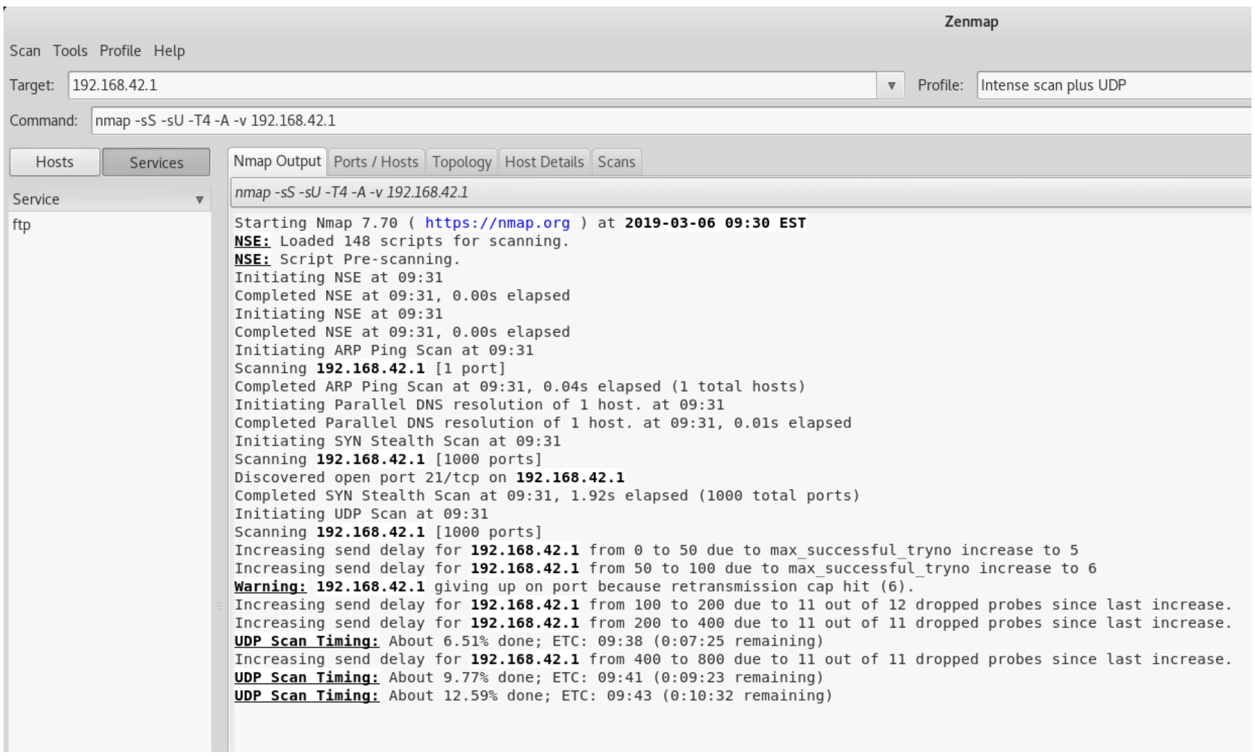


Now since we know that the network is 192.168.42.X, we conducted zenmap(which uses nmap) to map out the devices on this network:



As shown above, we did a quick scan for the most common tcp ports. Note that zenmap uses nmap that scans the most common 1000 ports for each scanned protocol. We could therefore determine that the IP address of the Bebop is 192.168.42.1 since we could see the MAC address of A0:14:3D:69:B0:76 was associated with the Parrot SA.

b) As shown above, the opened tcp port that was 21 running ftp service. Next, we did an intense scan plus UDP on zenmap to find the UDP ports that were opened:



This screenshot above shows that our intense zenmap scan was in progress. The result of this scan was saved in text file on the next page.

Open | bebop_1.nmap ~/Desktop/ethical_hacking/homework1 | Save

```
Starting Nmap 7.70 ( https://nmap.org ) at 2019-03-06 09:30 EST
NSE: Loaded 148 scripts for scanning.
NSE: Script Pre-scanning.
Initiating NSE at 09:31
Completed NSE at 09:31, 0.00s elapsed
Initiating NSE at 09:31
Completed NSE at 09:31, 0.00s elapsed
Initiating ARP Ping Scan at 09:31
Scanning 192.168.42.1 [1 port]
Completed ARP Ping Scan at 09:31, 0.04s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 09:31
Completed Parallel DNS resolution of 1 host. at 09:31, 0.01s elapsed
Initiating SYN Stealth Scan at 09:31
Scanning 192.168.42.1 [1000 ports]
Discovered open port 21/tcp on 192.168.42.1
Completed SYN Stealth Scan at 09:31, 1.92s elapsed (1000 total ports)
Initiating UDP Scan at 09:31
Scanning 192.168.42.1 [1000 ports]
Increasing send delay for 192.168.42.1 from 0 to 50 due to max_successful_tryno increase to 5
Increasing send delay for 192.168.42.1 from 50 to 100 due to max_successful_tryno increase to 6
Warning: 192.168.42.1 giving up on port because retransmission cap hit (6).
Increasing send delay for 192.168.42.1 from 100 to 200 due to 11 out of 12 dropped probes since last increase.
Increasing send delay for 192.168.42.1 from 200 to 400 due to 11 out of 11 dropped probes since last increase.
UDP Scan Timing: About 6.51% done; ETC: 09:38 (0:07:25 remaining)
Increasing send delay for 192.168.42.1 from 400 to 800 due to 11 out of 11 dropped probes since last increase.
UDP Scan Timing: About 9.77% done; ETC: 09:41 (0:09:23 remaining)
UDP Scan Timing: About 12.59% done; ETC: 09:43 (0:10:32 remaining)
UDP Scan Timing: About 31.50% done; ETC: 09:45 (0:09:56 remaining)
UDP Scan Timing: About 38.47% done; ETC: 09:45 (0:09:09 remaining)
UDP Scan Timing: About 44.30% done; ETC: 09:46 (0:08:23 remaining)
UDP Scan Timing: About 50.13% done; ETC: 09:46 (0:07:35 remaining)
UDP Scan Timing: About 55.44% done; ETC: 09:46 (0:06:48 remaining)
UDP Scan Timing: About 60.87% done; ETC: 09:46 (0:06:01 remaining)
UDP Scan Timing: About 66.10% done; ETC: 09:46 (0:05:14 remaining)
UDP Scan Timing: About 71.30% done; ETC: 09:46 (0:04:27 remaining)
UDP Scan Timing: About 76.83% done; ETC: 09:46 (0:03:37 remaining)
UDP Scan Timing: About 81.83% done; ETC: 09:46 (0:02:50 remaining)
UDP Scan Timing: About 87.26% done; ETC: 09:46 (0:02:00 remaining)
```

```
Discovered open port 5353/udp on 192.168.42.1
UDP Scan Timing: About 92.57% done; ETC: 09:46 (0:01:10 remaining)
Completed UDP Scan at 09:47, 981.21s elapsed (1000 total ports)
Initiating Service scan at 09:47
Scanning 16 services on 192.168.42.1
Service scan Timing: About 18.75% done; ETC: 09:51 (0:03:28 remaining)
Service scan Timing: About 68.75% done; ETC: 09:49 (0:00:35 remaining)
Completed Service scan at 09:49, 97.56s elapsed (16 services on 1 host)
Initiating OS detection (try #1) against 192.168.42.1
NSE: Script scanning 192.168.42.1.
Initiating NSE at 09:49
Completed NSE at 09:49, 1.25s elapsed
Initiating NSE at 09:49
Completed NSE at 09:49, 1.01s elapsed
Nmap scan report for 192.168.42.1
Host is up (0.044s latency).
Not shown: 1984 closed ports
PORT      STATE         SERVICE     VERSION
21/tcp    open          ftp         BusyBox ftpd (D-Link DCS-932L IP-Cam camera)
| ftp-anon: Anonymous FTP login allowed (FTP code 230)
| total 4
|_drwxr-xr-x    8 0        0             4096 Jan  1  1970 internal_000
|_ftp-bounce: bounce working!
| ftp-syst:
|   STAT:
| Server status:
|  TYPE: BINARY
|_Ok
67/udp    open|filtered dhcps
139/udp   open|filtered netbios-ssn
515/udp   open|filtered printer
1056/udp  open|filtered vfo
```

```
5353/udp  open          mdns        DNS-based service discovery
| dns-service-discovery:
|   9/tcp workstation
|     Address=192.168.42.1
|   44444/udp arsdk-0901
|     {"device_id":"PI040338AA5B037455"}
|_    Address=192.168.42.1
5555/udp  open|filtered rplay
18666/udp open|filtered unknown
18683/udp open|filtered unknown
19682/udp open|filtered unknown
19792/udp open|filtered unknown
20409/udp open|filtered unknown
21524/udp open|filtered unknown
24854/udp open|filtered unknown
30704/udp open|filtered unknown
53006/udp open|filtered unknown
MAC Address: A0:14:3D:69:B0:76 (Parrot SA)
Device type: general purpose
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.10
Uptime guess: 0.012 days (since Wed Mar  6 09:31:30 2019)
Network Distance: 1 hop
TCP Sequence Prediction: Difficulty=260 (Good luck!)
IP ID Sequence Generation: All zeros
Service Info: Device: webcam; CPE: cpe:/h:dlink:dcs-932l

TRACEROUTE
HOP RTT      ADDRESS
1   43.58 ms 192.168.42.1

NSE: Script Post-scanning.
Initiating NSE at 09:49
Completed NSE at 09:49, 0.00s elapsed
Initiating NSE at 09:49
Completed NSE at 09:49, 0.00s elapsed
Read data files from: /usr/bin/../share/nmap
OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
```

```
Nmap done: 1 IP address (1 host up) scanned in 1087.60 seconds
           Raw packets sent: 2469 (88.815KB) | Rcvd: 2085 (100.809KB)
```

Therefore, the opened UDP ports were 67, 139, 515, 1056, 5353(MDNS, a DNS-based service discovery), 5555, 18666, 18683, 19682, 19792, 20409, 21524, 24854, 30704 and 53006.

c) The OS used in the Bebop is Linux 2.6.32 - 3.10, as shown from the result of the intense scan:

```
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.10
```

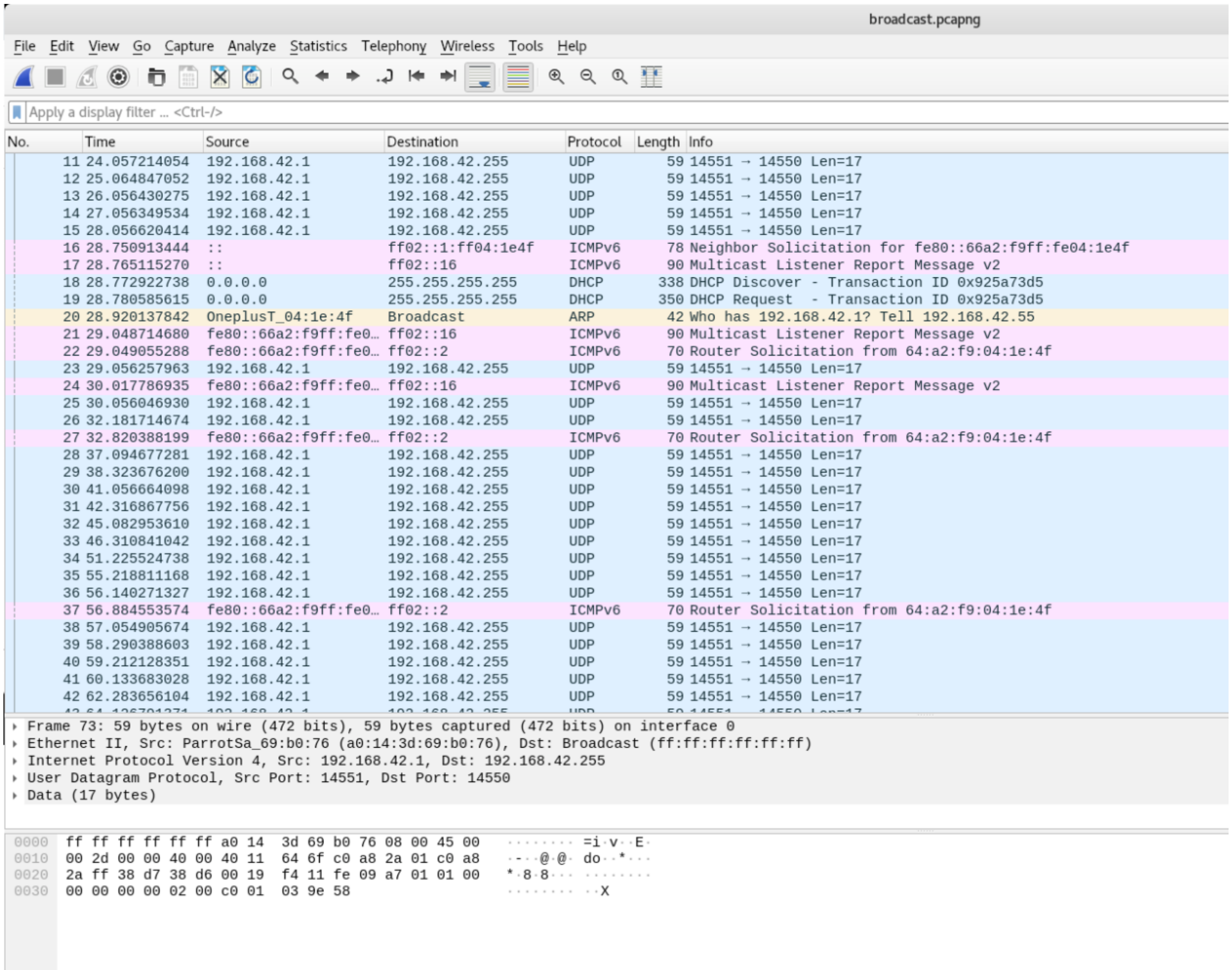
d) The new Bebop 2 ARDiscovery Process uses the MDNS protocol.

The MDNS protocol was initially used more in home networks. But now it is used in corporate networks. The idea is about plug and play. On a high level, there is a need for directory service to find IP addresses using hostnames(computerA.local). There can be a localhost multicast capability in a local VLAN, and the devices in the same VLAN can therefore shake hands to discover one another. Recall that a device that has only been authenticated will be put in a VLAN.

The purpose of multicast DNS is to resolve hostnames to IP addresses within small networks that do not have a local DNS server. It is a zero configuration service that is similar to the unicast Domain Name System. The mDNS protocol is published as RFC 6762 and uses the IP multicast UDP packets, implemented by Apple's Bonjour and Linux nss-mdns services. When an MDNS client in a local network needs to resolve a hostname to IP address, it sends an IP multicast query message that asks the host having that name to identify itself. The target machine then multicasts a message that includes its IP address. All machines in the network can then use that information update their mDNS caches. By default, MDNS resolves host names ending with .local domain.

As such the drone, the controller(iPhone) and my Kali box discovered one another via MDNS protocol on the same VLAN when both the Kali, with the external WiFi Card, and the controller connected to the drone's Wireless Access Point.

e) The JSON records were replaced MDNS query and response packets. During the connection process, we captured all the packets on wlan0 via Wireshark:


broadcast.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

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

No.	Time	Source	Destination	Protocol	Length	Info
11	24.057214054	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
12	25.064847052	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
13	26.056430275	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
14	27.056349534	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
15	28.056620414	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
16	28.750913444	::	ff02::1:ff04:1e4f	ICMPv6	78	Neighbor Solicitation for fe80::66a2:f9ff:fe04:1e4f
17	28.765115270	::	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
18	28.772922738	0.0.0.0	255.255.255.255	DHCP	338	DHCP Discover - Transaction ID 0x925a73d5
19	28.780585615	0.0.0.0	255.255.255.255	DHCP	350	DHCP Request - Transaction ID 0x925a73d5
20	28.920137842	OneplusT_04:1e:4f	Broadcast	ARP	42	Who has 192.168.42.1? Tell 192.168.42.55
21	29.048714680	fe80::66a2:f9ff:fe0…	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
22	29.049055288	fe80::66a2:f9ff:fe0…	ff02::2	ICMPv6	70	Router Solicitation from 64:a2:f9:04:1e:4f
23	29.056257963	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
24	30.017786935	fe80::66a2:f9ff:fe0…	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
25	30.056046930	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
26	32.181714674	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
27	32.820388199	fe80::66a2:f9ff:fe0…	ff02::2	ICMPv6	70	Router Solicitation from 64:a2:f9:04:1e:4f
28	37.094677281	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
29	38.323676200	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
30	41.056664098	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
31	42.316867756	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
32	45.082953610	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
33	46.310841042	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
34	51.225524738	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
35	55.218811168	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
36	56.140271327	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
37	56.884553574	fe80::66a2:f9ff:fe0…	ff02::2	ICMPv6	70	Router Solicitation from 64:a2:f9:04:1e:4f
38	57.054905674	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
39	58.290388603	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
40	59.212128351	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
41	60.133683028	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17
42	62.283656104	192.168.42.1	192.168.42.255	UDP	59	14551 → 14550 Len=17

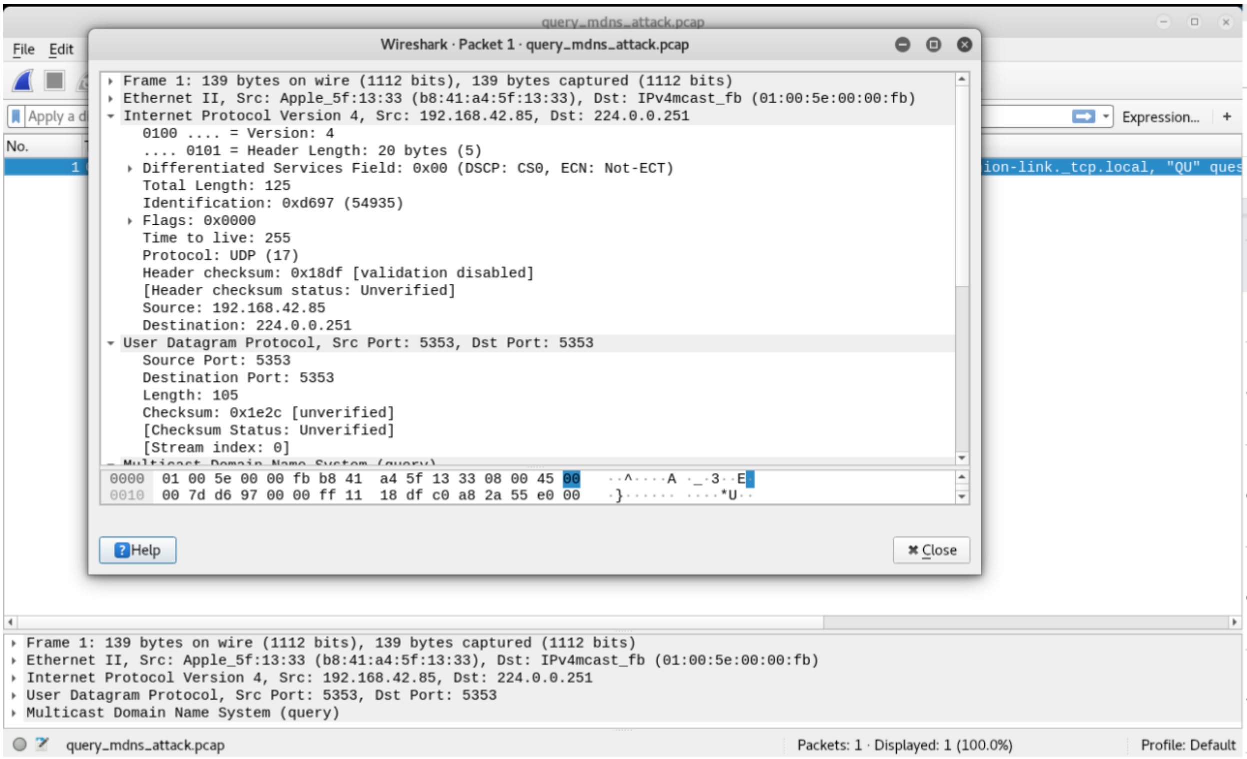
```
Frame 73: 59 bytes on wire (472 bits), 59 bytes captured (472 bits) on interface 0
Ethernet II, Src: ParrotSa_69:b0:76 (a0:14:3d:69:b0:76), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Internet Protocol Version 4, Src: 192.168.42.1, Dst: 192.168.42.255
User Datagram Protocol, Src Port: 14551, Dst Port: 14550
Data (17 bytes)
```

```
0000  ff ff ff ff ff ff a0 14  3d 69 b0 76 08 00 45 00
0010  00 2d 00 00 40 00 40 11  64 6f c0 a8 2a 01 c0 a8
0020  2a ff 38 d7 38 d6 00 19  f4 11 fe 09 a7 01 01 00
0030  00 00 00 00 02 00 c0 01  03 9e 58
```

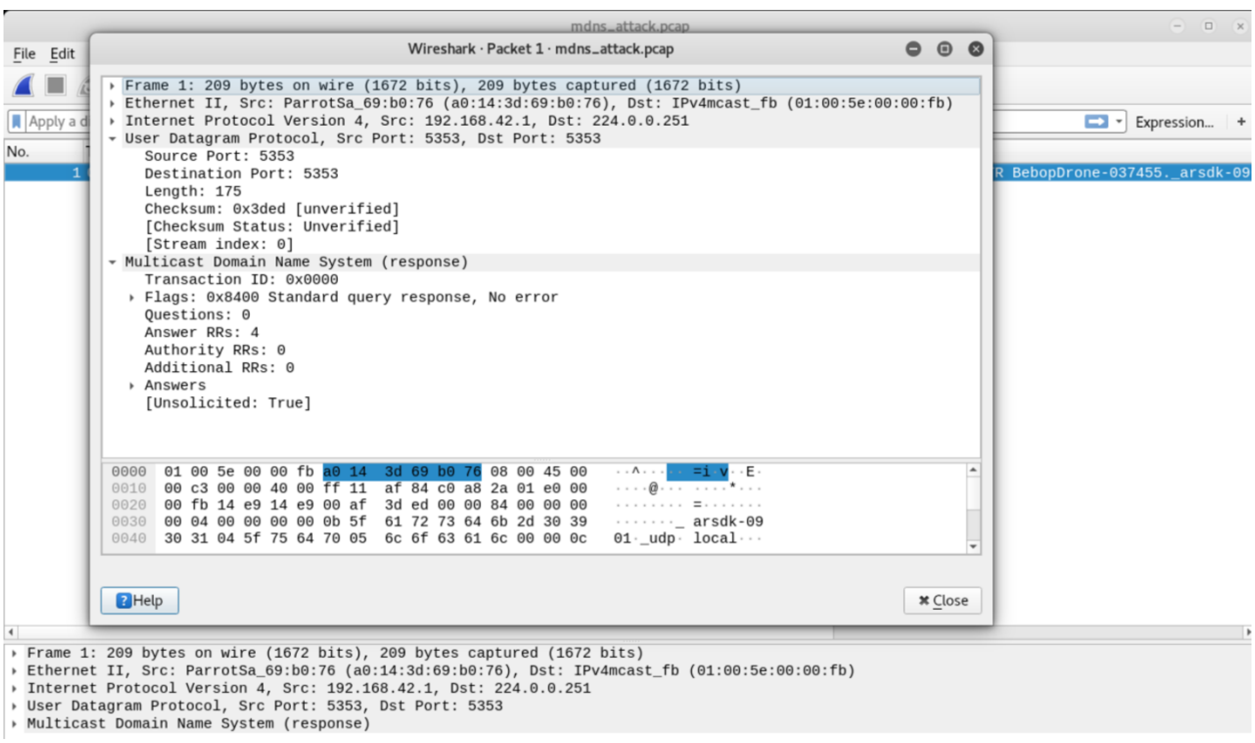
Note that 192.168.42.1 is the IP of the drone. There are UDP, ARP, MDNS and ICMPv6 packets.

For the discovery process, we managed to capture both a MDNS query packet and a MDNS response packet. MDNS uses the UDP protocol.

A MDNS query packet:



Also, we captured a MDNS response packet, which used to conduct a DOS attack on the drone:

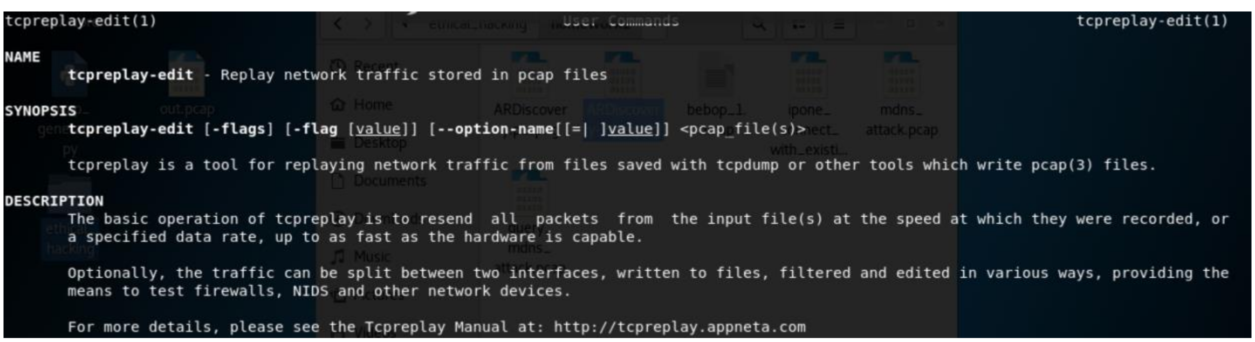


2a) We can wage a denial of service (DoS) attack against the Bebop ARDiscovery process by using tcpreplay in linux using a specific MDNS response packet from 192.168.42.1(drone) to 224.0.0.251. The weakness lies in the use of MDNS packets in the discovery process. Both the drone and the controller require the MDNS protocol to discover each other on the same wireless network.

We used Tcpreplay tool, which works in a linux machine, to initiate the DoS attack on the drone. The Tcpreplay tool will take a pcap file as an input of an already saved traffic which in this case is the mDNS transaction between the controller and the drone to resend the traffic again on a selected interface. The tool has an option to set the speed of the traffic, which is not used in this case, and also has an option to loop the traffic as many times as the user desire. In this assignment, we identified MDNS packets that contains JSON records into a pcap file to use them with this tool. The file is then resent to the drone several times using the loop flag to hopefully overflow the machine. The command used for the attack is as follows:

tcpreplay -i wlan0 --loop 10000 <pcap file of single MDNS reponse packet>

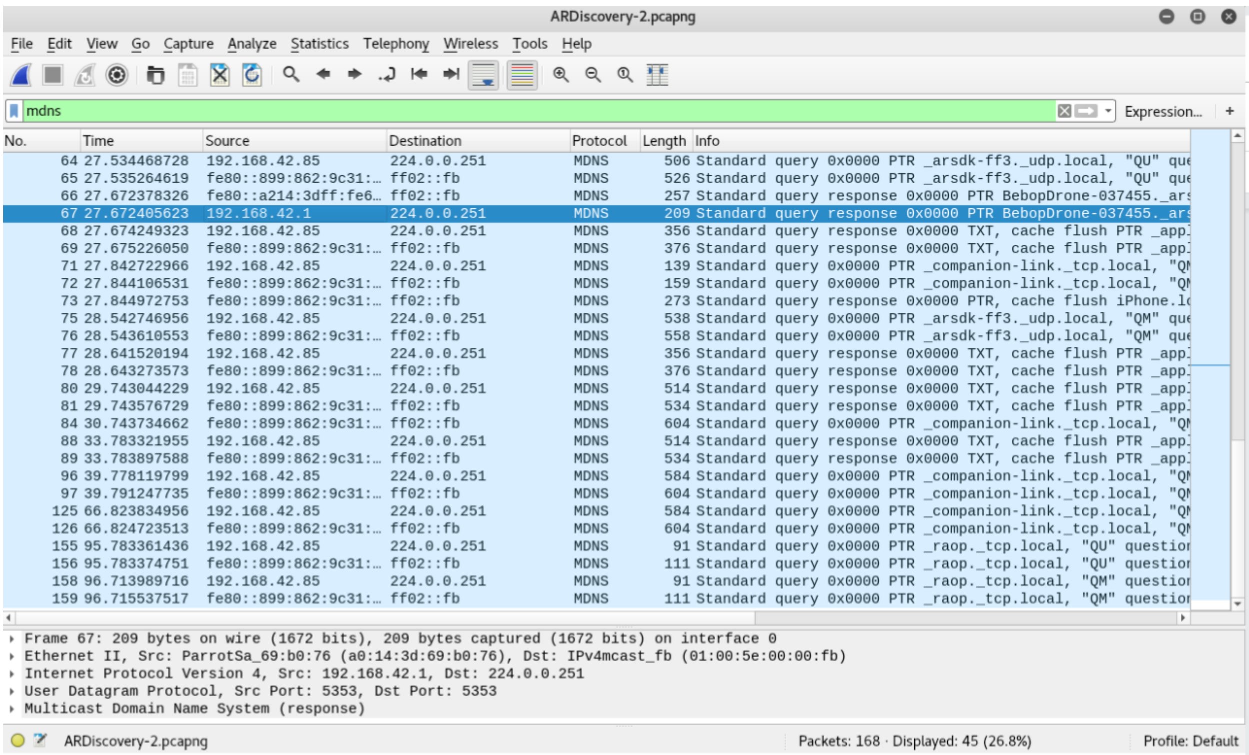
Looking at the man page of tcpreplay:



```
tcpreplay-edit(1)                    User Commands                    tcpreplay-edit(1)

NAME
       tcpreplay-edit - Replay network traffic stored in pcap files

SYNOPSIS
       tcpreplay-edit [-flags] [-flag [value]] [--option-name[[=| ]value]] <pcap_file(s)>

       tcpreplay is a tool for replaying network traffic from files saved with tcpdump or other tools which write pcap(3) files.

DESCRIPTION
       The basic operation of tcpreplay is to resend  all  packets  from  the input file(s) at the speed at which they were recorded, or
       a specified data rate, up to as fast as the hardware is capable.

       Optionally, the traffic can be split between two interfaces, written to files, filtered and edited in various ways, providing the
       means to test firewalls, NIDS and other network devices.

       For more details, please see the Tcpreplay Manual at: http://tcpreplay.appneta.com
```

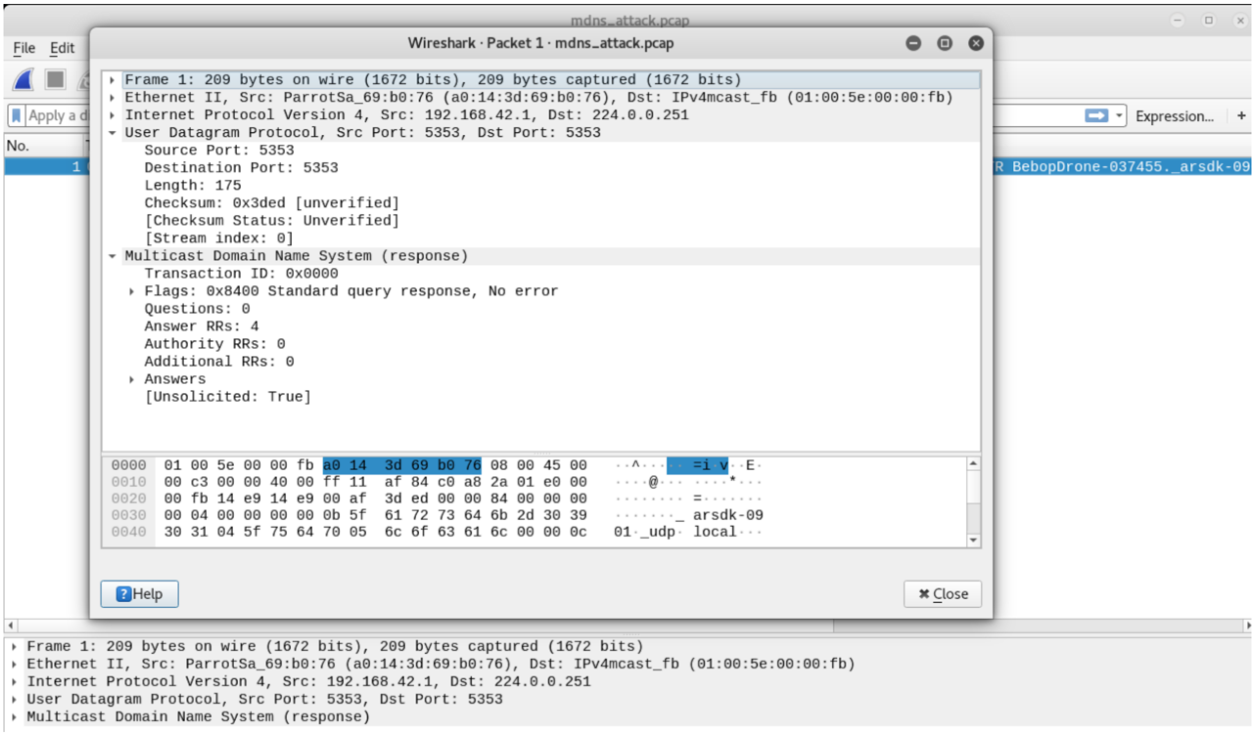
The video attached in the same email will explain how we conducted the MDNS DoS attack using tcpreplay, which replays network traffic stored in pcap files. Just to briefly explain, we first used the pcap file from 1e.

But, we have to only filter MDNS on wireshark first:

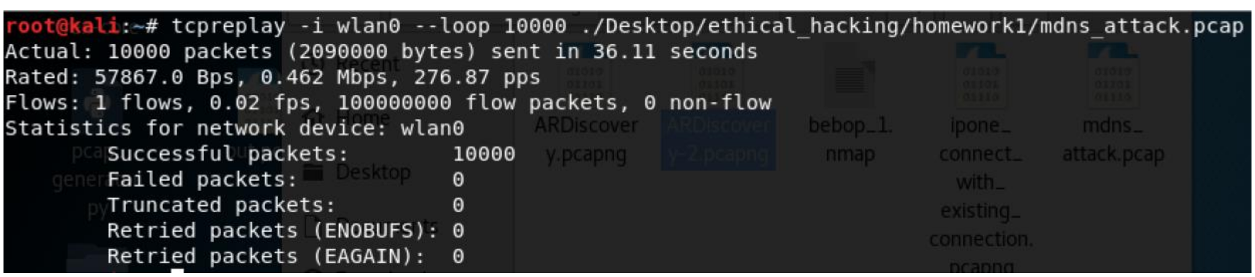


Then, we found a packet that was associated with the drone, as highlighted above. Next, we exported this single packet, which is a MDNS response packet, into another pcap file that will be used to conduct the DoS attack via tcpreplay.

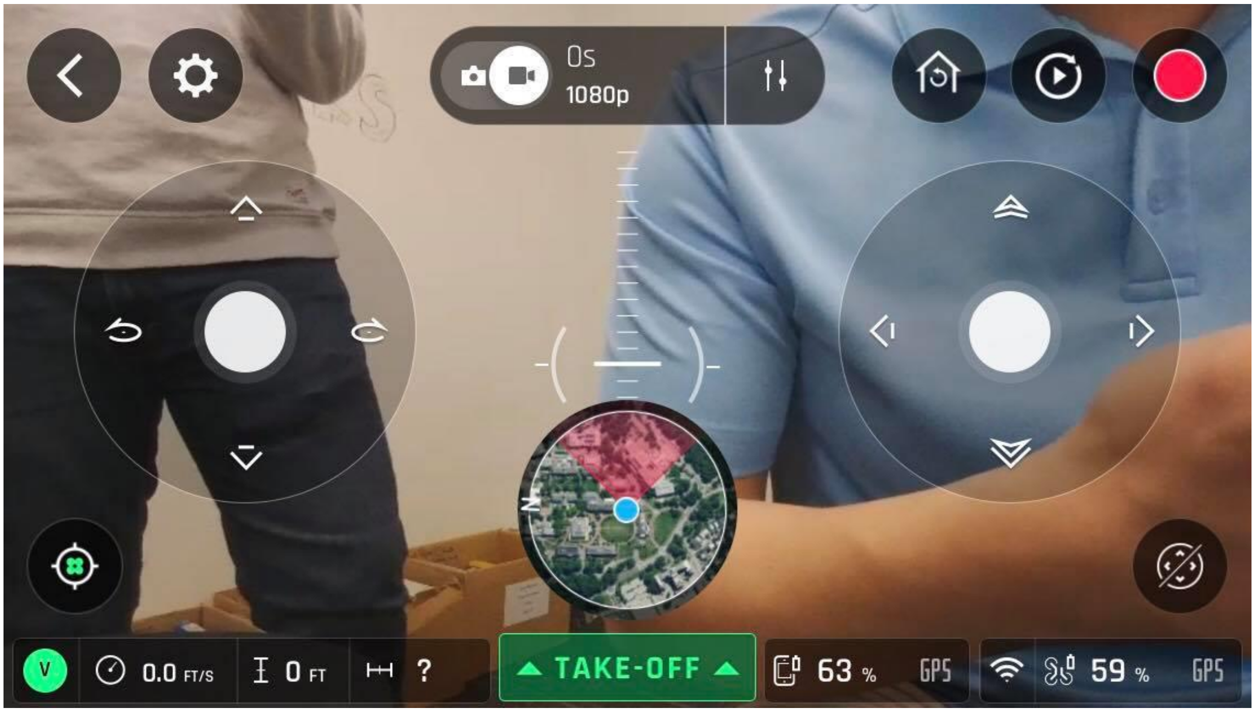
After exporting this single packet into a mdns_attack.pcap file:



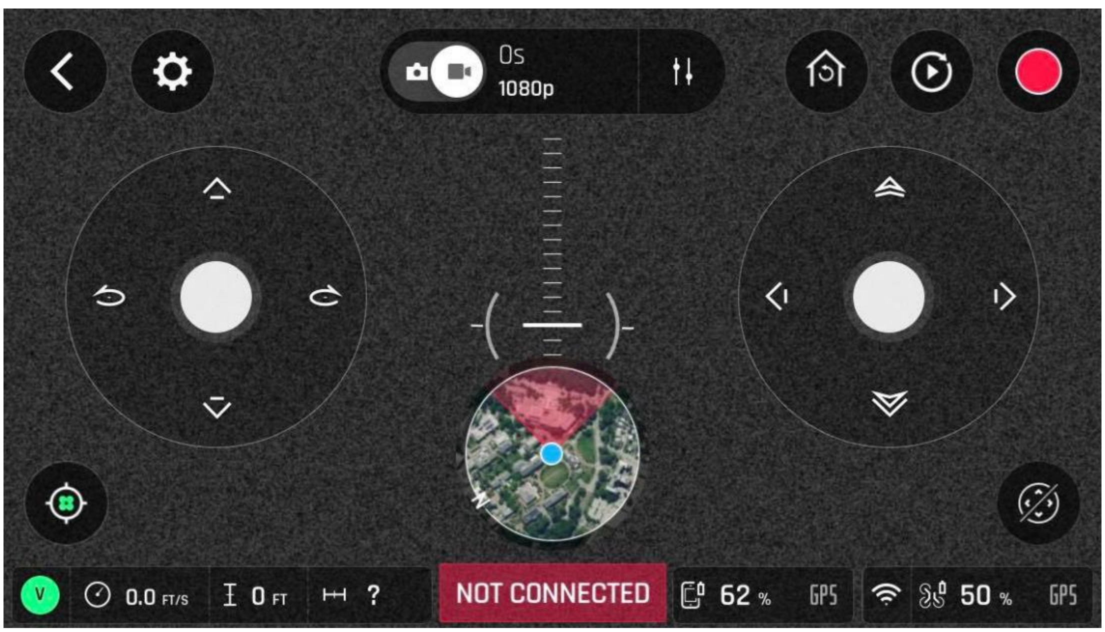
Now, using mdns_attack.pcap, we conducted this attack in Kali's terminal with an existing connection between my phone(controller) with the drone:



Initially, the phone showed this:



But, after a few seconds, we saw this on our phone:



This shows that we have successfully waged a DoS attack against the Bebop ARDiscovery process.